Deployment of learning techniques

Antonia Bertolino, Antonello Calabro, Sofia Cassel, Yu-Fang Chen, Falk Howar, Malte Isberner, Bengt Jonsson, Maik Merten, Patrizio Pelliccione, Bernhard Steffen, et al.

▶ To cite this version:

Antonia Bertolino, Antonello Calabro, Sofia Cassel, Yu-Fang Chen, Falk Howar, et al.. Deployment of learning techniques. [Research Report] 2012. hal-00695625

HAL Id: hal-00695625

https://inria.hal.science/hal-00695625

Submitted on 9 May 2012

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Emergent Connectors for

Eternal Software Intensive Networked Systems

ICT FET IP Project

Deliverable D4.3

Deployment of learning techniques

http://www.connect-forever.eu

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

Deliverable Number	:	D4.3
Title of Deliverable	:	Deployment of learning techniques
Nature of Deliverable	:	R
Dissemination Level	:	Public
Internal Version Number	:	1.0
Contractual Delivery Date	:	Jan. 2012
Actual Delivery Date	:	2012
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), Malte Isberner (TUDo), Bengt Jonsson (UU), Maik Merten (TUDo), Patrizio Pelliccione (UDA), Bernhard Steffen (TUDo), Massimo Tivoli (UDA)
Reviewer(s)	:	Massimo Tivoli (UDA), Gordon Blair (LANCS)

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. During Y2, work in* CONNECT *has focused on making conceptual advances that will enable the learning enabler to fulfill its purpose in the* CONNECT *architecture, in particular with respect to automatic learning of models. During Y3, the main challenges have been to further develop increasingly efficient techniques for automated learning of rich component models, building on the achievements during Y2, to integrate the learning enabler into the overall* CONNECT *architecture, and to integrate the learning and monitoring enablers.*
During Y3, we have, in a series of breakthroughs, develop and implemented techniques for efficient learning of rich behavioral models including data of various forms. First, we have worked out a much improved description of our canonical register automaton model (this work started during Y2), representing behaviors that combine both control and data. Second, on the basis of this canonical model we have developed a learning algorithm for behaviors that combine control and data. This learning algorithm can be systematically derived by exploiting the fact that the automaton model is based on a Nerode congruence. It represents a breakthrough in the area of automata learning, and outperforms previous approaches to learning models that combine control and data. Third, we have extended the canonical automaton model to cover behaviors that can manipulate data using an arbitrary set of relations, provided that the relations can be equipped with a certain structure. We have also developed and implemented an approach for using type analysis to enhance the power and efficiency of learning. During Y3, we have also revised our central tool for automata learning, the Next Generation LearnLib *(NGLL), in order to incorporate the newly developed concepts and learning algorithms in a structured way, as well as to make its use for new learning situations easier, and finally to integrate it into the overall* CONNECT *architecture. Finally, we have integrate the learning and monitoring enablers, so that it is possible to evaluate if the model correctness hypotheses are respected and also to learn the potential evolution of the system under test so to avoid that the corresponding automata become obsolete.*
During Y3, integration with other work packages has taken significant steps: the learning enabler has been integrated into the overall CONNECT *architecture, and the formalism for expressing learned models can be manipulated by WP2 and WP3 techniques.*

Document History

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

Document Review

Date	Version	Reviewer	Comment
Jan. 2012	1.0	Gordon Blair; Massimo Tivoli	
Feb. 2012	1.1	Valérie Issarny	

Table of Contents

LIST OF FIGURES 9

LIST OF TABLES 11

1 INTRODUCTION 13

1.1 The Role of Work Package 4 13
1.2 Brief Summary of Achievements in Y1 14
1.3 Brief Summary of Achievements in Y2 16
1.4 Review Recommendations 17
1.5 Challenges and Achievements during Y3 17
1.6 Overview of this Deliverable 19

2 REGISTER AUTOMATA WITH DATA 21

2.1 Register Automata 23
2.1.1 Data languages and register automata 23
2.1.2 Symbolic representation of data languages 25
2.1.3 Nerode congruence and canonical form 27
2.1.4 Comparison between different automata models 30
2.2 Register Automata for Richer Formalisms 31
2.2.1 Data languages and register automata 31
2.2.2 Symbolic representation of data languages 33
2.2.3 Nerode congruence and canonical form 38
2.2.4 Example: Seat-booking 40
2.2.5 Conclusions and future work 41

3 LEARNING REGISTER AUTOMATA WITH DATA 43

3.1 The Algorithm 43
3.1.1 Related work 43
3.1.2 Register automata and data languages 44
3.1.3 Active learning of canonical RAs 45
3.1.4 Example application 51
3.1.5 Conclusions 52
3.2 Integration of Register Automata in LearnLib 53

4 TYPE ANALYSIS FOR MODEL INFERENCE 55

4.1 Related Work 55
4.2 The Constituent Tools 56
4.2.1 StrawBerry 56

4.2.2 LearnLib and active automata learning 58
4.3 Motivating Example **59**
4.3.1 StrawBerry at work 61
4.3.2 LearnLib at work 62
4.4 The Integrated Approach **62**
4.5 Application to the Example and Discussion **64**
4.6 Experiments **65**
4.6.1 Case study subjects and experimental setup 65
4.6.2 Results 66
4.6.3 Threats to validity 66
4.7 Conclusions and Perspectives **66**

5 IMPLEMENTATION AND INTEGRATION OF THE LEARNING ENABLER 71
5.1 Conceptual Overview **71**
5.2 Structure of the Learning Enabler **72**
5.2.1 Alphabet generation and test driver 73
5.2.2 Result transformation 74
5.2.3 Model and data structure repository 75
5.3 Current Status **75**

6 INTEGRATION OF LEARNING AND MONITORING 77
6.1 Motivation **77**
6.1.1 GLIMPSE Infrastructure 78
6.2 GLIMPSE implementation **79**
6.3 Integration of GLIMPSE monitoring and the learning enabler **80**
6.4 Conclusions and perspectives **81**

7 CONCLUSION AND FUTURE WORK 85

BIBLIOGRAPHY 87

List of Figures

Figure 1.1: Overview of the CONNECT data-flow . . . 14

Figure 1.2: The Learning Enabler in the context of the connection phase architecture . . . 15

Figure 2.1: Partial model for a fragment of XMPP . . . 24

Figure 2.2: Partially specified prefix of minimal DCDT for the XMPP language . . . 27

Figure 2.3: Partial DURA model for a fragment of XMPP . . . 30

Figure 2.4: RA for the second-largest example . . . 32

Figure 2.5: DCDT for L_2 (words of length 3) . . . 37

Figure 2.6: RA for the seat booking example . . . 41

Figure 2.7: RA for the Blue web store example . . . 41

Figure 3.1: Partial model for a fragment of XMPP . . . 44

Figure 3.2: Counterexamples: a) new transition, b) new register, c) new permutation . . . 49

Figure 3.3: First and second hypothesis . . . 51

Figure 3.4: A modeled learning setup created in LearnLib Studio. The model is currently executed, with bold edges denoting the path of execution. The current query and its answer are made visible in the panel on the lower left side. . . . 53

Figure 4.1: Overview of the StrawBerry method . . . 57

Figure 4.2: Schematic view of a test driver for learning a reactive system. . . . 59

Figure 4.3: States of the dependencies automaton produced by StrawBerry . . . 59

Figure 4.4: Model created by StrawBerry. The edge labels are abbreviated for improved readability. . . 61

Figure 4.5: Integration of StrawBerry syntactic analysis (steps 1-2) and LearnLib (steps 5 and 6). Step 3 is a newly added feature. `wsimport` provides proxy classes to interact with the target system. . . 62

Figure 4.6: Excerpt of the setup specification for LearnLib generated by StrawBerry by means of syntactical analysis. . . . 68

Figure 4.7: Model created by LearnLib using the setup description created by StrawBerry. The edge labels are abbreviated for improved readability. . . . 69

Figure 5.1: The Learning Enabler in the context of the CONNECT enabler architecture 72

Figure 5.2: Data flow during operation of the Learning Enabler 73

Figure 5.3: Fragment of the learning setup 74

Figure 6.1: The GLIMPSE Architecture 78

Figure 6.2: The GlimpseBaseEvent Interface 81

Figure 6.3: The interaction between Learning and Monitoring 82

Figure 6.4: Integration of monitoring and learning 83

List of Tables

Table 3.1: Observation Table (only showing a subset of all prefixes) 51

Table 3.2: Experimental Results. 52

Table 4.1: Experiment results 65

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.

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 Chapter 5.1 of D1.1[1], as a system of various enablers that exchange information about the networked system to be constructed. The current system of enablers and how they interact is shown schematically 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 obtains interface descriptions from the Discovery Enabler, and creates a behavioral model (a Mealy machine) by interacting with the system using the Invocation Enabler. The resulting formal model is sent back to the Discovery Enabler. The process is illustrated in Figure 1.2. A more detailed description of this interaction can be found in Chapter 5 and in D1.3.

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 (see D4.2), 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 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,

[1] CONNECT Deliverable D1.1,'Initial Connect Architecture'

[2] CONNECT Grant Agreement, Annex I

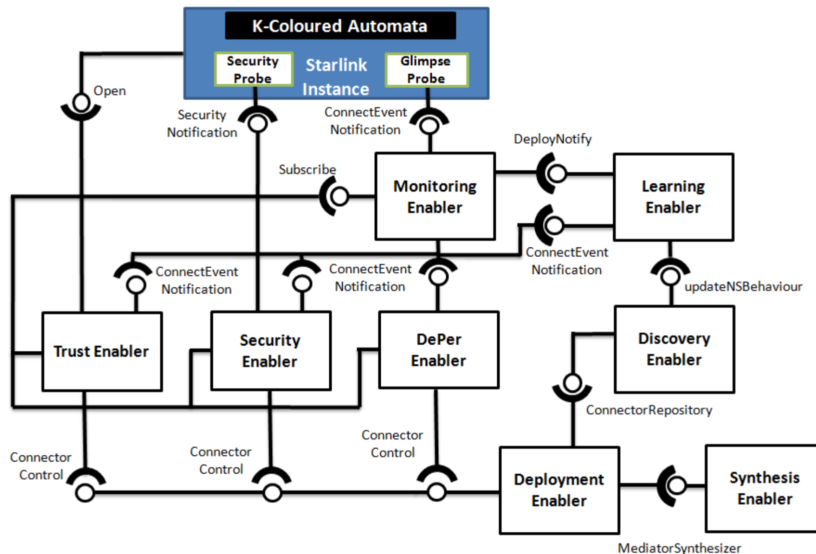

Figure 1.1: Overview of the CONNECT data-flow

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 3 .

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

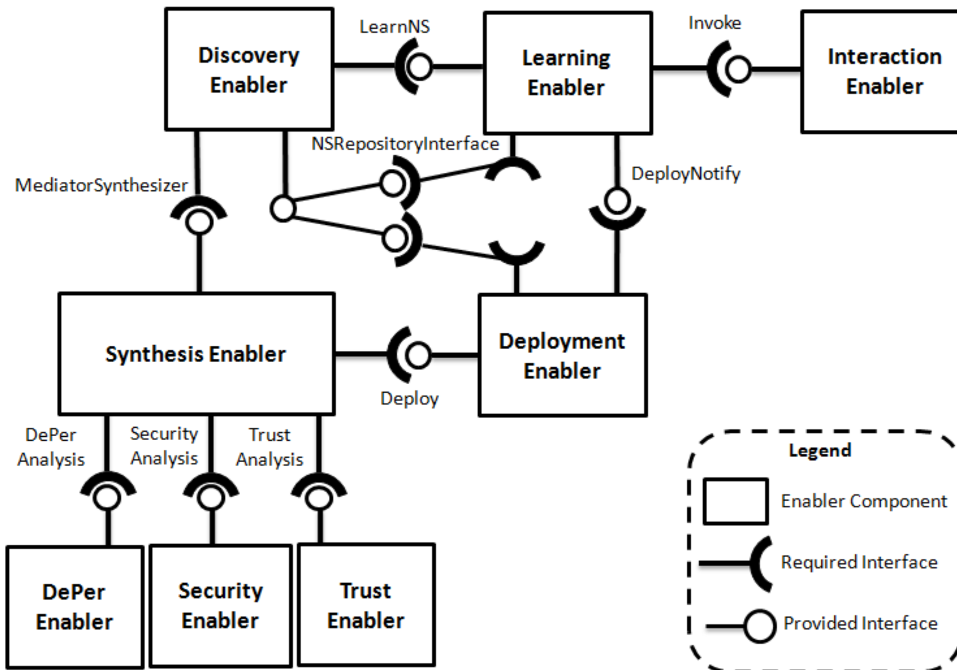

Figure 1.2: The Learning Enabler in the context of the connection phase architecture

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 [6], and of the new biometric passport [7]. Another technique was used to learn a model of a rather complex gateway protocol [19].

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 Brief Summary of Achievements in Y2

During Y2, work in CONNECT has focused on making conceptual advances that will enable the learning enabler to fulfill its purpose in the CONNECT architecture. The project made several important advances in this area. Briefly, the achievements were in the following directions.

Algorithmic improvements for effective learning: Active learning relies on performing a significant number of membership queries that explores the unknown behavior of a networked component. During Y2, we made significant progress towards making this exploration more efficient. We developed a learning algorithm that can be configured to have different characteristics wrt. resource consumption. In particular, it can be used to save a considerable amount of membership queries compared with classic algorithms. The power of our algorithm was confirmed in the ZULU challenge [25], which we won, in competition with several very strong groups in the language learning community.

We also addressed the challenge to make learning more efficient by limiting learning to those parts of component behavior that are needed for proper operation of the CONNECTed system. We developed 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.

Extending automata learning to rich models: State of the art learning algorithms work on rather unexpressive finite-state models. A major challenge in WP4 is to extend learning techniques to handle rich models including data of various forms. During Y2, we started to develop a novel foundation for learning automata with data. The motivation was the insight that active learning (e.g., using the L^* algorithm) is the well-known Nerode congruence on regular languages, which allows to identify words that lead to the same location in a canonical acceptor for some language. However, the literature contains very few definitions of Nerode congruences for languages with data. During Y2, we started to develop a novel canonical form for automata with data, formulated as *register automata*, i.e., finite-state machines extended with data variables, which is intuitive and succinct. This line of work has been brought to fruition in Y3.

Dynamic refinement of abstractions: During Y1 we identified abstractions as an important concept in the process of learning behavioral models of realistic systems. The work alluded to in the previous paragraph can be seen as a very systematically derived abstraction. But the abstraction technique is applicable in a quite general context to learn general classes of components using classical techniques such as L^*. One important part of such techniques is the ability to refine the abstraction when it is found to be inadequate. In Y2, we introduced 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.

Learning nonfunctional properties: During Y2, we addressed the problem of how the learning enabler is supposed to gather information about non-functional properties, primarily latencies. Achievements were made along two lines. One line is to consider latencies between interactions as a data parameter (e.g., in the form of time stamps), and use techniques for learning extended automata to infer them [33]. Another line of work is to augment an existing functional model with constraints on latencies between actions. We introduced and implemented the necessary extensions to learning technology to 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: During Y2, we initiated the implementation and integration of an infrastructure to continuously monitor the runtime behavior of the CONNECTed systems to respond to the growing needs of evolution and adaptation, and showed how it could be applied to parameterized models for various types of runtime analyses.

1.4 Review Recommendations

We here list comments and recommendatione from the Y2 review, together with our responses.

> *The reviewers would like to see learning demonstrated in the context of the testbed in the next period.*

Our response is that during Y3, the learning enabler has been developed so that it is able to interact with the other enablers in the implemented CONNECT architecture. Thus, the learning enabler can be demonstrated on whatever case studies are chosen for demonstration. In addition, the learning enabler has been tested on case studies that are intended to resemble components in the Terrorist Alert Scenario (now generalized in the context of GMES) and in other realistic scenarios (see, for example, Section 4.6 in this deliverable).

> *The relationship between the register automata used here and the interface automata used in WP2 should be clarified — for example, how are connectors synthesized using learning verified?*

In Chapters 2 and 4 of Deliverable 2.3, we present a version of interface automata, together with an extension of such automata with data. The learning techniques implemented in WP4 can be used to generate interface automata (cf. also [8]). Furthermore, the register automata, which we learn in Chapter 2, are a restricted form of interface automata. In D2.3 is described how the quotient operation can be used to synthesize interface automata which by construction perform an intended functionality.

> *Successful learning depends on the determinism of the system to be learned. The dependency of the interface behavior on non-functional properties is problematic for learning. The reason is that(1) dependency of functionality on non-functional properties is generally not part of the interface information, and (2) suppose it is,it may still be hard to learn a system in such a context.*

In many contexts, functional behavior depends on non-functional properties only to the extent of quiescence/nonquiescence. This non-functional property can in most cases be approximated with sufficient precision. For instance, the framework of testing developed by Tretmans [70] build on this precondition.

> *For the time being, only very simple data abstractions are considered — which is fine. On the long run, it will be necessary to aim at more general data abstractions.*

During Y3, we have generalized our work on register automata to richer formalisms: see Section 2.2 of this deliverable. As we demonstrate on several examples in that chapter, this generalization is able to handle a quite large number of practically occurring examples.

1.5 Challenges and Achievements during Y3

During Y3, the main challenges have been to further develop increasingly efficient techniques for automated learning of rich component models, building on the achievements during Y2, to integrate the learning enabler into the overall CONNECT architecture, and to integrate the learning and monitoring enablers. These challenges are reflected in the items planed for D4.3 in the Description of Work, from which we quote

- *Development of scalable learning algorithms*
- *Implementation of learning, and integration with existing tool (LearnLib)*
- *Combining learning and monitoring*

Below, we elaborate on the achievements with respect to these challenges.

Development of scalable learning algorithms: A major challenge for Y3 has been to develop techniques that can efficiently learn rich behavioral models including data of various forms. The modeling of both control and data aspects of a system's behavior is crucial for learning and synthesis of networked systems. For example, we may want to express that an entered password matches a previously registered one, that a sequence number falls in some interval, or that a user identity can be found in some specific group or set.

In Y1, we approached this problem through user-supplied abstraction modules, which allowed to use finite-state learning techniques to generate rich models with data. However, in order develop automated learning algorithms that can be implemented in a systematic way (e.g., in LearnLib), we need a new, canonical, representation of rich models. In Y2, we therefore started to develop a novel canonical form for automata with data, formulated as *register automata*, i.e., finite-state machines extended with data variables.

During Y3, this appraoch has been brought to fruition, leading to several significant achievements. First, we have worked out a much improved description of our canonical register automaton model, representing behaviors that combine both control and data. 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 Nerode congruence. Second, on the basis of this canonical model we have developed a learning algorithm for behaviors that combine control and data. This learning algorithm can be systematically derived by exploiting the fact that the automaton model is based on a Nerode congruence. It represents a breakthrough in the area of automata learning, and outperforms previous approaches to learning models that combine control and data. Third, we have extended the canonical automaton model to cover behaviors that can manipulate data using an arbitrary set of relations, provided that the relations can be equipped with a certain structure.

We have also developed and implemented an approach for using *Type Analysis* to enhance the power and efficiency of learning. An analysis infers types from interface descriptions, e.g., in WSDL, which are used to generate test harnesses for learning setups, and guide the selection of membership queries.

Implementation of learning: During Y3, the challenge has been to revise our central tool for automata learning, the *Next Generation LearnLib* (NGLL), in order to incorporate the newly developed concepts and learning algorithms in a structured way, as well as to make its use for new learning situations easier, and finally to integrate it into the overall CONNECT architecture. Concretely, the most notable extensions in Y3 have been (1) to implement the new learning algorithm for register automata: the fact that this could be done with modest effort shows the power of our register automata approach, (2) to improve the modeling and configuration in NGLL in order to ease the creation of learning processes, and (3) to integrate the Learning Enabler into the overall CONNECT architecture. Naturally, these extensions also led to further consolidation of the LearnLib framework.

Combining learning and monitoring: During Y3, the challenge has been to integrate the learning and monitoring enablers, so that it is possible to evaluate if the model correctness hypotheses are respected and also to learn the potential evolution of the system under test so to avoid that the corresponding automata become obsolete. To address this challenge, we propose a "Never-stop Learning" approach, where probes are inserted into the CONNECTor to collect information on actual run-time system interaction. The collected data is processed by the monitoring infrastructure and subsequently analyzed by LearnLib, possibly triggering additional learning effort.

Integration with other WPs: During Y3, integration with other work packages have taken significant steps. On the one hand, the learning enabler has been integrated into the overall CONNECT architecture, as described in Figure 1.2. On the other hand, the register automaton formalism that we have developed, and which is used to represent learned behaviors, fits well into the component algebra which has been developed by WP2 (see D2.3). More precisely, register automata can be seen as a restricted form of processes with data that are presented in Chapters 2 and 4 of Deliverable 2.3. Hence, learned models can be manipulated using the techniques developed in WP2, and also be used as a basis for mediator synthesis.

While the learning of non-functional properties was originally scheduled for D4.3, during the development process the focus shifted towards integrating learning and monitoring (planned for D4.4). Therefore, the former has been postponed, while a description of the integration between learning and monitoring is given in Chapter 6.

1.6 Overview of this Deliverable

This deliverable describes our progress and achievements in Y3. In this chapter, we survey our achievements in more detail while providing an overview of the material in this deliverable.

Automated learning of models with data In Section 2.1, we present a completely revised and conceptually simplified presentation of our novel canonical automaton model, based on register automata, that can easily be used to specify protocol or program behavior. More concretely, register automata are reminiscent of control flow graphs: they comprise a finite control structure, assignments, and conditionals, allowing to assign values of an infinite domain to registers (variables) and to compare them for equality. A major contribution is the definition of a canonical automaton representation of any language recognizable by a deterministic register automaton, by means of a Nerode congruence. Not only is this canonical form easier to comprehend than previous proposals, but it can also be exponentially more succinct than these. Key to the canonical form is the symbolic treatment of data languages, which overcomes the structural restrictions in previous formalisms, and opens the way to new practical applications, e.g., in automata learning.

In Section 2.2, we present an extension of the register automaton model to richer data languages. The extension is intended to preserve the advantages of the register automaton model: canonicity and succinctness. For very rich languages, these properties are very hard to preserve at the same time: cf. e.g., the situation for timed automata where canonicity has been obtained only by moving to a region-graph-like construction. However, the framework that we present in Section 2.2 can overcome these problems for data domains with an arbitrarily rich sect of operations, provided that it can be equipped with a certain structure. This turns out to be possible for the class of systems that are in the scope of CONNECT.

In Chapter 3, we present an extension of active automata learning to the register automaton model described in Section 2.1. Our active learning algorithm is unique in that it directly infers the effect of data values on control flow as part of the learning process. This effect is expressed by means of registers and guarded transitions in the resulting register automata models. Central to the inference of register automata is an involved three-dimensional treatment of counterexamples. This treatment can be regarded as an elaboration of an algorithmic pattern which was originally presented in [63] for learning regular languages. We have transferred it to Mealy machine learning in [69], and to learning with alphabet abstraction refinement in [38]. The application of our algorithm to a small example indicates the impact of learning register automata models: Not only are the inferred models much more expressive than finite state machines, but the prototype implementation also drastically outperforms the classic L^* algorithm, even when exploiting optimal data abstraction and symmetry reduction.

Type analysis for model inference In Chapter 4, we combine active automata learning with syntactic analysis of WSDL descriptions, which provides a means to infer types while learning black-box systems. In particular, we show how from interface descriptions test-harnesses and input alphabets for learning experiments can be generated in an automated fashion. The automated approach will serve as a basis for the practical application of the algorithm from Chapter 3 in the Learning Enabler.

Data aware test harnesses and interface alphabets for learning experiments are obtained by combining the StrawBerry tool, with LearnLib: In a first step a syntactic analysis of interface descriptions is performed by StrawBerry. Second, this analsys is used to generate a test harness and interface alphabet for learning by LearnLib. By combining these two tools, we no longer have to manually construct a bridge between the model level and the execution level when learning a target system. We evaluate the approach in a small case study comprising three web-services with promising results.

Integration of learning and monitoring In Chapter 6, we present the GLIMPSE monitoring infrastructure, and its integration with LearnLib. To achieve integrated learning and monitoring, we propose a "Never-stop Learning" approach. In particular, we equipped the CONNECTor with monitoring probes, which collect information on actual run-time system interaction. GLIMPSE then makes sense of the collected data, employing its complex event processor engine and rule-based delegation of observations. Transported over a shared message bus, the elaborated information is retrieved by the LearnLib and subsequently analysed. To implement this approach, we needed to enhance the GLIMPSE API to improve granularity of analysis increasing the detail of the information about the CONNECTor: instanceID, instanceExecutionID, eventID, eventInResponseToID. GLIMPSE is now at the third release (3.2) and we are now working to make it interacting with several ESB and Complex Event Processor. The communication with LearnLib is provided using ServiceMix and JMS technologies.

Implementation and integration of the Learning Enabler In Chapter 5, we describe the implementation and API of the learning enabler, alongside with its integration into the overall CONNECT architecture.

During Y3, our central tool, the *Next Generation LearnLib* for automata learning had to undergo further revisions in order to deal with newly developed concepts and to ease experimentation. An overview on LearnLib is given in [53]. Of the extensions that have been made to LearnLib, the most notable are:

Revision of the modeling paradigm in LearnLib Studio: LearnLib Studio is a graphical user interface to the LearnLib framework. Its modeling paradigm has recently undergone significant changes which greatly ease the creation of learning processes. For example, the configuration is mostly done on-the-fly during the learning phase itself, where in the old version a dedicated setup phase would precede the actual learning process.

Adaption to Register Automata: Significant effort was spent on implementing support for the automaton model described in Chapter 2, Register Automata, in order to handle data-independent systems with data from an infinite domain. This included in particular the implementation of the algorithms presented in Chapter 3 in the LearnLib framework.

Naturally, these extensions also led to further consolidation of the LearnLib framework. A recent version is available on `http://www.learnlib.de`.

2 Register Automata with Data

Within CONNECT, one of the goals for WP4 is to be able to fully learn networked systems. In order to do this, we need an automaton formalism that can be used to specify such a system, as well as a learning algorithm that is compatible with this formalism.

The ability to handle not only control, but also data aspects of a system's behavior plays a crucial role when learning and modeling networked systems. We need to be able to express different relations between data values and how they affect control flow. For example, we may want to express that a password entered matches a previously registered one, that a sequence number must be in some interval, or that a user identity can be found in some specific group or set.

Register automata are finite automata with a finite set of registers (aka state variables). A register automaton can process input symbols using a predefined set of operations (tests and updates) over input data and registers. Other specialized classes of automata augmented with data, such as timed automata [10], counter automata, and data-independent transition systems [47] have long been used for specification, verification, and testing [59].

The applicability of our register automaton formalism is illustrated by some examples: the *Blue* client (from D3.3), the XMPP protocol, and a seat-booking service (in addition to some more theoretical examples). The XMPP scenario is echoed later in Chapter 3 where we show how the protocol can be learned using active automata learning in LearnLib. The *Blue* client and seat-booking examples demonstrate how our Section 2.2 framework is able to deal with an extended set of relations, e.g., set membership. In CONNECT, the impact of our work is to take us one step further on the way to be able to fully learn networked systems, through the ability to handle different relations between data parameters.

Canonicity and succinctness In many applications, modeling and reasoning can be made much more efficient if automata can be transformed into a canonical form. Transformation into a canonical form is heavily used in verification, equivalence, and refinement checking, e.g., using (bi)simulation based criteria [46, 57]. 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 [12, 32, 63]. While for finite automata, there are standard algorithms for determinization and minimization, based on the Myhill-Nerode theorem [36, 55], it has proven difficult to carry over such constructions and define canonical forms for automata models over infinite alphabets, including timed automata [73]. Often, canonical forms are obtained at the price of (re-)encoding extensive information about the relation between parameter values in the state space (e.g., [50, 15]).

More recently, canonical automata based on the Myhill-Nerode theorem have been proposed for languages in which data values can be compared for equality [30, 15, 21], and also for inequality when the data domain is ordered (in [15, 21]). In these works, canonicity enforces restrictions on how data is stored in variables: two variables may not store the same data value, and in the ordered case each state enforces a fixed ordering between its variables. These restrictions often cause a blow-up in the number of states, since they require testing and encoding accidental as well as essential relations between data values in a word. For instance, a cross-product of two independent automata, representing, e.g., the interleaving of two independent languages, will result in a blow-up due to the recording of accidental relations between data values of the two languages.

In Section 2.1, we present a more succinct canonical automaton model, based on a Myhill-Nerode charactierization, for languages where data is compared for equality. Our construction does not impose restrictions such as uniqueness on stored variables, and allows to avoid representing accidental relations between data values. The result are register automata that are minimal in a certain class, and that can be exponentially more succinct than automata proposed previously [30, 15, 21]. These qualities are very important in many applications; we have, for instance, exploited this construction for active learning of data languages [37].

We present a Nerode congruence for register automata (RAs) that yields a canonical form. Key to this generalization of Nerode's right congruence [36, 55] to RAs is the symbolic treatment of data languages in a way that abstracts from concrete data values and rather concentrates on the relations between param-

eter values. This allows for the required flexibility, and also leads to a more elegant canonical form, which may even be exponentially more succinct than other suggested canonical forms. This is 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 automaton.

In Section 2.2 we also extend our canonical automaton model to data domains where data can be compared using an arbitrary set of relations. We consider data languages that are able to distinguish words using only these relations, and propose a form of register automata (RAs) that accept such data languages. 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. RAs can thus can be regarded as a simple programming language, with variables, conditions, and assignments.

To achieve succinctness, our construction must be able to filter out unnecessary tests between data values, and also produce the weakest possible guards that still make the necessary distinctions between data words. As an illustration, if our data domain is equipped with tests for equality and (ordered) inequality, then after processing three data values (say, d_1, d_2, d_3), it may be that the only relevant comparison between a fourth value d_4 and these three is whether $d_4 \leq d_1$ or not (i.e., all other comparisons are irrelevant to whether the data word is accepted). In previous automaton proposals, this would typically result in 7 different cases, representing all possible outcomes of testing d_4 against the three previous values. In our proposal, however, we take into account whether comparisons are relevant or not, resulting in only 2 cases.

It is a challenge to achieve such succinctness while maintaining canonicity. We approach this challenge by using a symbolic representation of data languages in the form of decision-tree-like structures, called *constraint decision trees*. The filtering of unnecessary guards is guided by a *merging structure*, which depends on the set of given relations on the data domain, and enables comparison of different cases while filtering out accidental relations. Under some conditions on this merging structure, which are satisfied when the relations on the data domain include, e.g., the inequality relation, or set membership, in addition to equality, we obtain the nontrivial result that our decision trees are minimal.

Our data structure (constraint decision tree), which is used to minimize the representation of data languages has superficial similarities with decision diagrams or BDDs. However, we cannot use the simple minimization techniques of, e.g., BDDs, since relations on the data domain typically impose asymmetries in the tree (for instance, the subtree following a failed equality test is richer than the subtree that follows a successful equality test). This is why we need to introduce a signature-specific merging structure to guide minimization.

By a non-technical analogy, we could compare the difference between previous canonical automata [30, 15, 21] and our canonical form to the difference between the region graph and zone graph constructions for timed automata.

Related Work An early work on generalizing regular languages to infinite alphabets is due to Kaminski and Francez [45], who introduced finite memory automata (FMA) that recognize languages with infinite input alphabets. Since then, a number of formalisms have been suggested (pebble automata, data automata, ...) that accept different flavors of data languages (see [65, 20, 18] for an overview). Many of these formalisms recognize data languages that are invariant under permutations on the data domain, corresponding to the ability to test for equality on the data domain. Much of the work focuses on non-deterministic automata and are concerned with closedness properties and expressiveness results of data languages. A different model, which avoids registers or variables and represents relations between data values in a different way is proposed by Grumberg et al. [34]., which avoids registers or variables

For applications that we have in mind, we are interested in canonical deterministic RAs that can be used to model the behavior of protocols or (restricted) programs. For the case, where equality is the only relation [30, 15, 21] present Myhill-Nerode theorems. Canonicity is achieved by restricting how state variables are stored, which prompted us to propose a more succinct construction in [23].

Extensions of Myhill-Nerorde theorems to more general sets of relations between data values are few. Benedikt et al. [15] and Bojanczyk et al. [21] propose models that employ restrictions on how variables can store data values. Such restrictions may lead to unintuitively large automata.

2.1 Register Automata

2.1.1 Data languages and register automata

In this section, we introduce formally the notions of data languages and register automata. While a very general definition of data languages would define them simply as sets of data words, for our modeling purposes, focus is on data languages that are closed under permutations on the data domain. Such languages are agnostic to the concrete identitiy of data values, which they all treat alike. With this restriction, data languages are ideal to describe the flow of data as required for an adequate modeling of systems, whose behavior does not depend on the data content they distribute.

We assume an unbounded domain D of data values and a set A of *actions*. Each action has a certain *arity* which determines how many parameters it takes from the domain D. A *data symbol* is a term of form $\alpha(d_1, \ldots, d_n)$, where α is an action with arity n, and $d_1, \ldots, d_n$ are data values in D. A *data word* is a (finite) sequence of data symbols. A *data language* is a set of data words, which is closed under permutations on D. We will often represent a data language as a mapping from the set of data words to $\{+, -\}$, e.g. accept and reject.

We will now present an automaton model that recognizes data languages. Assume a set of *formal parameters*, ranged over by $p_1, \ldots, p_n$, and a finite set of *variables* (or registers), ranged over by $x_1, \ldots, x_n$. A *parameterized symbol* is a term of form $\alpha(p_1, \ldots, p_n)$, consisting of an action α and formal parameters $p_1, \ldots, p_n$ (respecting the arity of α). A *guard* is a conjunction of equalities and inequalities (here, an inequality means a negated equality, e.g., $x_2 \neq p_3$) over formal parameters and variables. We write $\bar{p}$ for $p_1, \ldots, p_n$ and $\bar{d}$ for $d_1, \ldots, d_n$.

Definition 1. A *Register Automaton* (RA) is a tuple $\mathcal{A} = (L, l_0, X, \Gamma, \lambda)$, where

- L is a finite set of *locations*,
- $l_0 \in L$ is the *initial location*
- X maps each location $l \in L$ to a finite set $X(l)$ of variables, where $X(l_0)$ is the empty set,
- Γ is a finite set of *transitions*, each of which is of form $\langle l, \alpha(\bar{p}), g, \rho, l' \rangle$, where l is a *source location*, l' is a *target location*, $\alpha(\bar{p})$ is a parameterized symbol, g is a guard over $\bar{p}$ and $X(l)$, and ρ (the *assignment*) is a mapping from $X(l')$ to $X(l) \cup \bar{p}$ (intuitively, the value of $x \in X(l')$ is assigned to the value of $\rho(x)$), and
- $\lambda : L \mapsto \{+, -\}$ maps each location to either $+$ (accept) or $-$ (reject),

such that for any location l and action α, the disjunction of all guards g in transitions of form $\langle l, \alpha(\bar{p}), g, \rho, l' \rangle$ in Γ is equivalent to $true$ (i.e., $\mathcal{A}$ should be *completely specified*). □

Example: We model the behavior of a fragment of the XMPP protocol [5] as a running example (shown in Figure 2.1). XMPP is widely used in instant messaging. In our fragment of XMPP, a user can register an account (providing a username and a password), log in using this account, change the password, and delete the account. In the figure, arcs are labeled with actions, guards, and assignments. Actions and guards are written above the horizontal delimiter; assignments are written below it. Accepting locations (where the user is logged in) are denoted by two concentric circles. For example, the user Bob could register his account with the action register(`Bob`, `secret`) (providing his username and password), and then log in with the action login(`Bob`, `secret`). Once logged in, he could change his password to `boblovesalice` with the action pw(`boblovesalice`). (For reasons of brevity, several transitions are omitted.) □

A register automaton $\mathcal{A}$ classifies data words as either accepted or rejected. One way to describe how this is done is to define a state of $\mathcal{A}$ as consisting of a location and an assignment to the variables of that location. Then, one can describe how $\mathcal{A}$ processes a data word symbol by symbol: on each symbol, $\mathcal{A}$ finds a transition with a guard that is satisfied by the parameters of the symbol and the current assignment to variables; this transition determines a next location and an assignment to the variables of the new location. For the purposes of this section, it will be more convenient to use a different but

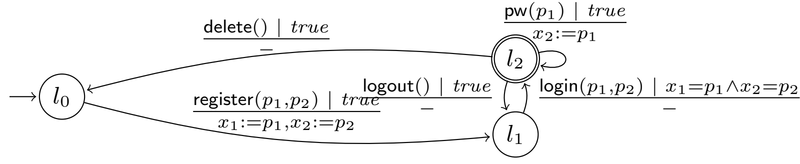

Figure 2.1: Partial model for a fragment of XMPP

equivalent definition. A *run* of $\mathcal{A}$ is defined as a pair consisting of a sequence of parameterized symbols and a guard over its formal parameters. Each run is extracted from some sequence of transitions, and is used to classify the data words that match its sequence of symbols and satisfy its guards. We will now discuss this in more detail.

A *parameterized word* w is a sequence of parameterized symbols in which all formal parameters are distinct; we assume a (re)naming scheme that avoids clashes. For a mapping ρ from a set X of variables, let $\tilde{\rho}$ denote the mapping obtained by extending the domain of X to include the set of formal parameters; these are all mapped to themselves (i.e., $\tilde{\rho}(x) = \rho(x)$ if x is a variable, and $\tilde{\rho}(p) = p$ if p is a formal parameter); we extend $\tilde{\rho}$ to expressions and guards in the natural way.

A sequence σ of transitions of $\mathcal{A}$ from l_0 to l_k is of form

$$\sigma \ = \ \langle l_0, \alpha_1(\bar{p}_1), g_1, \rho_1, l_1\rangle \quad \langle l_1, \alpha_2(\bar{p}_2), g_2, \rho_2, l_2\rangle \quad \cdots \quad \langle l_{k-1}, \alpha_k(\bar{p}_k), g_k, \rho_k, l_k\rangle \ ,$$

which starts in l_0 and ends in l_k. We define

- the *parameterized word* of σ as $\alpha_1(\bar{p}_1)\alpha_2(\bar{p}_2)\cdots\alpha_k(\bar{p}_k)$, and
- the *guard* of σ as $g = g_1 \wedge \tilde{\rho}_1(g_2 \wedge \tilde{\rho}_2(g_3 \wedge \tilde{\rho}_3(\cdots \wedge \tilde{\rho}_{k-1}(g_k))))$, i.e., essentially as the conjunction the guards $g_1, \ldots, g_k$ in σ, where the result of applying the mappings $\tilde{\rho}_1, \ldots, \tilde{\rho}_{k-1}$ is that each variable is replaced by the formal parameter from which it originally received its value.

A *run* of an RA $\mathcal{A}$ is a pair $\langle w, g\rangle$ such that w is the parameterized word and g is the guard of some sequence of transitions σ from the initial location l_0 to some l_k. A run is *accepting* if $\lambda(l_k) = +$. It is *rejecting* if $\lambda(l_k) = -$. (A run may be both accepting and rejecting if it can be extracted from two different sequences of transitions.)

A data word $\mathtt{w}_d = \alpha_1(\bar{d}_1)\cdots\alpha_k(\bar{d}_k)$ *satisfies* a run $\langle w, g\rangle$, denoted $\mathtt{w}_d \models \langle w, g\rangle$, if $\mathtt{w}_d$ has the same sequence of actions as w, and the parameters of $\mathtt{w}_d$ satisfy g in the obvious way (i.e., $d_{i_p} = d_{j_q}$ whenever $p_{i_p} = p_{j_q}$ is a conjunct in g, and $d_{i_p} \neq d_{j_q}$ whenever $p_{i_p} \neq p_{j_q}$ is a conjunct in g).

Example: The data word register(Bob, secret)login(Bob, secret) takes the automaton in Figure 2.1 from l_0 to l_2. The sequence σ of transitions is of the form $\langle l_0, \text{register}(p_1, p_2), true, \rho, l_1\rangle\langle l_1, \text{login}(p_3, p_4), (x_1 = p_3 \wedge x_2 = p_4), id, l_2\rangle$, where ρ is $(x_1 := p_1, x_2 := p_2)$ (note that parameters have been renamed to avoid clashes). The guard of σ is $g = (true \wedge \tilde{\rho}(x_1 = p_3 \wedge x_2 = p_4))$, i.e., $g = (p_1 = p_3 \wedge p_2 = p_4)$. Then $\langle \text{register}(p_1, p_2)\text{login}(p_3, p_4), g\rangle$ is a run of $\mathcal{A}$. □

An RA is *determinate* (called a DRA) if no data word satisfies both accepting and rejecting runs. A data word is *accepted* (*rejected*) by a DRA $\mathcal{A}$ if all runs that it satisfies are accepting (rejecting). We define $\mathcal{A}(\mathtt{w}_d)$ to be $+$ ($-$) if $\mathtt{w}_d$ is accepted (rejected) by $\mathcal{A}$. The language recognized by $\mathcal{A}$ is the set of data words that it accepts.

We have chosen to work with determinate, rather than deterministic, RAs, since a determinate RA can be easily transformed into a deterministic RA by strengthening its guards, and a deterministic RA, by definition, is also determinate. Our construction of canonical automata in Theorem 2 will generate determinate RAs which are not necessarily deterministic. They can easily be made deterministic, but this conversion can be done in several ways.

We call two variables $x_i, x_j \in X(l)$ in the same location of a DRA *independent* if the behavior of the DRA does not depend on the relation between the values of x_i and x_j. Technically, this means that (1) no guard of any transition may compare x_i and x_j when l is the source location, and (2) no combination of a

guard and an assignment may imply the equality of x_i and x_j when l is the target location of a transition. If all variables of a DRA are pairwise independent, i.e., no relation between variables influences the DRA's branching behavior, we refer to it as a *right-invariant* DRA (in reminiscence of the right-congruence that is represented in the locations of the automaton).

For the remainder of this section we will restrict our attention to right-invariant DRAs. Any DRA $\mathcal{A}$ can be transformed into an equivalent right-invariant DRA by expanding locations with dependent variables into sub-locations representing different valuations of the variables. This may, however, result in an exponential (in the number of variables) blow-up of the number of locations.

2.1.2 Symbolic representation of data languages

A given data language may be accepted by many different DRAs. In order to obtain a succinct, canonical form of DRAs, we will in this section define a canonical representation of data languages; in the next section we will describe how to derive canonical DRAs from this representation.

Our plan for this section is to first introduce a canonical form for runs of a DRA, called *constrained words*, which can only contain equalities (no inequalities) between parameters. Since now constrained words are less expressive than runs, each data word typically satisfies several constrained words. We therefore define a new notion of satisfaction between sets of constrained words and data words, which intuitively selects a "best matching" constrained word for a given data word. We can then use sets of constrained words, together with a classification of these words as "accepted" or "rejected", as a representation of data languages. We establish, as a central result (in Theorem 1), that any data language can be represented by a *minimal* set of constrained words. This minimal set will correspond to the set of runs of our canonical automaton, and will serve several purposes during automata construction:

(1) it will allow us to keep only the essential relations between data values and filter out inessential ("accidental") relations between data values, (2) from it, we can derive the parameters an automaton must store in variables after processing a data word, and (3) we can transform parts of it directly into transitions when constructing the canonical DRA.

Constrained words A *constraint* is a conjunction of equalities over formal parameters (i.e., without any inequalities). We always write constraints as ordered lists of equalities without parentheses (using associativity). We use $true$ to denote the empty constraint. For a parameterized word w, let $p \sqsubset_w p'$ denote that p and p' are formal parameters in w such that p occurs before p'.

A *constrained word* is a pair $\langle w, \varphi \rangle$ consisting of a parameterized word w and a constraint φ of form $p_1 = p'_1 \wedge p_2 = p'_2 \wedge \cdots \wedge p_k = p'_k$ over the formal parameters of w, in which the constraint φ satisfies the following conditions:

- $p_i \sqsubset_w p'_i$ for each $i = 1, \ldots, k$,
- $p'_1 \sqsubset_w \cdots \sqsubset_w p'_k$, and
- all $p_1, \ldots, p_k$ are distinct.

In other words, in each equality the arguments are ordered, the right-hand sides of φ are ordered, and each parameter occurs at most once as a left-hand side. Constrained words that differ only by permutation of formal parameters are regarded as equivalent. We can easily see that for each pair $\langle w, \varphi \rangle$ of a parameterized word w and constraint φ, there is a unique equivalent constrained word.

Since a constrained word is a special case of a run, we directly inherit a definition of satisfaction between data words and constrained words. Let *cw*$[\mathtt{w}_d]$ be the 'strongest' (w.r.t. number of equalities) constrained word that $\mathtt{w}_d$ satisfies, i.e., *cw*$[\mathtt{w}_d]$ contains exactly the equalities that $\mathtt{w}_d$ satisfies, put on the special form of constrained words. For example, *cw*$[\text{register}(\mathtt{Bob}, \mathtt{secret})\text{login}(\mathtt{Bob}, \mathtt{secret})] = = \langle \text{register}(p_1, p_2)\text{login}(p_3, p_4), p_1 = p_3 \wedge p_2 = p_4 \rangle$.

Constraint decision trees We will now define how sets of constrained words can be used to classify data words as accepted or rejected. This view of a set of constrained words is called a *constraint decision tree* (CDT). A CDT consists of a set of constrained words together with a mapping from this set to $\{+,-\}$, and classifies a data word by finding a "best matching" constrained word.

A set Φ of constrained words is *prefix-closed* if $\langle wv, \phi \wedge \psi\rangle \in \Phi$ implies $\langle w, \phi\rangle \in \Phi$ whenever $\langle w, \phi\rangle$ is a constrained word. (We recall that constraints are regarded as ordered lists of equalities, so that $\langle wv, \phi \wedge \psi\rangle$ is a constrained word when equalities appear exactly in the order defined by ϕ and ψ.) It is *extension-closed* if $\langle w, \varphi\rangle \in \Phi$ implies $\langle wv, \varphi\rangle \in \Phi$ for any parameterized word v. It follows that any non-empty prefix-closed and extension-closed set of constrained words also contains $\langle w, true\rangle$ for each parameterized word w.

Definition 2. A *constraint decision tree* (CDT) $\mathcal{T}$ pair $\langle Dom(\mathcal{T}), \lambda_{\mathcal{T}}\rangle$ where $Dom(\mathcal{T})$ is a non-empty prefix-closed and extension-closed set of constrained words, and $\lambda_{\mathcal{T}} : Dom(\mathcal{T}) \mapsto \{+,-\}$ is a mapping from $Dom(\mathcal{T})$ to $\{+,-\}$. ☐

For a constraint ψ, let $p \sqsubset_w \psi$ denote that $p \sqsubset_w p_j$ whenever $p_i = p_j$ is an equality in ψ (note that ψ may also be empty). We define a strict partial order $<$ on constrained words by defining $\langle w, \varphi\rangle < \langle w', \varphi'\rangle$ if $w = w'$ and there are constraints ϕ'', ψ, and ψ', such that

- φ is of form $\phi'' \wedge \psi$, and
- φ' is of form $\phi'' \wedge p{=}p' \wedge \psi'$, with $p' \sqsubset_w \psi$.

Example: For $w = \text{register}(p_1, p_2)\text{login}(p_3, p_4)$ we have $\langle w, p_1{=}p_3\rangle < \langle w, p_1{=}p_2\rangle$ since $p_1 = p_2$ is not present in $\langle w, p_1{=}p_3\rangle$, and since $p_2 \sqsubset_w (p_1{=}p_3)$. ☐

For a set Φ of constrained words, define a relation $\preceq_\Phi$ between constrained words in Φ and data words, by letting $\langle w, \varphi\rangle \preceq_\Phi \mathtt{w}_d$ iff $\langle w, \varphi\rangle$ is a maximal (w.r.t. $<$) constrained word in Φ such that $\mathtt{w}_d \models \langle w, \varphi\rangle$.

Intuitively, if $\langle w, \varphi\rangle \preceq_\Phi \mathtt{w}_d$, then $\langle w, \varphi\rangle$ can be viewed as a constrained word in Φ which "best matches" $\mathtt{w}_d$, obtained by adding equalities in φ from left to right. More precisely, given $\mathtt{w}_d$, we successively build $\langle w, \varphi\rangle$ as the limit of a sequence of constrained words in Φ. We start with $\langle w, true\rangle$, and whenever we have built $\langle w, \phi\rangle$ we extend it to some $\langle w, \phi \wedge p_i{=}p_j\rangle$, where $p_i{=}p_j$ is chosen such that $\mathtt{w}_d$ satisfies the equality $p_i{=}p_j$, and such that there is no other extension $\langle w, \phi \wedge p'_i{=}p'_j\rangle$ with $p'_j \sqsubset_w p_j$, where $\mathtt{w}_d$ satisfies $p'_i{=}p'_j$. If there is no such extension (of form $\langle w, \phi \wedge p_i{=}p_j\rangle$), we know that $\langle w, \varphi\rangle \preceq_\Phi \mathtt{w}_d$.

We call a CDT $\mathcal{T}$ *determinate* (a DCDT) if $\lambda_{\mathcal{T}}(\langle w, \varphi\rangle) = \lambda_{\mathcal{T}}(\langle w, \varphi'\rangle)$ whenever $\langle w, \varphi\rangle \preceq_{Dom(\mathcal{T})} \mathtt{w}_d$ and $\langle w, \varphi'\rangle \preceq_{Dom(\mathcal{T})} \mathtt{w}_d$ for some data word $\mathtt{w}_d$.

Example: A partially specified prefix of a DCDT for our running example can be seen in Figure 2.2. Here, the root node is the leftmost one, and the ordering $<$ is from top to bottom in the figure (i.e., lower nodes are bigger w.r.t. $<$). Let us illustrate the process of finding the maximal (w.r.t. $<$) constrained word $\langle w, \varphi\rangle$ that $\mathtt{w}_d = \text{register}(\mathtt{Bob}, \mathtt{secret})\text{login}(\mathtt{Bob}, \mathtt{secret})$ satisfies. The idea is to start from the root node and then successively add equalities to the constraint φ, until we have obtained the maximal one. We start with $\langle w, \varphi\rangle = \langle \text{register}(p_1, p_2)\text{login}(p_3, p_4), true\rangle$ and add the equality $p_1 = p_3$ which $\mathtt{w}_d$ satisfies. We can finally add the equality $p_2{=}p_4$, and we see that $\langle w, p_1{=}p_3 \wedge p_2{=}p_4\rangle \preceq_\Phi \mathtt{w}_d$. (In fact, we also see that $\langle w, p_1{=}p_3\rangle < \langle w, p_1{=}p_3 \wedge p_2{=}p_4\rangle$.) ☐

We can now define the data language represented by a DCDT, i.e., as a mapping from the set of data words to $\{+,-\}$.

Definition 3. For a DCDT $\mathcal{T}$, define $\lambda_{\mathcal{T}}(\mathtt{w}_d) = \lambda_{\mathcal{T}}(\langle w, \varphi\rangle)$ whenever $\langle w, \varphi\rangle \preceq_{Dom(\mathcal{T})} \mathtt{w}_d$. ☐

We now establish as a central result that for any data language λ there is a unique minimal DCDT that recognizes λ.

Theorem 1 (Minimal DCDT)**.** For any data language λ, there is a unique minimal DCDT $\mathcal{T}$ such that $\lambda = \lambda_{\mathcal{T}}$. ☐

By minimal, we mean that if $\mathcal{T}'$ is any other DCDT with $\lambda = \lambda_{\mathcal{T}'}$, then $Dom(\mathcal{T}) \subseteq Dom(\mathcal{T}')$. We will sometimes use the term λ-*essential* (constrained) words for members of $Dom(\mathcal{T})$ where $\mathcal{T}$ is the minimal DCDT with $\lambda = \lambda_{\mathcal{T}}$.

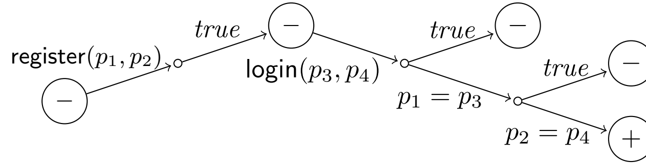

Figure 2.2: Partially specified prefix of minimal DCDT for the XMPP language

Proof. (Sketch) We prove Theorem 1 by defining how a minimal set $Dom(\mathcal{T})$ of constrained words can be constructed incrementally for any data language λ. We first extend the ordering $<$ so that it relates constrained words of different lengths, by defining $\langle w, \varphi \rangle < \langle w', \varphi' \rangle$ if w is a prefix of w' or vice versa and $\langle w'', \varphi \rangle < \langle w'', \varphi' \rangle$, where w'' is the longest of the two words w and w'. We construct $Dom(\mathcal{T})$ incrementally, starting with the set of constrained words of form $\langle w, true \rangle$, and then considering constrained words in increasing $<$-order (using the extended definition of $<$). Each such constrained word is added to $Dom(\mathcal{T})$ if it is needed in order to classify some data word correctly.

More precisely, consider a constrained word $\langle w, \varphi \rangle$, and let φ' be such that $\langle w, \varphi \rangle$ is of form $\langle w, \varphi' \wedge p = p' \rangle$. Let $\Phi^{<\langle w, \varphi \rangle}$ be the set of λ-essential constrained words that are less than (w.r.t. $<$) $\langle w, \varphi \rangle$. Then $\langle w, \varphi' \wedge p = p' \rangle$ is λ-essential if $\langle w, \varphi' \rangle$ is λ-essential (by prefix-closure), and if there is a data word $\mathtt{w}_d$, a constraint ψ, and some extension $w' = wv$ of w such that

- *cw*$[\mathtt{w}_d] = \langle w', \varphi' \wedge p = p' \wedge \psi \rangle$,
- $\langle w', \varphi'' \rangle \preceq_{\Phi^{<\langle w, \varphi \rangle}} \mathtt{w}_d$ for some λ-essential constrained word $\langle w', \varphi'' \rangle \in \Phi^{<\langle w, \varphi \rangle}$,
- and $\lambda(\langle w', \varphi'' \rangle) \neq \lambda(\mathtt{w}_d)$.

The incremental construction works, because only the set $\Phi^{<\langle w, \varphi \rangle}$ of λ-essential constrained words is needed to determine whether $\langle w, \varphi \rangle$ is λ-essential. □ □

Example: To illustrate the above procedure, we will partially sketch how to obtain the λ-essential constrained words of the form $\langle w_1, \phi \rangle$ where $w_1 = \text{register}(p_1, p_2)$, and of the form $\langle w_2, \phi \rangle$ where $w_2 = \text{register}(p_1, p_2)\text{login}(p_3, p_4)$. Initially, the words $\langle w_1, true \rangle$ and $\langle w_2, true \rangle$ are λ-essential.

We then consider constrained words in increasing $<$-order, beginning with a smallest constrained word, say $\langle w_2, p_2 = p_4 \rangle$. We find a data word $\mathtt{w}_d = \text{register}(\mathtt{Bob}, \mathtt{secret})\text{login}(\mathtt{Alice}, \mathtt{secret})$ such that *cw*$[\mathtt{w}_d] = \langle w_2, p_2 = p_4 \rangle$. We also find a λ-essential word $\langle w_2, true \rangle$ such that $\langle w_2, true \rangle \preceq_{\Phi^{<\langle w_2, p_2 = p_4 \rangle}} \mathtt{w}_d$. Since $\lambda(\mathtt{w}_d) = -$ and $\lambda(w_2, true) = -$ we see that $\mathtt{w}_d$ is already correctly classified and thus $\langle w_2, p_2 = p_4 \rangle$ is not λ-essential.

Next, we pick the constrained word $\langle w_2, p_1 = p_3 \rangle$ which is larger than $\langle w_2, p_2 = p_4 \rangle$ w.r.t. $<$. Consider the data word $\mathtt{w}'_d = \text{register}(\mathtt{Bob}, \mathtt{secret})\text{login}(\mathtt{Bob}, \mathtt{secret})$ such that *cw*$[\mathtt{w}'_d] = \langle w_2, p_1 = p_3 \wedge p_2 = p_4 \rangle$. We find a λ-essential word $\langle w_2, true \rangle$ such that $\langle w_2, true \rangle \preceq_{\Phi^{<\langle w_2, p_1 = p_3 \rangle}} \mathtt{w}'_d$. Since $\lambda(\mathtt{w}'_d) = +$ but $\lambda(\langle w_2, true \rangle) = -$ we see that $\mathtt{w}'_d$ is incorrectly classified and thus $\langle w_2, p_1 = p_3 \rangle$ is λ-essential.

We now test the constrained word $\langle w_2, p_1 = p_3 \wedge p_2 = p_4 \rangle$ with $\mathtt{w}'_d$. However, since the set of λ-essential constrained words has increased, we get a different λ-essential word $\langle w_2, p_1 = p_3 \rangle$ such that $\langle w_2, p_1 = p_3 \rangle \preceq_{\Phi^{<\langle w_2, p_1 = p_3 \wedge p_2 = p_4 \rangle}} \mathtt{w}'_d$. We see that $\lambda(\mathtt{w}'_d) = +$ but $\lambda(\langle w_2, p_1 = p_3 \rangle = -$, so $\langle w_2, p_1 = p_3 \wedge p_2 = p_4 \rangle$ is also λ-essential.

The λ-essential constrained words are now $\langle w_2, p_1 = p_3 \wedge p_2 = p_4 \rangle$, $\langle w_2, p_1 = p_3 \rangle$, $\langle w_2, true \rangle$, and $\langle w_1, true \rangle$. Note that these (together with the empty word) are exactly the constrained words in the DCDT of Figure 2.2. □

2.1.3 Nerode congruence and canonical form

In this section, we define a Nerode-type congruence on the set of constrained words of some (minimal) DCDT, which is then used to construct a succinct DRA that recognizes a data language.

Following standard Nerode, we will define equivalence of words w.r.t. suffixes. When splitting a constrained word into a prefix and a suffix, however, the equalities between parameters in the prefix and parameters in the suffix are also split. In the resulting RA, the “loose” connections will be represented by variables. These will be derived from the concept of *memorable* parameters, which is the set of parameters that need to be remembered after processing a prefix. Based on the minimal DCDT representation, this will guarantee that the number of variables stored by a canonical DRA is minimal. Similar definitions of data values that need to be remembered after a sequence of input symbols are also found in [16, 15].

In order for our canonical form to capture exactly the causal relations between parameters, we will allow memorable parameters to be re-shuffled when comparing words. Two words will be considered equivalent if they require equivalent parameters to be stored, independent of their ordering or their names.

Let us first see how a constrained word can be split into a prefix and a suffix. Consider a constrained word $\langle w, \varphi\rangle$, where w is a concatenation uv. We can make a corresponding split of φ as $\phi' \wedge \psi$, where the right-hand sides of equalities in ϕ' are parameters of u and the right-hand sides of equalities in ψ are parameters of v. Then $\langle u, \phi'\rangle$ (the prefix) is a constrained word, but $\langle v, \psi\rangle$ (the suffix) is in general not, since ψ refers to parameters that are not in v. We therefore define a *$\langle w, \phi\rangle$-suffix* as a tuple $\langle v, \psi\rangle$, where ψ is a constraint in which right-hand sides of equalities are parameters of v, and such that $\langle uv, \phi \wedge \psi\rangle$ (which we often denote $\langle u, \phi'\rangle; \langle v, \psi\rangle$) is a constrained word.

We define the *potential* of a constrained word $\langle w, \phi\rangle$, denoted $pot(\langle w, \phi\rangle)$, as the set of formal parameters in w that do not occur as the left argument of any equality in ϕ; for example, $pot(\langle \alpha_1(p_1, p_2)\alpha_2(p_3, p_4), p_1 = p_2 \wedge p_2 = p_3\rangle) = \{p_3, p_4\}$.

Definition 4 (Memorable)**.** Let λ be a data language, and let $\mathcal{T}$ be the minimal DCDT recognizing λ. The *λ-memorable parameters* of a constrained word $\langle w, \phi\rangle \in Dom(\mathcal{T})$, denoted $\text{mem}_\lambda(\langle w, \phi\rangle)$, is the set of parameters in $pot(\langle w, \phi\rangle)$ that occur in some $\langle w, \phi\rangle$-suffix $\langle v, \psi\rangle$ such that $\langle w, \phi\rangle; \langle v, \psi\rangle \in Dom(\mathcal{T})$. □

We are now ready to define our Nerode congruence on constrained words.

Definition 5 (Nerode Congruence)**.** Let λ be a data language, and let $\mathcal{T}$ be the minimal DCDT recognizing λ. We define the equivalence $\equiv_\lambda$ on constrained words by $\langle w, \phi\rangle \equiv_\lambda \langle w', \phi'\rangle$ if there is a bijection $\gamma : \text{mem}_\lambda(\langle w, \phi\rangle) \mapsto \text{mem}_\lambda(\langle w', \phi'\rangle)$ such that

- $\langle v, \psi\rangle$ is a $\langle w, \phi\rangle$-suffix with $\langle w, \phi\rangle; \langle v, \psi\rangle \in Dom(\mathcal{T})$ iff $\langle v, \gamma(\psi)\rangle$ is a $\langle w', \phi'\rangle$-suffix with $\langle w', \phi'\rangle; \langle v, \gamma(\psi)\rangle \in Dom(\mathcal{T})$, and then
- $\lambda(\langle w, \phi\rangle; \langle v, \psi\rangle) = \lambda(\langle w', \phi'\rangle; \langle v, \gamma(\psi)\rangle)$,

where $\gamma(\psi)$ is obtained from ψ by replacing all parameters in $\text{mem}_\lambda(\langle w, \phi\rangle)$ by their image under γ. □

Intuitively, two constrained words are equivalent if they induce the same residual languages modulo a remapping of their memorable parameters. The equivalence $\equiv_\lambda$ is also a congruence in the following sense. If $\langle w, \phi\rangle \equiv_\lambda \langle w', \phi'\rangle$ is established by the bijection $\gamma : \text{mem}_\lambda(\langle w, \phi\rangle) \mapsto \text{mem}_\lambda(\langle w', \phi'\rangle)$, then for any $\text{mem}_\lambda(\langle w, \phi\rangle)$-suffix $\langle v, \psi\rangle$ we have $\langle w, \phi\rangle; \langle v, \psi\rangle \equiv_\lambda \langle w', \phi'\rangle; \langle v, \gamma(\psi)\rangle$.

Example: In the data language that is accepted by the DRA of Figure 2.3, the word $\langle \text{register}(p_1, p_2)\text{login}(p_3, p_4)\text{pw}(p_5), p_1$ $p_3 \wedge p_2 = p_4\rangle$ and the word $\langle \text{register}(p_1, p_2)\text{login}(p_3, p_4), p_1 = p_3 \wedge p_2 = p_4\rangle$ are equivalent w.r.t. $\equiv_\lambda$. For the remapping $\gamma(p_4) = p_5$, and $\gamma(p_3) = p_3$ the residuals become identical. E.g., the suffix $\langle \text{logout}()\text{login}(p_6, p_7), p_3 = p_6 \wedge p_4 = p_7\rangle$, will become the suffix $\langle \text{logout}()\text{login}(p_6, p_7), p_3 = p_6 \wedge p_5 = p_7\rangle$ under remapping. Concatenation with the original words will lead to accepted words in both cases. □

Guard transformation We will introduce a transformation from suffixes to guards, which will be needed in Theorem 2 when constructing DRAs from DCDTs.

Let Φ be a set of constrained words, with $\langle w, \varphi\rangle \in \Phi$. We say that $p_i \neq p_j$ is an *implicit inequality* of $\langle w, \varphi\rangle$ w.r.t. Φ if φ is of form $\phi' \wedge \psi$ for some ψ with $p_j \sqsubset_w \psi$, and Φ contains a constrained word of form $\langle w, \phi' \wedge p_i = p_j \wedge \psi'\rangle$. Let *ineqs*$_\Phi(\langle w, \varphi\rangle)$ be the conjunction of all implicit inequalities of $\langle w, \varphi\rangle$ w.r.t. Φ. Define the guard $g_\Phi^{\langle w, \varphi\rangle}$ as $g_\Phi^{\langle w, \varphi\rangle} \equiv \varphi \wedge$ *ineqs*$_\Phi(\langle w, \varphi\rangle)$. Then, $g_\Phi^{\langle w, \varphi\rangle}$ has the property that $\mathtt{w}_d \models \langle w, g_\Phi^{\langle w, \varphi\rangle}\rangle$ iff $\langle w, \varphi\rangle \preceq_\Phi \mathtt{w}_d$

Example: Consider the DCDT from Figure 2.2. Let Φ contain $\langle w, true\rangle$, $\langle w, p_1 = p_3\rangle$, and $\langle w, p_1 = p_3 \wedge p_2 = p_4\rangle$, and let $w = \mathsf{register}(p_1, p_2)\mathsf{login}(p_3, p_4)$. Then $p_1 \neq p_3$ is an implicit inequality of $\langle w, true\rangle$, because $p_3 \sqsubset_w true$, and because $\langle w, p_1 = p_3\rangle$ contains $p_1 = p_3$. Similarly, $p_2 \neq p_4$ is an implicit inequality of $\langle w, p_1 = p_3\rangle$. We then obtain the guard $g_\Phi^{\langle w,true\rangle}$ as $p_1 \neq p_3$, the guard $g_\Phi^{\langle w,p_1=p_3\rangle}$ as $p_1 = p_3 \wedge p_2 \neq p_4$, and the guard $g_\Phi^{\langle w,p_1=p_3 \wedge p_2=p_4\rangle}$ as $p_1 = p_3 \wedge p_2 = p_4$. □

We now relate our Nerode congruence to DRAs.

Theorem 2 (Myhill-Nerode)**.** A data language λ is recognizable by a DRA iff the equivalence $\equiv_\lambda$ on λ-essential words has finite index.

Proof. If: The if-direction follows by constructing a DRA from a given $\equiv_\lambda$, as the DRA $\mathcal{A} = (\, locs, l_0, X, \Gamma, \lambda)$, where

- L is given by the finitely many equivalence classes of the equivalence relation $\equiv_\lambda$ on λ-essential words. For each equivalence class, we choose a representative λ-essential constrained word.
- l_0 is $[\langle \epsilon, true\rangle]_{\equiv_\lambda}$, with the empty word as representative element.
- X maps each location $[\langle w, \phi\rangle]_{\equiv_\lambda}$ with representative word $\langle w, \phi\rangle$ to the set $X([\langle w, \phi\rangle]_{\equiv_\lambda})$ of λ-memorable parameters of $\langle w, \phi\rangle$. Note that we here use parameters as variables.
- Γ is constructed as follows. For each location $l = [\langle w, \phi\rangle]_{\equiv_\lambda}$ in L with representative element $\langle w, \phi\rangle$ and each λ-essential one-symbol extension of $\langle w, \phi\rangle$ of form $\langle w, \phi\rangle; \langle \alpha(\bar{p}), \psi\rangle$, there is a transition in Γ of form $\langle l, \alpha(\bar{p}), g, \gamma, l'\rangle$, where
  - $l' = [\langle w, \phi\rangle; \langle \alpha(\bar{p}), \psi\rangle]_{\equiv_\lambda}$; let $\langle w', \phi'\rangle$ be the representative element of the equivalence class $[\langle w, \phi\rangle; \langle \alpha(\bar{p}), \psi\rangle]_{\equiv_\lambda}$,
  - γ is the bijection $\gamma : \mathsf{mem}_\lambda(\langle w', \phi'\rangle) \mapsto \mathsf{mem}_\lambda(\langle w, \phi\rangle; \langle \alpha(\bar{p}), \psi\rangle)$ which is used to establish $\langle w', \phi'\rangle \equiv_\lambda \langle w, \phi\rangle; \langle \alpha(\bar{p}), \psi\rangle$ in Definition 5,
  - g is obtained as g_Φ^ψ, where Φ is the set of all λ-essential extensions of $\langle w, \phi\rangle$ by the action α, i.e., the set of λ-essential words of form $\langle w, \phi\rangle; \langle \alpha(\bar{p}), \psi'\rangle$.
- $\lambda([\langle w, \phi\rangle]_{\equiv_\lambda}) = \lambda(\mathtt{w}_d)$ whenever $\langle w, \phi\rangle = cw[\mathtt{w}_d]$.

The constructed DRA is well defined: it has finitely many locations since the index of $\equiv_\lambda$ is finite, the initial location is defined as the class of the empty word, and λ is defined from λ for the representative elements of the locations. The transition relation is total and determinate. This is guaranteed by construction of guards from DCDTs, and by construction of DCDTs.

To complete this direction of the proof, we need to show that the constructed automaton $\mathcal{A}$ indeed recognizes λ. Consider an arbitrary sequence of transitions of $\mathcal{A}$, of form

$$\langle l_0, \alpha_1(\bar{p}_1), g_1, \rho_1, l_1\rangle \quad \cdots \quad \langle l_{k-1}, \alpha_k(\bar{p}_k), g_k, \rho_k, l_k\rangle \ ,$$

which generates a run of form $\langle \alpha_1(\bar{p}_1) \cdots \alpha_k(\bar{p}_k) \ , g\rangle$, where g is $g_1 \wedge \tilde{\rho}_1(\cdots \wedge \tilde{\rho}_{k-1}(g_k)))$. Let $w = \alpha_1(\bar{p}_1) \cdots \alpha_k(\bar{p}_k)$, and let φ be the ordered sequence of equalities in g (i.e., omitting inequalities). By construction, $\langle w, \varphi\rangle$ is a λ-essential constrained word such that g is equivalent to $g_{Dom(\mathcal{T})}^{\langle w,\varphi\rangle}$, which implies that $\mathtt{w}_d \models \langle w, g\rangle$ iff $\langle w, \varphi\rangle \preceq_{Dom(\mathcal{T})} \mathtt{w}_d$ for any data word $\mathtt{w}_d$. In summary, this implies that $\mathcal{A}$ correctly classifies data words that satisfy any of its runs.

Only if: For the only-if direction, we assume any (right-invariant) DRA that accepts λ. The proof idea then is to show that two λ-essential constrained words corresponding to sequences of transitions that lead to the same location are also equivalent w.r.t. $\equiv_\lambda$, i.e., that one location of a DRA cannot represent more than one class of $\equiv_\lambda$. This can be shown straight-forwardly using right-invariance. □ □

We get as a corollary result from the only-if direction of the proof that the automaton generated in the first part of this proof is in fact a minimal (in the set of locations) right-invariant DRA recognizing λ. As stated already, minimality of the DCDT representation guarantees that the automaton will also use a minimal number of variables.

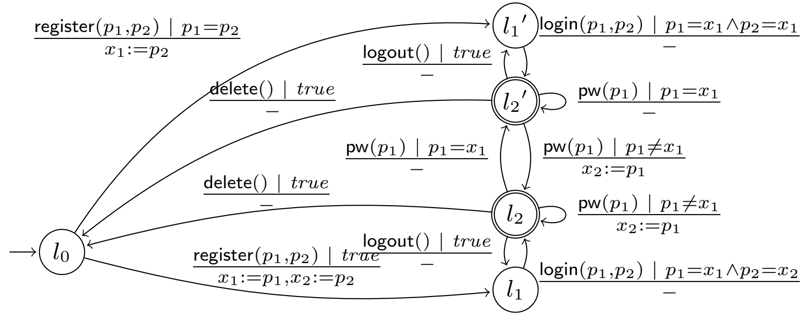

Figure 2.3: Partial DURA model for a fragment of XMPP

2.1.4 Comparison between different automata models

In this section we will compare our register automata to previously proposed formalisms. We will show that our models can be exponentially more succinct.

There are already proposals for DRAs that accept data languages, which, however, fail to be simple and do not exactly match the flavor of data languages we are using [45, 15]. For instance, in these automata, variables have to be unique, or can only be accessed in a queue-like fashion. A Myhill-Nerode-like theorem has been proposed for these data languages and automata [30, 15]. It is, however, formulated on the level of concrete data words. This makes it difficult to identify essential relations between parameters in the corresponding canonical form.

Both the design of the DRAs and the Nerode congruence on the level of data words thus require encoding information about accidental relations between parameters into the set of locations. This makes the models harder to understand and work with. We will show that in the worst case the resulting canonical models can be exponentially bigger than our canonical models.

Let us define a class of RAs that resembles the automata of [15]. 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 can never store the same data value. An RA is *ordered* (called an ORA) if state variables are ordered (we will use $<$ to represent this ordering), and data values are stored only in order of appearance. That is if x_i and x_j are two state variables with $x_i < x_j$, then in any reachable state, either x_j is undefined, or the transition at which x_i was assigned a value must coincide with or precede the transition at which x_j was last assigned a value. We will also define an OURA, which is both *ordered* and *unique-valued*. We will refer to the automata resulting from our Nerode congruence as DRAs. The automata of [15] correspond to deterministic OURAs (DOURAs).

In the worst case, there are two exponential blow-ups: between DRAs and DURAs, and between DURAs and DOURAs. The first exponential blow-up between DRAs and DURAs can be shown by constructing a DRA that can store n independent variables, while the corresponding DURA has to maintain in the set of locations which of the n variables have the same value. The second exponential blow-up between DURAs and DOURAs can be shown by constructing a DURA that allows random (write) access to n variables. The corresponding DOURA has to maintain in the set of locations the order in which the variables are written.

These blow-ups will not always be exponential. We will illustrate the difference between DURAs and our canonical form using our running example. Figure 2.3 shows a partial DURA model for the DRA from Figure 2.1. The DURA has to maintain if provided username and password (p_1, p_2 from $\text{register}(p_1, p_2)$) accidentally coincide. In this case this leads to replication of each location from which these two data values can be accessed, namely l_1 and l_2. A DOURA in this case would look the same as the DURA. Adding a primitive to change the username, however, would lead to another blow-up in the DOURA: the order in which username and password have been set would have to be encoded in the set of locations.

2.2 Register Automata for Richer Formalisms

2.2.1 Data languages and register automata

In this section, we introduce data languages and register automata that are parameterized on a set of (binary) relations, which may be used to distinguish between data words and may appear in guards of automata. Our notion of data language will treat words alike if they cannot be distinguished by the given set of relations. For instance, if the relations include only equality between data values, then data languages will be closed under permutations on the data domain. We will repeat some of the formalisms from the last section here, whenever they are necessary for the full understanding of our model.

Data languages We assume an unbounded domain D of data values, and a set $\mathcal{R}$ of binary relations on D. Our definitions and results generalize to the setting with several different domains (sorts) and sorted relations, but for simplicity this exposition will consider a single domain. We assume a set of *actions*. Each action has a certain *arity* which determines how many parameters it takes from the domain D. A *data symbol* is a term of form $\alpha(d_1, \ldots, d_n)$, where α is an action with arity n, and $d_1, \ldots, d_n$ are data values in D. A *data word* is a sequence of data symbols. Two data words $w_d = \alpha_1(d_1^1, \ldots, d_{n_1}^1) \cdots \alpha_k(d_1^k, \ldots, d_{n_k}^k)$ and $w'_d = \alpha_1(c_1^1, \ldots, c_{n_1}^1) \cdots \alpha_k(c_1^k, \ldots, c_{n_k}^k)$, are *equivalent*, denoted $w_d \approx_{\mathcal{R}} w'_d$, if $d_i^j \; R \; d_{i'}^{j'} \leftrightarrow c_i^j \; R \; c_{i'}^{j'}$ whenever $R \in \mathcal{R}$, for $1 \leq j, j' \leq k$ and $1 \leq i \leq n_j, 1 \leq i' \leq n_{j'}$. A *data language* is a set L of data words, which respects $\mathcal{R}$ in the sense that $w_d \approx_{\mathcal{R}} w'_d$ implies $w_d \in L \leftrightarrow w'_d \in L$. We will often represent a data language as a mapping from the set of data words to $\{+, -\}$, where $+$ stands for accept and $-$ for reject.

Register automata We will 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 α and formal parameters $p_1, \ldots, p_n$ (respecting the arity of α). A *guard* is a conjunction of negated and unnegated relations (from $\mathcal{R}$) between formal parameters or variables. We write $\bar{p}$ for $p_1, \ldots, p_n$ and $\bar{d}$ for $d_1, \ldots, d_n$.

Definition 6 (RA). *A* Register Automaton *(RA) is a tuple $\mathcal{A} = (L, l_0, X, \Gamma, \lambda)$, where*

- L *is a finite set of* locations*,*
- $l_0 \in L$ *is the* initial location*,*
- X *maps each location $l \in L$ to a finite set $X(l)$ of variables, where $X(l_0)$ is the empty set,*
- Γ *is a finite set of* transitions*, each of which is of form $\langle l, \alpha(\bar{p}), g, \rho, l' \rangle$, where l is a* source location*, l' is a* target location*, $\alpha(\bar{p})$ is a parameterized symbol, g is a guard over $\bar{p}$ and $X(l)$, and ρ (the* assignment*) is a mapping from $X(l')$ to $X(l) \cup \bar{p}$ (intuitively, the value of $x \in X(l')$ is assigned to the value of $\rho(x)$), and*
- $\lambda : L \mapsto \{+, -\}$ *maps each location to either $+$ (accept) or $-$ (reject),*

such that for any location l and action α, the disjunction of all guards g in transitions of form $\langle l, \alpha(\bar{p}), g, \rho, l' \rangle$ in Γ is equivalent to $true$ (i.e., $\mathcal{A}$ should be completely specified*).* □

Example.

Consider the data languageL_2, consisting of words where the last data value seen is the second-largest one in the entire data word. We will call this data language L_2. (For convenience, we can let data values be rational numbers.)

More precisely, a word is in L_2 either if the last data value is the second-largest data value in the word, or if it is equal to the largest data value in the word (whenever the largest data value occurs several times). For example, the data words `9 1 4 2 8` and `3 4 4` are both in L_2: in the first case, the last data value is equal to the largest data value in the word; in the second case, the last data value is smaller than the largest

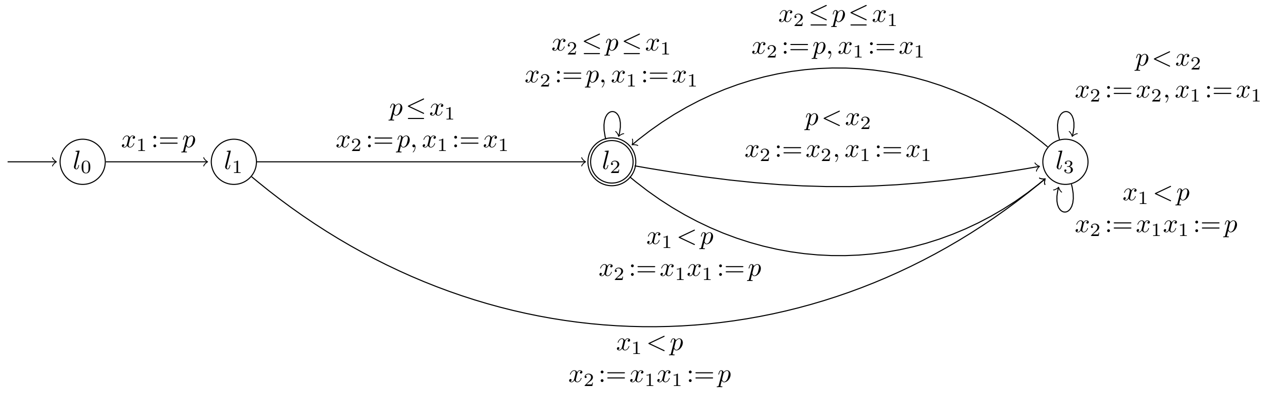

Figure 2.4: RA for the second-largest example

data value in the word. We define a set $\mathcal{R} = \{<, =\}$ of binary relations in order to compare parameters; we write $p_i \geq p_j$ for $\neg(p_i < p_j)$.

We can represent L_2 by the register automaton $\mathcal{A}_2$ in Figure 2.4. As usual, we let formal parameters represent concrete data values. For reasons of brevity, we have omitted the actions in this example, so each parameterized symbol is simply a formal parameter p. At each transition, a new parameter is passed to the system and compared to the existing location variables. Depending on the outcome of the comparison, the variables may be assigned new values, either p or the value of another variable.

The automaton always keeps the second-largest parameter seen so far in x_2 and the largest parameter seen so far in x_1. This means that in practice, whenever a new parameter is passed to the system, the register automaton need only distinguish between three cases: p is smaller than x_2, p is larger than x_1, or $x_2 \leq p \leq x_1$.

Throughout the paper, we will use L_2 and $\mathcal{A}_2$ as a running example. □

Semantics of a register automaton A register automaton $\mathcal{A}$ classifies data words as either accepted or rejected. A standard way to describe how this is done is to define a state of $\mathcal{A}$ as consisting of a location and an assignment to the variables of that location. Then, one can describe how $\mathcal{A}$ processes a data word symbol by symbol: on each data symbol, $\mathcal{A}$ finds a transition with a guard that is satisfied by the parameters of the symbol and the current assignment to variables; this transition determines a next location and an assignment to the variables of the new location. For the purposes of this section, it will be more convenient to use a different but equivalent definition. A *run* of $\mathcal{A}$ is defined as a pair consisting of a sequence of parameterized symbols and a guard over its formal parameters. Each run is extracted from some sequence of transitions, and is used to classify the data words that match its sequence of symbols and satisfy its guards. We will now present this in more detail.

A *parameterized word* w is a sequence of parameterized symbols in which all formal parameters are distinct; we assume a (re)naming scheme that avoids clashes. For a mapping ρ from a set X of variables, let $\tilde{\rho}$ denote the mapping obtained by extending the domain of X to include the set of formal parameters; these are all mapped to themselves (i.e., $\tilde{\rho}(x) = \rho(x)$ if x is a variable, and $\tilde{\rho}(p) = p$ if p is a formal parameter); we extend $\tilde{\rho}$ to expressions and guards in the natural way.

A sequence σ of transitions of $\mathcal{A}$ from l_0 to l_k is of form

$$\sigma \;=\; \langle l_0, \alpha_1(\bar{p}_1), g_1, \rho_1, l_1 \rangle \quad \cdots \quad \langle l_{k-1}, \alpha_k(\bar{p}_k), g_k, \rho_k, l_k \rangle \;,$$

which starts in l_0 and ends in l_k. We define

- the *parameterized word* of σ as $\alpha_1(\bar{p}_1) \cdots \alpha_k(\bar{p}_k)$, and
- the *guard* of σ as the conjunction

$$g_1 \wedge \tilde{\rho}_1(g_2 \wedge \tilde{\rho}_2(g_3 \wedge \tilde{\rho}_3(\cdots \wedge \tilde{\rho}_{k-1}(g_k)))) \;,$$

i.e., essentially as the conjunction the guards $g_1, \ldots, g_k$ in σ, where the result of applying the mappings $\tilde{\rho}_1, \ldots, \tilde{\rho}_{k-1}$ is that each variable is replaced by the formal parameter from which it originally received its value.

A *run* of an RA $\mathcal{A}$ is a pair $\langle w, g \rangle$ such that there is a sequence of transitions σ from the initial location l_0 to some l_k, where w is the parameterized word of σ and g is the guard of σ. A run is *accepting* if $\lambda(l_k) = +$. It is *rejecting* if $\lambda(l_k) = -$. (A run may be both accepting and rejecting if it can be derived from two different sequences of transitions.)

A data word $\mathtt{w}_d = \alpha_1(\bar{d}_1) \cdots \alpha_k(\bar{d}_k)$ *satisfies* a run $\langle w, g \rangle$, denoted $\mathtt{w}_d \models \langle w, g \rangle$, if $\mathtt{w}_d$ has the same sequence of actions as w, and the parameters of $\mathtt{w}_d$ satisfy g in the obvious way (i.e., $d_{i_p}\ R\ d_{j_q}$ whenever $p_{i_p}\ R\ p_{j_q}$ is a conjunct in g, and $\neg(d_{i_p}\ R\ d_{j_q})$ whenever $\neg(p_{i_p}\ R\ p_{j_q})$ is a conjunct in g).

Example: The data word 7 4 takes the automaton in Figure 2.4 from l_0 to l_2. The sequence σ of transitions is of the form $\langle l_0, p_1, true, (\rho_1), l_1 \rangle \langle l_1, p_2, (p_2 \leq x_1), \rho_2, l_2 \rangle$, where $\rho_1 = (x_1 := p_1)$ and $\rho_2 = (x_2 := p_2, x_1 := x_1)$ $\langle l_0, p_1, true, (x_1 := p_1), l_1 \rangle \langle l_1, p_2, (p_2 \leq x_1), \rho, l_2 \rangle$, where $\rho = (x_2 := p_2, x_1 := x_1)$ (note that parameters have been renamed to avoid clashes). The guard of σ is $g = (true \wedge \tilde{\rho}(p_2 \leq x_1))$, i.e., $g = (p_2 \leq p_1)$. Then $\langle p_1 p_2, g \rangle$ is a run of $\mathcal{A}$. □

An RA is *determinate* (called a DRA) if no data word satisfies both accepting and rejecting runs. A data word is *accepted* (*rejected*) by a DRA $\mathcal{A}$ if all runs that it satisfies are accepting (rejecting). We define $\mathcal{A}(\mathtt{w}_d)$ to be $+$ $(-)$ if $\mathtt{w}_d$ is accepted (rejected) by $\mathcal{A}$. The language recognized by $\mathcal{A}$ is the set of data words that it accepts.

We have chosen to work with determinate, rather than deterministic, RAs; the distinction is not important, since a determinate RA can be easily transformed into a deterministic RA by strengthening its guards, and a deterministic RA, by definition, is also determinate. Our construction of canonical automata in Theorem 4 will generate determinate RAs which are not necessarily deterministic. They can easily be made deterministic, but this conversion can be done in several ways.

We call two variables $x_i, x_j \in X(l)$ in the same location of a DRA *independent* if the behavior of the DRA does not depend on the relation between the values of x_i and x_j. Technically, this means that (1) no guard of any transition may compare x_i and x_j when l is the source location, and (2) no combination of a guard and an assignment may imply the equality of x_i and x_j when l is the target location of a transition. If all variables of a DRA are pairwise independent, i.e., no relation between variables influences the DRA's branching behavior, we refer to it as a *right-invariant* DRA (in reminiscence of the right-congruence that is represented in the locations of the automaton).

For the remainder of this section we will restrict our attention to right-invariant DRAs. Any DRA $\mathcal{A}$ can be transformed into an equivalent right-invariant DRA; this may, however, result in an exponential (in the number of variables) blow-up of the number of locations.

2.2.2 Symbolic representation of data languages

A given data language may be accepted by many different DRAs. In order to obtain a succinct, canonical form of DRAs, we will in this section define a canonical representation of data languages; in the next section we will describe how to derive canonical DRAs from this representation.

We first introduce a restricted form of runs of a DRA, called *constrained words*. We can then use sets of constrained words, together with a classification of these words as "accepted" or "rejected", as a representation of data languages. Such sets will be called *constraint decision trees*. We establish, as a central result (in Theorem 3), that any data language can be represented by a *minimal* set of constrained words, corresponding to a minimal constraint decision tree. This minimal set will correspond to the set of runs of our canonical automaton, and will serve several purposes during automata construction:

- it will allow us to keep only the essential relations between data values and filter out inessential (accidental) relations between data values,
- from it, we can derive the parameters an automaton must store in variables after processing a data word, and

- we can transform parts of it directly into transitions when constructing a canonical DRA.

Constrained words Recall that a parameterized word is a sequence $w = \alpha_1(p_1, \ldots, p_i) \cdots \alpha_k(p_j, \ldots, p_n)$ of parameterized symbols in which all formal parameters are distinct. From now on, we will assume that the formal parameters in a parameterized word are indexed from 1 and upwards, i.e., the sequence of formal parameters in w is $p_1, p_2, \ldots, p_n$. Let $|w|$ be the number of formal parameters in w. A *literal* is of the form $p_i\ R\ p_j$ or $\neg(p_i\ R\ p_j)$, where p_i and p_j are formal parameters and $R \in \mathcal{R}$. A *constraint* φ is a conjunction of literals. We use $true$ to denote the empty constraint.

A *constrained word* is a pair $\langle w, \varphi \rangle$ consisting of a parameterized word w and a consistent constraint φ over the formal parameters of w. If l is a literal of form $p_i\ R\ p_j$ or $\neg(p_i\ R\ p_j)$, then the *level of l in w* is the maximum of i and j. A constraint φ is *of level k in w* if it contains only literals of level k in w. Let $\varphi^w|_k$ denote the conjunction of all literals in φ of level k in w. Similarly, define $\varphi^w|_{<k}$ as the conjunction of all literals of φ of level smaller than k in w. Define $\varphi^w|_{>k}$ and $\varphi^w|_{\geq k}$ analogously.

If $\mathcal{R}$ includes the equality relation $=$, then the constraint φ in a constrained word $\langle w, \varphi \rangle$ must satisfy the restriction that whenever φ contains an equality $p_i = p_k$ or $p_i = p_k$ with $i < k$, then there must be no other occurrence of the parameter p_i in $\varphi^w|_{\geq k}$.

A data word w_d *satisfies* a constrained word $\mathfrak{w}$, denoted $w_d \models \mathfrak{w}$, if w and w_d have the same sequence of actions and the data values in w_d satisfies the constraint φ. When checking whether a data word w_d satisfies a constrained word $\langle w, \varphi \rangle$ (e.g., by an automaton), the literals in $\varphi^w|_k$ can be checked only when the k first data parameters in w_d have appeared. Thus, the literals in a constraint are checked in order of increasing level. Therefore, we will write constraints as ordered lists of literals without parentheses (using associativity), in order of increasing level.

A constraint φ is a *k-atom* if all its literals are of level at most k, and φ implies either $p_i\ R\ p_j$ or $\neg(p_i\ R\ p_j)$ for any $R \in \mathcal{R}$ whenever $i, j \leq k$. A constrained word $\langle w, \varphi \rangle$ is an *atom* if φ is a $|w|$-atom. It follows that a data language is uniquely defined by a classification of atoms, and that a data language L induces a mapping from atoms to $\{+, -\}$ defined by $L(\langle w, \varphi \rangle) = L(w_d)$ whenever $w_d \models \langle w, \varphi \rangle$.

Example. Let $w = p_1,\ p_2,\ p_3$ be a parameterized word, and let $\phi = p_1 \leq p_3 \leq p_2$. Then $\mathfrak{w}$ is a constrained word. Let $w_d = 3\ 7\ 4$ be a data word. We can easily see that $w_d \models \mathfrak{w}$. We also note that $\mathfrak{w}$ is a 3-atom. □

Constraint decision trees Having defined the notion of satisfaction between data words and constrained words, we will now define how sets of constrained words can be used to classify data words as accepted or rejected. This view of a set of constrained words is embodied by a so-called *constraint decision tree* (CDT). A CDT consists of a set of constrained words together with a mapping from this set to $\{+, -\}$.

A set Φ of constrained words is *prefix-closed* if $\langle wv, \varphi \rangle \in \Phi$ implies $\langle w, \varphi^w|_{\leq |w|} \rangle \in \Phi$. A set Φ is *extension-closed* if for any $\langle w, \varphi \rangle \in \Phi$ with $|w| = k$, and any action α of arity n, we have that the disjunction $\psi_1 \vee \ldots \vee \psi_m$ is equivalent to $true$, where $\{\psi_1, \ldots, \psi_m\}$ is the set of constraints of of level k such that $\langle w\alpha(p_{k+1}, \ldots, p_{k+n}), \varphi \wedge \psi_i \rangle \in \Phi$ for $i = 1, \ldots, m$.

Definition 7 (CDT). *A* constraint decision tree *(CDT) $\mathcal{T}$ is a pair $\langle Dom(\mathcal{T}), \lambda_{\mathcal{T}} \rangle$ where $Dom(\mathcal{T})$ is a non-empty prefix-closed and extension-closed set of constrained words, and $\lambda_{\mathcal{T}} : Dom(\mathcal{T}) \mapsto \{+, -\}$ is a mapping from $Dom(\mathcal{T})$ to $\{+, -\}$.* □

A CDT $\mathcal{T}$ is *determinate* (called a DCDT) if $\lambda_{\mathcal{T}}(\langle w, \varphi \rangle) = \lambda_{\mathcal{T}}(\langle w, \varphi' \rangle)$ whenever $\mathtt{w}_d \models \langle w, \varphi \rangle$ and $\mathtt{w}_d \models \langle w, \varphi' \rangle$ for some data word $\mathtt{w}_d$. A DCDT defines a language $\lambda_{\mathcal{T}}$ defined by $\lambda_{\mathcal{T}}(\mathtt{w}_d) = \lambda_{\mathcal{T}}(\langle w, \varphi \rangle)$ whenever $\mathtt{w}_d \models \langle w, \varphi \rangle$.

Our goal is that each language is represented by a canonical DCDT. This canonical DCDT is dependent on imposing a structure on the order in which literals can appear in constrained words

We will now describe a construction of a canonical DCDT which defines a given data language. Our goal is to construct a canonical DCDT $\mathcal{T}$ which is also minimal (with some restrictions).

Constructing a canonical DCDT Let, as a first attempt, $\mathcal{T}$ consist of all atoms. Then, if D is the set of rational numbers, and $\mathcal{R}$ contains the relations $<$ and $=$, the constrained words in $Dom(\mathcal{T})$ of the form $\langle a(p_1)b(p_2)c(p_3), \varphi\rangle$ would be such that φ completely specifies some ordering between p_1, p_2, and p_3. However, if the data language is insensitive to the ordering between p_2 and p_3, or does not distinguish the case $p_2 < p_3$ from $p_2 = p_3$, then we would like $Dom(\mathcal{T})$ not to contain all atoms, but maybe a constrained word of form $\langle a(p_1)b(p_2)c(p_3), p_1 < p_2 \wedge p_2 \leq p_3\rangle$ rather than both atoms $\langle a(p_1)b(p_2)c(p_3), p_1 < p_2 \wedge p_2 < p_3\rangle$ and $\langle a(p_1)b(p_2)c(p_3), p_1 < p_2 \wedge p_2 = p_3\rangle$. We would somehow like to 'merge' them.

Merging of atoms can be performed if they have the same continuations. However, the atoms $\langle a(p_1)b(p_2)c(p_3), p_1 < p_2 \wedge p_2 < p_3\rangle$ and $\langle a(p_1)b(p_2)c(p_3), p_1 < p_2 \wedge p_2 = p_3\rangle$ do *not* have the same continuations. For instance, the constraint $p_1 < p_2 \wedge p_2 < p_3$ can be extended by the constraint $p_2 < p_4 \wedge p_4 < p_3$ of level 4, whereas $p_1 < p_2 \wedge p_2 = p_3$ does not have such an extension.

To overcome this hurdle, we introduce an ordering $\sqsubseteq$ between extensions of an atom. This ordering determines which extension will be used to define the different cases. For instance, we would order the two extensions $p_2 < p_3$ and $p_2 = p_3$ in the above example by $p_2 < p_3 \sqsubseteq p_2 = p_3$. This means that the continuations after $p_1 < p_2 \wedge p_2 = p_3$ will be checked against the continuations of $p_1 < p_2 \wedge p_2 < p_3$ in order to determine whether these atoms should be merged into $p_1 < p_2 \wedge p_2 \leq p_3$.

Let us now describe the structure that must be predefined in order to define a canonical DCDT. We assume that the $\mathcal{R}$ of binary relations on D has been fixed.

Definition 8. *A* merging structure *is a mapping $\mathcal{M}$ which to each $k-1$-atom φ over $p_1, \ldots, p_{k-1}$ assigns a pair $\langle \Psi, \sqsubseteq\rangle$, where*

- Ψ *is a set of constraints of level k, called* branches*, such that $\varphi \wedge \psi$ is a k-atom for each $\psi \in \Psi$, and such that*
$$\bigvee_{\psi \in \Psi} \psi \equiv true$$
i.e., the disjunction of all branches in Ψ is equivalent to $true$.
- $\sqsubseteq$*, the* merging relation*, is a partial order on Ψ.* □

Intuitively, the branches represent the "most constrained" guards that completely characterize how a new parameter p_k may be related to existing parameters $p_1, \ldots, p_{k-1}$, provided that they satisfy φ. Thus, for a data word that satisfies φ, the processing of p_k yields a unique branch. The intention is that also data words that satisfy other constraints than φ may use the branching structure provided after φ. Hence the disjunction of all branches should be equivalent to $true$.

Note that different branches need not be mutually exclusive, i.e., in general it is not necessary that $\psi_i \wedge \psi_j$ is false. However, $\varphi \wedge \psi_i \wedge \psi_j$ is equivalent to $false$, since $\varphi \wedge \psi_i$ and $\varphi \wedge \psi_j$ are different atoms.

A merging structure imposes a structure on atoms, which is used to obtain canonical DCDTs by merging branches into constraints that are not atoms. Let φ be a $k-1$-atom and $\mathcal{M}(\varphi) = \langle \Psi, \sqsubseteq\rangle$. For any level-$k$ constraint g, define the *support of g after φ*, denoted *supp*$^\varphi(g)$, as the set of branches $\psi \in \Psi$ such that $(\varphi \wedge \psi) \rightarrow (\varphi \wedge g)$. Then any level-$k$ constraint g represents a merging of branches in the sense that if *supp*$^\varphi(g) = \{\psi_1, \ldots, \psi_k\}$ then
$$(\varphi \wedge g) \leftrightarrow (\varphi \wedge (\psi_1 \vee \cdots \vee \psi_k)) \ .$$

Definition 9. *A merging structure $\mathcal{M}$ is* admissible *if whenever $\mathcal{M}(\varphi) = \langle \Psi, \sqsubseteq\rangle$ for a $k-1$ atom φ, then for each k-level constraint g, it holds that*

- *supp$^\varphi(g)$ contains a unique minimal branch (wrp. to $\sqsubseteq$), called the* principal branch *of g, and*
- *if supp$^\varphi(g)$ and supp$^\varphi(g')$ have the same principal branch, then there is a level-k constraint g'' such that supp$^\varphi(g'') =$ supp$^\varphi(g) \cup$ supp$^\varphi(g')$ (Here we have $(\varphi \wedge g'') \leftrightarrow (\varphi \wedge (g \vee g'))$).* □

Define a set of k-level constraints to be *conformant* with a pair $\langle \Psi, \sqsubseteq\rangle$, if it is the set of guards with maximal support that are obtained from some transitive subset of $\sqsubseteq$.

The concept of principal branch can be extended to arbitrary constraints as follows. For a CDT $\mathcal{T}$ and an admissible merging structure $\mathcal{M}$, define the *principal atom* of any constraint φ over $p_1, \ldots, p_k$ as an atom, which is defined inductively as follows:

- The principal atom of the empty constraint over the empty tuple of parameters is the empty atom
- If φ is a constraint over $p_1, \ldots, p_k$, then let φ' be the principal atom of $\varphi^{\varphi}|_{\leq k-1}$ over $p_1, \ldots, p_{k-1}$, and let $\mathcal{M}(\varphi')$ be $\langle \Psi, \sqsubseteq \rangle$. Then the principal atom of φ over $p_1, \ldots, p_k$ is the atom $\varphi' \wedge \psi$, where ψ is the minimal (wrp. to $\sqsubseteq$) branch in Ψ.

Example Consider the case where $\mathcal{R}$ is $\{=\}$. Then our merging structure will assign to each k and atom which states that $p_1, \ldots, p_{k-1}$ are all different the pair $\langle \Psi, \sqsubseteq \rangle$, where

- Ψ consists of the constraint $p_1 \neq p_k \wedge \cdots \wedge p_{k-1} \neq p_k$ and the $k-1$ constraints of form $p_i = p_k$ for $i = 1, \ldots, k-1$, and
- $\sqsubseteq$ states that $(p_1 \neq p_k \wedge \cdots \wedge p_{k-1} \neq p_k) \sqsubseteq p_i = p_k$ for $i = 1, \ldots, k-1$, but does not relate $p_i = p_k$ with $p_j = p_k$ for $i \neq j$.

Consequently, each guard g with nontrivial support will have $p_1 \neq p_k \wedge \cdots \wedge p_{k-1} \neq p_k$ as principal branch. Also note that the equality branches are not mutually exclusive, unless one assumes that $p_1, \ldots, p_{k-1}$ are all different. □

Example Let us next consider the case where D is the set of rational numbers, and $\mathcal{R}$ is $\{<, =\}$ (it is important not to let D be some set of integers, since that case is more complicated). An atom over $p_1, \ldots, p_{k-1}$ will specify some order between $p_1, \ldots, p_{k-1}$, let us say $p_1 < \cdots < p_{k-1}$. Then the merging structure will assign $\langle \Psi, \sqsubseteq \rangle$, where

- Ψ contains constraints of form
  - $p_i < p_k < p_{i+1}$ for $i = 1, \ldots, k-2$, and
  - $p_k = p_i$ for $i = 1, \ldots, k-1$, and
  - the constraints $p_k < p_1$, and $p_{k-1} < p_k$.
- The relation $\sqsubseteq$ is the transitive closure of the relations
  - $p_i < p_k < p_{i+1} \sqsubseteq p_{i-1} < p_k < p_i$
for $i = 2, \ldots, k-2$, and
$p_1 < p_k < p_2 \sqsubseteq p_k < p_1$, and
$p_{k-1} < p_k \sqsubseteq p_{k-2} < p_k < p_{k-1}$,
  - $p_{i-1} < p_k < p_i \sqsubseteq p_k = p_i$ for $i = 2, \ldots, k-1$, and
$p_k < p_1 \sqsubseteq p_k = p_1$. □

Our next aim is to show that any data language can be represented by a canonical DCDT, which is also minimal under some (mild) conditions on the merging structure. Let us first say how a merging structure constrains the DCDTs that we consider.

Definition 10. *Let $\mathcal{T}$ be a CDT, and let $\mathcal{M}$ be an admissible merging structure. Then $\mathcal{T}$ is $\mathcal{M}$-*conformant *if for any $\langle w, \varphi \rangle \in Dom(\mathcal{T})$, such that $\langle \Psi, \sqsubseteq \rangle = \mathcal{M}(\varphi')$, where φ' is the principal atom of φ, the set of constraints g such that $\varphi \wedge g$ extend $\langle w, \varphi \rangle$ in $\mathcal{T}$ conforms to $\langle \Psi, \sqsubseteq \rangle$.* □

Definition 11. *An admissible merging structure $\mathcal{M}$ is* interpolating *if whenever $\mathcal{M}(\varphi) = \langle \Psi, \sqsubseteq \rangle$ and $\psi_1, \psi_2, \psi_3 \in \Psi$ satisfy $\psi_1 \sqsubseteq \psi_2 \sqsubseteq \psi_3$, then if θ is a $\varphi \wedge \psi_3$-suffix which is consistent with $\varphi \wedge \psi_1$, then θ is also consistent with $\varphi \wedge \psi_2$.* □

We now establish as a central result that for any data language L there is a unique minimal DCDT that recognizes L.

Theorem 3 (Minimal DCDT)**.** Let $\mathcal{M}$ be an admissible merging structure, which is also interpolating. Then for any data language L, there is a unique minimal $\mathcal{M}$-conformant DCDT $\mathcal{T}$ such that $L = \lambda_{\mathcal{T}}$. □

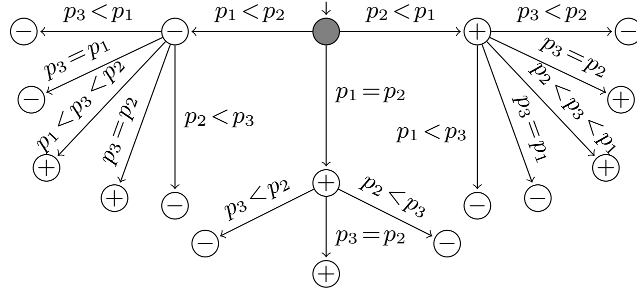

Figure 2.5: DCDT for L_2 (words of length 3)

By minimal, we mean that if $\mathcal{T}'$ is any other $\mathcal{M}$-conformant DCDT with $L = \lambda_{\mathcal{T}'}$, then any constrained word in $Dom(\mathcal{T}')$ is contained in a constrained word in $Dom(\mathcal{T})$.

Proof. We prove Theorem 3 by a construction, which constructs the minimal DCDT for L. The construction starts bottom up from leaves in the tree, starting from $\sqsubseteq$-minimal atoms. Such a construction can only be performed by assuming a bounded length of words that are classified by L. We will therefore assume a maximal length n of data words and construct a minimal "truncated" DCDT $\mathcal{T}_n$, which correctly classifies data words of length at most n. We will then show that $\mathcal{T}_n$ "grows monotonically" with increasing n, so that $\mathcal{T}$ can be taken as a limit of the trees $\mathcal{T}_n$.

So, let L_n be the restriction of L to data words of length at most n. Our construction of a canonical DCDT $\mathcal{T}_n$ is recursive, following the tree structure. The main construction constructs the subtree of $\mathcal{T}_n$, which is rooted at an atom φ at level k. Let this CDT be denoted $\mathcal{T}_n^{\langle w,\varphi\rangle}$. Its domain $Dom(\mathcal{T}_n^{\langle w,\varphi\rangle})$ is a set of constrained words of form $\langle w, \varphi \wedge \psi\rangle$ of length at most n, which it classifies using the function $\lambda_{\mathcal{T}_n^{\langle w,\varphi\rangle}}$. Let $\langle \Psi, \sqsubseteq\rangle = \mathcal{M}(\varphi)$. The construction of $\mathcal{T}_n^{\langle w,\varphi\rangle}$ proceeds as follows.

1. Initialize Ψ_0 to Ψ and $Dom(\mathcal{T}_n^{\langle w,\varphi\rangle})$ to $\emptyset$.

2. Let ψ be the smallest (acc. to $\sqsubseteq$) branch in Ψ_0. Recursively construct $\mathcal{T}_n^{\varphi\wedge\psi}$.

3. Now let Ψ_1 be the set of branches $\psi' \in \Psi_0$ such that $\psi \sqsubseteq \psi'$ and such that whenever $\langle w, \varphi \wedge \psi' \wedge \varphi'\rangle$ is an atom and $\langle w, \varphi \wedge \psi \wedge \varphi''\rangle \in Dom(\mathcal{T}_n^{\varphi\wedge\psi})$ is a constrained word in $Dom(\mathcal{T}_n^{\varphi\wedge\psi})$ such that $(\varphi \wedge \psi' \wedge \varphi') \rightarrow \varphi''$, then $L(\langle w, \varphi \wedge \psi' \wedge \varphi'\rangle) = L(\langle w, \varphi \wedge \psi \wedge \varphi''\rangle)$. (Intuitively, this condition says that any atom which extends $\varphi \wedge \psi'$ agrees with the classification already performed after $\varphi \wedge \psi$, which was done in step 2.)

4. Let g_ψ now be a maximal guard such that *supp*$^\varphi(g_\psi) \subseteq \Psi_1$, and extend $Dom(\mathcal{T}_n^{\langle w,\varphi\rangle})$ by all constrained words of form $\langle w, \varphi \wedge g_\psi \wedge \varphi'\rangle$ such that $\langle w, \varphi \wedge \psi \wedge \varphi'\rangle \in \mathcal{T}_n^{\varphi\wedge\psi}$.

5. Remove *supp*$^\varphi(g_\psi)$ from Ψ_0. If then Ψ_0 becomes empty, we are finished, otherwise go back to step 2.

The guard which is constructured at step 4 is a maximal guard. Namely suppose that g'_ψ would be any other guard whose support overlaps with that of g_ψ. Then, since $\mathcal{M}$ is interpolating, it follows that all branches in *supp*$^\varphi(g_\psi) \cup$ *supp*$^\varphi(g'_\psi)$ are in Ψ_1. By repeating this argument inductively, we conclude that $\mathcal{T}_n^{\langle\varepsilon,true\rangle}$ is a minimal DCDT for L_n, which we can take as $\mathcal{T}_n$.

We must next show that the restriction of $\mathcal{T}_{n+1}$ to constrained words of length at most n is larger than $\mathcal{T}_n$, in the sense that each constrained word in $Dom(\mathcal{T}_{n+1})$ of length at most n is included in some constrained word in $Dom(\mathcal{T}_n)$. This can again be proven from the assumption that $\mathcal{M}$ is interpolating. It follows that we can take $\mathcal{T}$ as the limit of $\mathcal{T}_n$ for increasing n, i.e., letting $Dom(\mathcal{T})$ include all constrained words that are in all $Dom(\mathcal{T}_n)$ for all n larger than some threshold. ☐

Example:

Figure 2.5 depicts a full DCDT for words of length at most 3 in our running example. The root node is gray in the figure. From the root node, we start by testing the first two parameters against each other. There are three outcomes, represented by three nodes. Each of these nodes represent a constrained word that is either accepted (denoted $+$ in the figure) or rejected (denoted $+$ in the figure). We continue to nodes at level 3, testing the third parameter against any previous ones. This decision tree can be minimized according to the above algorithm, by merging some branches. For example, the branches $p_1 < p_3 < p_2$ and $p_3 = p_2$ can be represented as $p_1 < p_3 \leq p_2$ and the branches $p_3 < p_1$, $p_3 = p_1$, and $p_3 < p_2$ can all be represented as $\not(p_1 < p_3 \leq p_2)$.

2.2.3 Nerode congruence and canonical form

In this section, we define a Nerode-type congruence on the set of constrained words of some (minimal) DCDT, which is then used to construct a succinct DRA that recognizes a data language.

Following standard Nerode, we will define equivalence of words w.r.t. suffixes. When splitting a constrained word into a prefix and a suffix, however, the equalities between parameters in the prefix and parameters in the suffix are also split. These “loose” connections will be represented by variables in the resulting RA. These will be derived from the concept of *memorable* parameters, which is the set of parameters that need to be remembered after processing a prefix. Based on the minimal representation as DCDT, this will guarantee that the number of variables stored by a canonical DRA is minimal. Similar definitions of data values that need to be remembered after a sequence of input symbols can also be found in [16, 15].

In order for our canonical form to capture exactly the causal relations between parameters, we will allow memorable parameters to be permuted. when comparing words. Two words will be considered equivalent if they require equivalent parameters to be stored, independent of their ordering or their names.

Let us first define how a constrained word can be split into a prefix and a suffix. Consider a constrained word $\langle w, \varphi \rangle$, where the parameterized word w is a concatenation uv, and u has k parameters. We can make a corresponding split of φ as $\varphi^{\varphi}|_{\leq k} \wedge \varphi^{\varphi}|_{>k}$. Then $\langle u, \varphi^{\varphi}|_{\leq k} \rangle$ (the prefix) is a constrained word, but $\langle u, \varphi^{\varphi}|_{>k} \rangle$ (the suffix) is in general not, since $\varphi^{\varphi}|_{>k}$ refers to parameters that are not in v. We therefore define a $\langle w, \phi \rangle$*-suffix* as a tuple $\langle v, \psi \rangle$, where ψ is a constraint over parameters of w and v, in which each literal contains at least one parameter from v, and such that $\langle uv, \phi \wedge \psi \rangle$ (which we often denote $\langle u, \phi' \rangle ; \langle v, \psi \rangle$) is a constrained word.

For a constrained word $\langle w, \phi \rangle$ we define its *potential*, denoted $pot(\langle w, \phi \rangle)$, as the set of formal parameters in w that do not occur as the left argument of any equality in ϕ. For example, $pot(\langle \alpha_1(p_1, p_2)\alpha_2(p_3, p_4), p_1 = p_2 \wedge p_2 = p_3 \rangle) = \{p_3, p_4\}$.

Definition 12 (Memorable)**.** *Let λ be a data language, and let $\mathcal{T}$ be the minimal DCDT recognizing λ. The λ-*memorable parameters *of a constrained word $\langle w, \phi \rangle \in Dom(\mathcal{T})$, denoted mem$_\lambda(\langle w, \phi \rangle)$, is the set of parameters in $pot(\langle w, \phi \rangle)$ that occur in some $\langle w, \phi \rangle$-suffix $\langle v, \psi \rangle$ such that $\langle w, \phi \rangle ; \langle v, \psi \rangle \in Dom(\mathcal{T})$.*

We are now ready to define our Nerode congruence on constrained words.

Definition 13 (Nerode Congruence)**.** *Let λ be a data language, and let $\mathcal{T}$ be the minimal DCDT recognizing λ. We define the equivalence $\equiv_\lambda$ on constrained words by $\langle w, \phi \rangle \equiv_\lambda \langle w', \phi' \rangle$ if there is a bijection γ : mem$_\lambda(\langle w, \phi \rangle) \mapsto$ mem$_\lambda(\langle w', \phi' \rangle)$ such that*

- *$\langle v, \psi \rangle$ is a $\langle w, \phi \rangle$-suffix with $\langle w, \phi \rangle ; \langle v, \psi \rangle \in Dom(\mathcal{T})$ iff $\langle v, \gamma(\psi) \rangle$ is a $\langle w', \phi' \rangle$-suffix with $\langle w', \phi' \rangle ; \langle v, \gamma(\psi) \rangle \in Dom(\mathcal{T})$, and then*
- *$\lambda(\langle w, \phi \rangle ; \langle v, \psi \rangle) = \lambda(\langle w', \phi' \rangle ; \langle v, \gamma(\psi) \rangle)$,*

where $\gamma(\psi)$ is obtained from ψ by replacing all parameters in mem$_\lambda(\langle w, \phi \rangle)$ by their image under γ.

Intuitively, two constrained words are equivalent if they induce the same residual languages modulo a remapping of their memorable parameters. The equivalence $\equiv_\lambda$ is also a congruence in the following

sense. If $\langle w, \phi \rangle \equiv_\lambda \langle w', \phi' \rangle$ is established by the bijection γ : $\text{mem}_\lambda(\langle w, \phi \rangle) \mapsto \text{mem}_\lambda(\langle w', \phi' \rangle)$, then for any $\text{mem}_\lambda(\langle w, \phi \rangle)$-suffix $\langle v, \psi \rangle$ we have $\langle w, \phi \rangle; \langle v, \psi \rangle \equiv_\lambda \langle w', \phi' \rangle; \langle v, \gamma(\psi) \rangle$.

We are now able to relate our Nerode congruence to DRAs.

Theorem 4 (Myhill-Nerode)**.** A data language λ is recognizable by a DRA iff the equivalence $\equiv_\lambda$ on constrained words in $Dom(\mathcal{T})$ has finite index, where $\mathcal{T}$ is the canonical CDT for λ.

Proof. If: The if-direction follows by constructing a DRA from a given $\equiv_\lambda$, as the DRA $\mathcal{A} = (L, l_0, X, \Gamma, \lambda)$, where, letting $\mathcal{T}$ be the canonical CDT for λ,

- L is given by the finitely many equivalence classes of the equivalence relation $\equiv_\lambda$ on $Dom(\mathcal{T})$ For each equivalence class, we choose a representative constrained word in $Dom(\mathcal{T}).\lambda$-essential constrained word.
- l_0 is $[\langle \epsilon, true \rangle]_{\equiv_\lambda}$, with the empty word as representative element.
- X maps each location $[\langle w, \phi \rangle]_{\equiv_\lambda}$ with representative constrained word $\langle w, \phi \rangle$ to the set $X([\langle w, \phi \rangle]_{\equiv_\lambda})$ of λ-memorable parameters of $\langle w, \phi \rangle$. Note that we here use parameters as variables.
- Γ is constructed as follows. For each location $l = [\langle w, \phi \rangle]_{\equiv_\lambda}$ in L with representative element $\langle w, \phi \rangle$ and each one-symbol extension of $\langle w, \phi \rangle$ of form $\langle w, \phi \rangle; \langle \alpha(\bar{p}), \psi \rangle$ in $Dom(\mathcal{T})$, there is a transition in Γ of form $\langle l, \alpha(\bar{p}), \psi, \gamma, l' \rangle$, where
  - $l' = [\langle w, \phi \rangle; \langle \alpha(\bar{p}), \psi \rangle]_{\equiv_\lambda}$; let $\langle w', \phi' \rangle$ be the representative element of the equivalence class $[\langle w, \phi \rangle; \langle \alpha(\bar{p}), \psi \rangle]_{\equiv_\lambda}$,
  - γ : $\text{mem}_\lambda(\langle w', \phi' \rangle) \mapsto \text{mem}_\lambda(\langle w, \phi \rangle; \langle \alpha(\bar{p}), \psi \rangle)$ is the bijection which is used to establish $\langle w', \phi' \rangle \equiv_\lambda \langle w, \phi \rangle; \langle \alpha(\bar{p}), \psi \rangle$ in Definition 13,

  Note that ψ is a constraint over memorable parameters of w (which are variables in l) and $\bar{p}$.
- $\lambda([\langle w, \phi \rangle]_{\equiv_\lambda}) = \lambda(\mathtt{w}_d)$ whenever $w_d \models \langle w, \phi \rangle$.

The constructed DRA is well defined: it has finitely many locations since the index of $\equiv_\lambda$ is finite, the initial location is defined as the class of the empty word, and λ is defined from λ for the representative elements of the locations. The transition relation is total and determinate. This is guaranteed by construction of guards from DCDTs, and by construction of DCDTs.

To complete this direction of the proof, we need to show that the constructed automaton $\mathcal{A}$ indeed recognizes λ. Consider an arbitrary sequence of transitions of $\mathcal{A}$, of form

$$\langle l_0, \alpha_1(\bar{p}_1), g_1, \rho_1, l_1 \rangle \quad \cdots \quad \langle l_{k-1}, \alpha_k(\bar{p}_k), g_k, \rho_k, l_k \rangle \ ,$$

which generates a run of form $\langle \alpha_1(\bar{p}_1) \cdots \alpha_k(\bar{p}_k) \ , g \rangle$, where g is $g_1 \wedge \tilde{\rho}_1(\cdots \wedge \tilde{\rho}_{k-1}(g_k)))$. Let $w = \alpha_1(\bar{p}_1) \cdots \alpha_k(\bar{p}_k)$. Then (by the construction of $\mathcal{A}$), $\langle w, g \rangle$ is a constrained word in $Dom(\mathcal{T})$, and (also by the construction of $\mathcal{A}$), $\mathcal{A}(\langle w, g \rangle) = \lambda_{\mathcal{T}}(\langle w, g \rangle)$. For any data word w_d with $\mathtt{w}_d \models \langle w, g \rangle$, this implies that $\mathcal{A}$ accepts w_d if $w_d \in L$, otherwise $\mathcal{A}$ rejects w_d. In summary, this implies that $\mathcal{A}$ correctly classifies data words that satisfy any of its runs.

Only if: For the only-if direction, we assume any (right-invariant) DRA that accepts λ. The proof idea then is to show that two “principal” constrained words in $Dom(\mathcal{T})$ that cause sequences of transitions that lead to the same location are also equivalent w.r.t. $\equiv_\lambda$, i.e., that one location of a DRA cannot represent more than one class of $\equiv_\lambda$. This can be shown straight-forwardly using right-invariance. □

We get as a corollary result from the only-if direction of the proof that the automaton generated in the first part of this proof is in fact a minimal (in the set of locations) right-invariant DRA recognizing λ. As stated already, minimality of the DCDT representation guarantees that the automaton will also use a minimal number of variables.

2.2.4 Example: Seat-booking

Seat-booking Consider an example of a (web) service for booking seats. The aim of this service is to allow a user to book a seat at some venue (for example, at the opera house). We can describe the seat booking service in terms of a data language L_s, where words in the language represent successful seat bookings. Figure 2.6 depicts the seat-booking service. User and system communicate by exchanging messages. The user can request available venues from the system and receive a reply in the form of a set of venues. The user can then select a venue from this set, and request available seats at this venue. Finally, the user can select an available seat and book it.

When describing the seat-booking example formally, will use the register automaton terminology with the addition of a basic scheme of sorts for formal parameters. We let $\mathcal{R} = \{=, \in\}$ be the set of binary relations (equality, set membership, and their negations). We use two sorts: Integers and Sets. Parameters of the same sort can be related by $=$ (equality); parameters of different sorts can be related using $\in$ (set membership; integers can be parts of sets).

The partial order $\sqsubseteq$ is such that $p_i \neq p_j \sqsubseteq p_i = p_j$ and $p_i \notin p_j \sqsubseteq p_i \in p_j$.

The set of actions in the seat booking example is

$$\{\mathsf{getVenues}(t_1, t_2),\ \mathsf{reply}(s_3),\ \mathsf{getSeats}(t_1, t_2, t_3),\ \mathsf{bookSeat}(t_1, t_2, t_3)\}$$

An accepting sequence of actions is

$$\mathsf{getVenues}(t_1, t_2),\ \mathsf{reply}(s_3),\ \mathsf{getSeats}(t_4, t_5, t_6),\ \mathsf{reply}(s_7),\ \mathsf{bookSeat}(t_8, t_9, t_{10})$$

where $t_1 = t_4 = t_7 \wedge t_2 = t_5 = t_8 \wedge t_6 \in s_3 \wedge t_{10} \in s_7$.

Since actions may carry more than one formal parameter in this example, we will clarify that the notion of length for a constrained word denotes the number of formal parameters.

Figure 2.6 depicts the seat booking language modeled as a register automaton. For brevity, we have written $\mathsf{getVenues}(t_1, t_2)/s_3$ short for $\mathsf{getVenues}(t_1, t_2),\ \mathsf{reply}(s_3)$, and similarly, $\mathsf{getSeats}(t_1, t_2, t_3)/s_4$ for $\mathsf{getSeats}(t_1, t_2, t_3),\ \mathsf{reply}(s_4)$.

We will now describe how to construct the canonical CDT for the seat-booking language. Recall that whenever $\mathfrak{w}$ is a constrained word, $\varphi^w|_{\leq k}$ denotes the conjunction of all literals of ϕ with level at most k. In the seat-booking example, we will let $k = 10$ in order to capture the accepting state, which can be reached after having processed 10 parameters.

Initially, we let $k = 1$; the only atom is the empty one. We apply the merging structure to the $k-1$-atom and obtain a set Ψ_0 of atomic branches for $k+1$, i.e., $\{t_1 = t_2, t_1 \neq t_2\}$. The least branch (by $\sqsubseteq$) is $t_1 \neq t_2$. We continue this procedure until we have a tree-structure with all the least branches up to length 10, the last one being the branch where all parameters are inequal and not in any set.

We now attempt to merge branches in the tree, starting at the 'leaves'. $\Psi_1 = \Psi$ since λ does not differentiate between any two level 10 constraints. We thus proceed back in the tree towards the root. When we reach level 4, Ψ_1 contains all branches where λ maps to $-$. We construct the maximal guard $t_1 \neq t_4$ whose support is a subset of Ψ_1. We add all constrained words with the constraint $t_1 \neq t_4$ to the CDT. We remove the support of this guard from Ψ, but there are still some branches left in Ψ, namely ones where $t_1 = t_4$.

Now we know that the guard $t_1 \neq t_4$ does not cover all the possible cases of constrained words. Thus, we go back and look at the remaining branches in Ψ. The least one is now $\psi = t_1 \neq t_2 \wedge t_2 \neq t_3 \wedge t_3 \neq t_1 \wedge t_1 = t_4$. We continue to construct the tree-structure with all the least branches up to length 10 again, the last one being the branch where $t_1 = t_4$ but all other parameters are inequal and not in any set.

Again we attempt to merge branches in the same manner as before, moving back in the tree towards the root. We will discover that the tree needs a subtree for $t_2 = t_5$ in order to classify all words correctly. We will then go back and construct the subtree with all the least branches up to length 10 again, and so on.

We are now ready to 'fold' the decision tree into an automaton for the seat booking example, which will look like the one in Figure 2.6.

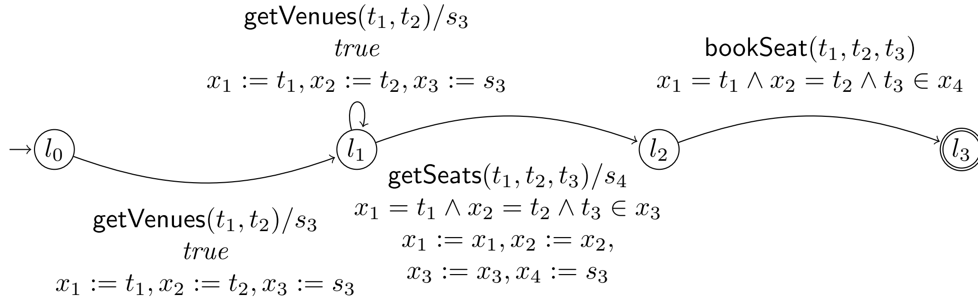

Figure 2.6: RA for the seat booking example

The *Blue* client A similar example, using a part of the *Blue* client system presented in D3.3 can be seen in Figure 2.7. We have used the same sorts as in the seat-booking example, and shortened actions with output (the reply action) accordingly.

Some small modifications have been made compared to the original D3.3 model: the startOrder action returns a set of possible items to buy, and the final state represents simply being able to place an order (order confirmation details have been omitted). Accepting sequences of actions are those where the item selected for purchase is in the set of items to buy, and the order ID matches the one initially supplied. An example of an accepted word is

$$\mathsf{startOrder}(t_1, t_2)/s_3,\ \mathsf{addToOrder}(t_4, t_5),\ \mathsf{placeOrder}(t_6, t_7)$$

where $t_2 = t_4 = t_6 \wedge t_5 \in s_3$.

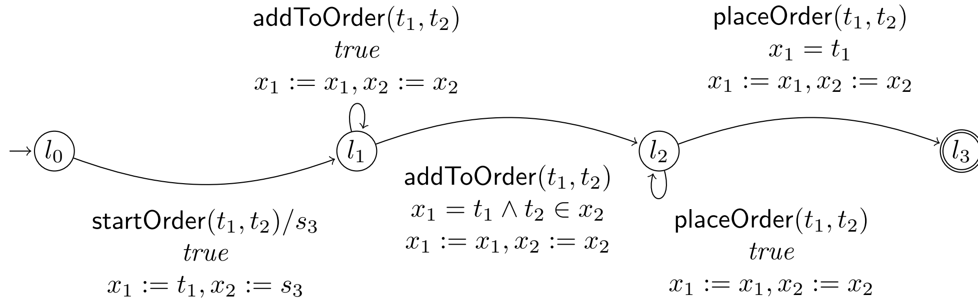

Figure 2.7: RA for the Blue web store example

2.2.5 Conclusions and future work

In this chapter, we have presented 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 have also presented an extended form of these register automata that is able to handle arbitrary relations between data parameters.

Our immediate plans within the scope of CONNECT is to extend the model to be able to cover output parameters in a more sophisticated way. This will allow us to faithfully model communication between networked systems. We also want to generalize our Angluin-style learning algorithm (see Chapter 3) to be able to handle arbitrary relations between data values.

3 Learning Register Automata with Data

A key task of WP4 in CONNECT is the development of methods for inferring models of networked systems. These inferred models will serve as a basis for synthesized CONNECTors.

In Chapter 2, we introduced a new formalism for modeling systems. This new formalism, a variant of register automata, is capable of expressing the influence of data (parameters) on the behavior of a networked system. More precisely, register automata can handle parameters representing, e.g., user names, passwords, or identifiers of connections.

In order to infer register automata models from black-box networked systems, active automata learning has to be extended correspondingly. Angluin's L^* [12] is a well-known active learning algorithm for DFA and Mealy machines that can be extended to the new automata model. In this chapter, we present this non-trivial extension.

As a motivating example, we have applied our learning algorithm using a small fragment of the XMPP protocol (cf. Figure 3.1), which serves as a representative for a large class of communication protocols and has also previously been an object of CONNECT case studies (see D4.2). Finally, we discuss the implementation and integration of this new method in LearnLib, which serves as a basis for the Learning Enabler (see Chapter 5).

3.1 The Algorithm

The active learning algorithm presented in the following subsections is unique in that it directly infers the effect of data values on control flow as part of the learning process. Conceptually, our new learning algorithm is based on a generalized Myhill-Nerode theorem for register automata, which, like in the classical regular case, identifies the required control locations [23]. Algorithmically, the L^*-typical partition refinement process [12] needs to be elaborated to a three-dimensional maximum fixpoint computation for simultaneously determining locations, register assignments, and guards of transitions. Technically, working on sequences of interactions with data requires additional care. It involves a "data-aware" way of composing prefixes and suffixes, as well as an adequate way of analyzing counterexamples with data values. We will show the impact of our approach by applying it to a small fragment of the XMPP protocol. The prototype implementation of our new technology drastically outperforms alternative approaches, even when they exploit optimizations like data abstraction and symmetry reduction.

3.1.1 Related work

For many applications in testing and verification (e.g., [24, 22]), and also in commercial model-based testing tools (e.g., ConformiQ Qtronic [39]), it is important to generate models that capture combined behavior of control and data.

Sadly, often such models do not exist for real systems. Automata learning techniques [35] have been used suggested for automated construction of models of such "black-box" systems. This has been illustrated in a number of case studies like, e.g., the concrete setting of Computer Telephony Integrated (CTI) systems [35], and in protocol specification [53], analysis [68], and testing [70].

Black-box techniques for learning component models broadly fall into two classes. One class generates finite-state models of control skeletons, modeling the sequences of interactions of a component [35, 41, 11, 68], or automata learning techniques (e.g., [12, 63]). Another class generates invariants over state variables [29] or exchanged data values by generalizing from concrete observations.

One approach for producing combined models capturing behavior of control and data [49, 52, 48] first generates control skeletons with data-agnostic control actions, which are then extended with data constraints in a post-process using a tool like Daikon [29]. This allows one to infer constraints on data parameters that are exchanged after specific sequences of method invocations, but not to analyze the influence of data parameter on subsequent control behavior. The method presented in [6] achieves a deeper integration of control and data at the price of user-supplied abstraction scheme (mapper), whereas [16]

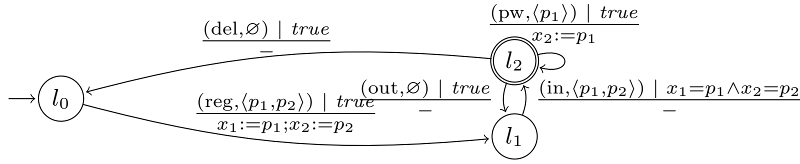

Figure 3.1: Partial model for a fragment of XMPP

requires a predefined fixed finite data domain. [64] constructs memory automata [45] from sequences of learned deterministic finite automata for increasing finite data domains. This approach could probably be generalized to infer register automata. However, such a generalization would be some exponentials more complex than our algorithm and yield automata of undetermined quality.

Technically, our involved three-dimensional treatment of counterexamples can be regarded as an elaboration of an algorithmic pattern which was originally presented in [63] for learning regular languages. We elaborated this pattern earlier to cover Mealy machine learning [69], and to support automated alphabet abstraction refinement [38].

3.1.2 Register automata and data languages

We will now recapitulate the register automaton formalism introduced in Section 2.1, but with some modifications that better suit our learning algorithm.

We assume an unbounded domain D of data values and a set A of *actions*. Each action has a certain *arity* which determines how many parameters it takes from the domain D. A *data action* is a term of form $(\alpha, \bar{d})$, where α is an action with arity n, and $\bar{d} = \langle d_1, \ldots, d_n \rangle$ are data values in D. A *data word* is a sequence of data actions. A *data language* is a set of data words, which is closed under permutations on D. We have presented an automaton model that recognizes data languages in [23].

Let a *parameterized action* be a term of form $(\alpha, \bar{p})$, consisting of an action α and formal parameters $\bar{p} = \langle p_1, \ldots, p_n \rangle$ respecting the arity of α. Let $X = \langle x_1, \ldots, x_m \rangle$ be a finite set of *registers*. A *guard* is a conjunction of equalities and negated equalities, e.g., $p_i \neq x_j$, over formal parameters and registers. An *assignment* is a partial mapping $\rho : \; X \to X \cup P$ for a set P of formal parameters.

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

- A is a finite set of *actions*.
- L is a finite set of *locations*.
- $l_0 \in L$ is the *initial location*.
- X is a finite set of *registers*.
- Γ is a finite set of *transitions*, each of which is of form $\langle l, (\alpha, \bar{p}), g, \rho, l' \rangle$, where l is the *source location*, l' is the *target location*, $(\alpha, \bar{p})$ is a parameterized action, g is a guard, and ρ is an assignment.
- $\lambda : L \mapsto \{+, -\}$ maps each location to either $+$ (accept) or $-$ (reject). □

Let us define the semantics of an RA $\mathcal{A} = (A, L, l_0, X, \Gamma, \lambda)$. A *valuation*, denoted by ν, is a (partial) mapping from X to D. A *state* of $\mathcal{A}$ is a pair $\langle l, \nu \rangle$ where $l \in L$ and ν is a valuation. The *initial state* is the pair of initial location and empty valuation $\langle l_0, \nu_0 \rangle$.

A *step* of $\mathcal{A}$, denoted by $\langle l, \nu \rangle \xrightarrow{(\alpha, \bar{d})} \langle l', \nu' \rangle$, transfers $\mathcal{A}$ from $\langle l, \nu \rangle$ to $\langle l', \nu' \rangle$ on input $(\alpha, \bar{d})$ if there is a transition $\langle l, (\alpha, \bar{p}), g, \rho, l' \rangle \in \Gamma$ such that (1) g is modeled by $\bar{d}$ and ν, i.e., if it becomes true when replacing all p_i by d_i and all x_i by $\nu(x_i)$, and such that (2) ν' is the updated valuation, where $\nu'(x_i) = \nu(x_j)$ wherever $\rho(x_i) = x_j$, and $\nu'(x_i) = d_j$ wherever $\rho(x_i) = p_j$.

A *run* of $\mathcal{A}$ over a data word $(\alpha_1, \bar{d}_1) \ldots (\alpha_k, \bar{d}_k)$ is a sequence of steps

$$\langle l_0, \nu_0 \rangle \xrightarrow{(\alpha_1, \bar{d}_1)} \langle l_1, \nu_1 \rangle \quad \ldots \quad \langle l_{k-1}, \nu_{k-1} \rangle \xrightarrow{(\alpha_k, \bar{d}_k)} \langle l_k, \nu_k \rangle.$$

A run is *accepting* if $\lambda(l_k) = +$, otherwise it is *rejecting*. The data language recognized by $\mathcal{A}$, denoted $L(\mathcal{A})$ is the set of data words that it accepts.

For the remainder of this chapter, we will work with RAs that are *completely specified*, meaning for any reachable state $\langle l, \nu \rangle$ and input $(\alpha, \bar{d})$, there is a transition with a guard modeled by $\bar{d}$ and ν, and *determinate*, i.e., no data word has both accepting and rejecting runs. We refer to such automata as DRAs. Data languages that are accepted by a DRA are called *regular*. We will restrict our attention to regular data languages.

Example 1. *We model the behavior of a fragment of the XMPP protocol [5] as an example (shown in Figure 3.1). XMPP is widely used in instant messaging. In our fragment of XMPP, a user can register an account (providing a username and a password), log in using this account, change the password, and delete the account. For example, the user Bob could register his account with the action* $\mathbf{reg}(\texttt{Bob}, \texttt{secret})$ *(providing his username and password), and then log in with the action* $\mathbf{in}(\texttt{Bob}, \texttt{secret})$*. Once logged in, he could change his password to* `boblovesalice` *with the action* $\mathbf{pw}(\texttt{boblovesalice})$*. In the figure, accepting locations are denoted by two concentric circles. Note that several transitions are omitted for brevity. We will use the XMPP example in Section 3.1.4 to demonstrate our learning algorithm.* □

As shown in [23], data languages can be represented concisely using a symbolic representation of data words. Here, we provide a summary using different but isomorphic representations of the concepts in [23] that allow a more amenable presentation.

Let $\mathcal{W}_D$ be the set of all data words over some set A of actions. For some data word $w = (\alpha_1, \bar{d}_1) \ldots (\alpha_n, \bar{d}_n)$ from $\mathcal{W}_D$ let $Acts(w)$ be the ordered sequence of actions in w, and $Vals(w) = d_1 \ldots d_m$ the (ordered) sequence of data values in w. Let $ValSet(w)$ be the set of distinct data values in $Vals(w)$.

Let $w \sqsubseteq w'$ denote that w' can be obtained from w by a not necessarily injective mapping on D, i.e., for two data words w, w' with $Vals(w) = d_1 \ldots d_m$, and $Vals(w') = d'_1 \ldots d'_m$,

$$w \sqsubseteq w' \;\Leftrightarrow\; Acts(w) = Acts(w') \;\wedge\; \forall 1 \leq i < j \leq m \,.\, d_i = d_j \;\Rightarrow\; d'_i = d'_j.$$

For example, $\mathbf{reg}(\texttt{Bob}, \texttt{test}) \sqsubseteq \mathbf{reg}(\texttt{Alice}, \texttt{Alice})$. Note that $\sqsubseteq$ is a preorder on $\mathcal{W}_D$. The smallest elements wrt. $\sqsubseteq$ are data words where all data values are pairwisely different. The greatest ones are data words where all data values are equal. For data words w, w', let $w \simeq w'$ denote that $w \sqsubseteq w'$ and $w \sqsupseteq w'$. The equivalence relation $\simeq$ induces a partitioning of data words into equivalence classes.

Let $Vals(w)|_k$ the prefix of length k of $Vals(w)$. For data words w, w' with $Acts(w) = Acts(w')$ let $w < w'$ denote that for some $k > 0$, (1) $Vals(w)|_{k-1} = Vals(w')|_{k-1}$ and (2) the kth data value of $Vals(w)$ is different from any of the $k-1$ first data values, but (3) the kth data value of $Vals(w')$ is equal to some of the $k-1$ first data values. For example, $\mathbf{reg}(\texttt{Bob}, \texttt{test})\mathbf{in}(\texttt{Bob}, \texttt{oth})$ is smaller (wrt. $<$) than $\mathbf{reg}(\texttt{Alice}, \texttt{test})\mathbf{in}(\texttt{Alice}, \texttt{test})$.

We assume an infinite ordered set $D_V = \{\mathbf{1}, \mathbf{2}, \mathbf{3}, \ldots\}$, which is disjoint from D. Let a *suffix* be a data word whose data values are in $D \cup D_V$. To allow for comparing suffixes by equality, we require that data values from D_V appear in canonical order in a suffix v, i.e., such that for every prefix p of $Vals(v)$ the set $ValSet(p) \setminus D$ is of form $\{\mathbf{1}, \mathbf{2}, \ldots, \mathbf{k}\}$ for some $\mathbf{k}$. For a data word u, let an u-suffix be a suffix v where all data values from D in v are also in $ValSet(u)$. We concatenate u and an u-suffix v, denoted by $u; v$ to the word $u\pi(v)$, where $\pi : D_V \rightarrow (D \setminus ValSet(u))$ is an injective mapping, and $\pi(v)$ denotes the application of π to all data values from D_V in v. For example, $\mathbf{in}(\texttt{Bob}, \mathbf{1})$ is a $\mathbf{reg}(\texttt{Bob}, \texttt{secret})$-suffix. Concatenation will result in the unique (up to equivalence wrt. $\simeq$) word $\mathbf{reg}(\texttt{Bob}, \texttt{secret})\mathbf{in}(\texttt{Bob}, \texttt{new})$.

3.1.3 Active learning of canonical RAs

We present a novel active learning algorithm, which infers a canonical DRA for an unknown data language $\mathcal{L}$, of which it initially knows only the set of actions. Active learning proceeds by asking two kinds of queries.

- A *membership query* consists in asking if a data word w 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 data word from the symmetric difference of $\mathcal{L}$ and $\mathcal{L}(\mathcal{H})$.

Key to (classic) L^*-like learning [12] is the well known Nerode congruence, which allows to identify words that lead to the same location in a canonical acceptor for some language $\mathcal{L}$. The Nerode congruence is formulated in terms of residual languages, i.e., languages after some prefix. Words with identical residuals will lead to the same location in a canonical acceptor. Active learning algorithms exploit this by means of two sets of words: (1) a finite prefix-closed set of *prefixes*, which is successively extended until it covers every transition of the canonical acceptor for $\mathcal{L}$, and (2) a finite set of *suffixes*, i.e., selected words from residuals, that allows to approximate the Nerode congruence on the set of prefixes. The necessary information is usually stored in an observation table. The rows and columns of this table are labeled with prefixes and suffixes, respectively. The table cell for a row labeled by u, and a column labeled by v, contains the information whether $uv \in \mathcal{L}$, i.e., whether v is in the $\mathcal{L}$-residual of u.

Active learning iterates two phases: hypothesis construction and hypothesis validation. During hypothesis construction the two sets of prefixes and suffixes are successively extended, using a sequence of membership queries, until the table satisfies satisfies certain “closure conditions”, under which a hypothesis automaton can be constructed in a consistent way. Hypothesis validation is performed using equivalence queries, to check if the current hypothesis is correct. From the returned counterexamples, new suffixes can be generated, that will drive a new round of hypothesis construction [63, 69]. During learning, hypothesis automata will grow monotonically in size, until they have the size of the canonical acceptor for $\mathcal{L}$. Then, by definition an equivalence query will confirm that the hypothesis is correct.

Our learning algorithm for regular data languages will strictly follow this pattern, and construct the canonical DRA for some data language $\mathcal{L}$. Theoretical backbone will be the new succinct Nerode congruence for data languages that we have presented in [23]. We will use sets of so-called $\mathcal{L}$-essential data words (cf. Section 3.1.3) and abstract suffixes (cf. Section 3.1.3) as prefixes and suffixes, from which membership queries for data words can be immediately derived. Due to the potentially complex patterns of relationship between data values in data languages, however, residuals will be more complicated in our algorithm than in the classic regular case, reflected in the more complex cells of our observation table. In the remainder of this section we will show

1. how abstract suffixes can be used to approximate the Nerode congruence (Section 3.1.3 and Section 3.1.3),
2. how an observation table can be realized and how at certain points well-defined hypothesis automata can be constructed from the observation table (Section 3.1.3), and
3. how counterexamples can be exploited to guarantee strictly monotone progress as in the classic regular case [69] (Section 3.1.3).

Strictly monotone progress together with an invariant on the size of hypothesis automata will deliver a correctness argument resembling the one from the classic case (Section 3.1.3). The invariant, however, is more complicated than in the classic case: We will show that for all hypothesis automata, the *number of transitions*, the *number of locations*, and the *sum of the number of register assignments* at some location will never exceed the corresponding numbers of the canonical DRA for $\mathcal{L}$. In essence, the overall pattern of learning DRA is a three-dimensional maximum fix-point computation, determining (a) the locations, (b) the required register assignments, and (c) the guarded transitions in a partition-refinement fashion.

Residual data languages In this section we will define residual data languages and present our Nerode congruence for data languages from [23] in terms of these. The development of this section is relative to canonical DRAs for regular data languages, whose existence has been proved in [23]. This allows us to avoid reciting the technically involved constructions presented in [23] without sacrificing the precision required to establish the correctness of our learning algorithm. The learning algorithm itself, however, does not depend on any a priori knowledge about the canonical DRA for an inferred data language.

Let $\mathcal{A}$ be the canonical DRA of some data language $\mathcal{L}$. For a run of $\mathcal{A}$ on some data word w of length n, i.e., with $|Acts(w)| = n$, let the *trace* of this run be the sequence of transitions $\tau = t_1, \dots, t_n$ of the run in the order they are traversed, and $Traces_{\mathcal{A}}(w)$ the set of all traces of runs of $\mathcal{A}$ on w (due to determinacy there may be more than one). For a trace τ, let $[\tau]$ be the set of smallest data words triggering this trace. These smallest words are important for the construction of canonical DRAs. Let $Traces_{\mathcal{A}}$ be the set of traces of $\mathcal{A}$.

Definition 15 ($\mathcal{L}$-essential words)**.** Given a data language $\mathcal{L}$ and its canonical DRA $\mathcal{A}$, we define $E_{\mathcal{L}} = \bigcup_{\tau \in Traces_{\mathcal{A}}} [\tau]$ to be the set of $\mathcal{L}$-*essential* words. □

Intuitively, the set of $\mathcal{L}$-essential words is an infinite prefix-closed set of smallest data words that trigger runs in the canonical DRA for $\mathcal{L}$, i.e., which have just enough equal data values to satisfy the guards of all traversed transitions.

When learning an unknown data language $\mathcal{L}$, the canonical DRA for $\mathcal{L}$ is, of course, unknown and cannot be used for the construction of $E_{\mathcal{L}}$. Our algorithm will find a representation system of $\mathcal{L}$-essential words by means of membership queries (cf. Section 3.1.3). In the XMPP example in Figure 3.1, ε (the empty word), $\mathbf{reg}(\mathtt{Bob}, \mathtt{secret})$, and $\mathbf{reg}(\mathtt{Bob}, \mathtt{secret})\mathbf{in}(\mathtt{Alice}, \mathtt{other})$ are examples of $\mathcal{L}$-essential words. They are smallest words triggering corresponding traces. Also, $\mathbf{reg}(\mathtt{Bob}, \mathtt{oth})\mathbf{in}(\mathtt{Bob}, \mathtt{oth})$ is $\mathcal{L}$-essential, triggering the **reg**-transition from l_0 and the "correct login" from l_1 to l_2 The word $\mathbf{reg}(\mathtt{Bob}, \mathtt{Bob})\mathbf{in}(\mathtt{Bob}, \mathtt{Bob})$, on the other hand, is not $\mathcal{L}$-essential. It, too, triggers the **reg**-transition and the "correct login" but it is not the unique (up to $\simeq$) smallest word for its trace.

In [23] we showed how from $E_{\mathcal{L}}$ the canonical DRA for $\mathcal{L}$ can be constructed. To determine the locations of this canonical automaton, we compare $\mathcal{L}$-essential words by their residual languages. Let therefore $\lambda_{\mathcal{L}} : \mathcal{W}_D \to \{+, -\}$ such that $\lambda_{\mathcal{L}}(w) = +$ if $w \in \mathcal{L}$ and $\lambda_{\mathcal{L}}(v) = -$ otherwise. For an $\mathcal{L}$-essential data word u and a set S of u-suffixes, we want to characterize the set of words $\{u; v \mid v \in S\}$ wrt. $\mathcal{L}$ in a concise and canonical way. For a subset $\lfloor S \rfloor$ of S, let $rep_{\lfloor S \rfloor} : S \to 2^{\lfloor S \rfloor}$ be a mapping that maps every suffix in S to a set of suffixes in $\lfloor S \rfloor$. We fix the definition of $rep_{\lfloor S \rfloor}$ independent of u and S. Let

$$rep_{\lfloor S \rfloor}(v) = max_{<}\{v' \in \lfloor S \rfloor \mid v' \sqsubseteq v\}.$$

We say that $\lfloor S \rfloor$ characterizes S faithfully after u if $\lambda_{\mathcal{L}}(u; v) = \lambda_{\mathcal{L}}(u; v')$ for $v' \in rep_{\lfloor S \rfloor}(v)$ and $v \in S$.

Definition 16 (Closures)**.** For an $\mathcal{L}$-essential word u and a set S of u-suffixes, the u-closure $C_u^S : \lfloor S \rfloor \to \{+, -\}$ is a mapping with unique minimal domain $\lfloor S \rfloor \subseteq S$ faithfully characterizing S after u, and $C_u^S(v) = \lambda_{\mathcal{L}}(u; v)$. □

We denote the u-closure for the set of all u-suffixes by C_u. In [23], we have shown that the unique minimal domain of C_u is the set of suffixes that extend u to $\mathcal{L}$-essential words.

For the $\mathcal{L}$-essential word $\mathbf{reg}(\mathtt{Bob}, \mathtt{oth})$, e.g., the $\mathbf{reg}(\mathtt{Bob}, \mathtt{oth})$-suffixes $\mathbf{in}(\mathbf{1}, \mathbf{2})$ and $\mathbf{in}(\mathtt{Bob}, \mathtt{oth})$ are in the domain of C_u, extending $\mathbf{reg}(\mathtt{Bob}, \mathtt{oth})$ to a word equivalent to $\mathbf{reg}(\mathtt{Bob}, \mathtt{oth})\mathbf{in}(\mathtt{Alice}, \mathtt{secret})$ and to $\mathbf{reg}(\mathtt{Bob}, \mathtt{oth})\mathbf{in}(\mathtt{Bob}, \mathtt{oth})$. These two words suffice to characterize faithfully the behavior of $\mathbf{reg}(\mathtt{Bob}, \mathtt{oth})$ for all suffixes v with only **in** as action: $C_u(\mathbf{in}(\mathbf{1}, \mathbf{2}))$ maps to $-$, corresponding to an unsuccessful login from l_1 in the DRA in Figure 3.1. $C_u(\mathbf{in}(\mathtt{Bob}, \mathtt{oth}))$ maps to $+$, characterizing correct logins.

Since the suffixes in $Dom(C_u)$ extend u to $\mathcal{L}$-essential words, the data values from D occurring in these suffixes are exactly the ones that are needed to satisfy the guards in the canonical DRA for $\mathcal{L}$. We refer to these data values as the *memorable* data values of u, and denote them by $mem_{\mathcal{L}}(u)$. In the above example, $\mathtt{Bob}$ and $\mathtt{oth}$ are in $mem_{\mathcal{L}}(\mathbf{reg}(\mathtt{Bob}, \mathtt{oth}))$. Note, however, that in general $mem_{\mathcal{L}}(u)$ will only be a subset of $ValSet(u)$.

Let π be a permutation on D. We apply π to closures, denoted by πC_u^S, by applying π to all data values from D in suffixes of $Dom(C_u^S)$ simultaneously, thereby exchanging values from D in the suffixes.

Definition 17 (Nerode congruence for essential words)**.** Two $\mathcal{L}$-essential words u and u' are equivalent w.r.t. $\mathcal{L}$, denoted by $u \equiv_{\mathcal{L}} u'$ if there exists a permutation π on D such that $\pi C_u = C_{u'}$. □

Note that $\equiv_{\mathcal{L}}$ is an equivalence relation. The bijection π used in Definition 17 need only relate memorable data values, i.e., it is enough to define it as a bijection $\pi : mem_{\mathcal{L}}(u) \to mem_{\mathcal{L}}(u')$. We say that two

closures are incompatible, denoted by $C_u \not\simeq C_{u'}$ if there is no permutation on D under which the closures become equal.

In our example, $\mathbf{reg}(\texttt{Alice}, \texttt{secret})$ and $\mathbf{reg}(\texttt{Bob}, \texttt{oth})\mathbf{in}(\texttt{Bob}, \texttt{oth})\mathbf{out}()$ are equivalent wrt. $\equiv_{\mathcal{L}}$ since their closures become equal under a permutation π on D, mapping `Alice` to `Bob` and `secret` to `oth`. In the canonical DRA in Figure 3.1 both words lead to l_1. Intuitively, π exchanges the data values stored in registers after processing the one word by data values stored in registers after processing the other word.

Hypothesis construction Our learning algorithm will use an observation table as underlying data structure. In this section we will define this data structure and explain how hypothesis automata can be generated from observation tables.

So far, we have defined suffixes only relative to fixed prefixes. We assume an infinite set Z of *placeholders*, ranged over by $z_1, z_2, \ldots$, which is disjoint from D and D_V. An *abstract suffix* is a data word with parameters in $Z \cup D_V$. One abstract suffix yields a number of (concrete) u-suffixes for a particular prefix u. For a set of abstract suffixes V, let $V(u)$ be the set of u-suffixes that can be generated from V via injective partial mappings $\sigma : Z \rightarrow ValSet(u)$. The abstract suffix $\mathbf{in}(z_1, z_2)$ for example will yield the $\mathbf{reg}(\texttt{bob}, \texttt{oth})$-suffixes $\mathbf{in}(\mathbf{1}, \mathbf{2})$, $\mathbf{in}(\texttt{bob}, \mathbf{1})$, $\mathbf{in}(\texttt{oth}, \mathbf{1})$, $\mathbf{in}(\mathbf{1}, \texttt{bob})$, $\mathbf{in}(\mathbf{1}, \texttt{oth})$, $\mathbf{in}(\texttt{oth}, \texttt{bob})$, and $\mathbf{in}(\texttt{bob}, \texttt{oth})$. The abstract suffix $\mathbf{in}(z_1, \mathbf{1})$, on the other hand, will result in $\mathbf{in}(\mathbf{1}, \mathbf{2})$, $\mathbf{in}(\texttt{bob}, \mathbf{1})$, and $\mathbf{in}(\texttt{oth}, \mathbf{1})$, only.

During learning, we will use membership queries for all words $u; v$ with $v \in V(u)$ to find the optimal, i.e., minimal, domain of $C_u^{V(u)}$ (along the lines of finding $\mathcal{L}$-essential words [23]). For the u-closure $C_u^{V(u)}$ let $mem_V(u)$ denote the set of data values from $ValSet(u)$ that occur in suffixes in the domain of $C_u^{V(u)}$. Even though the u-closure of an $\mathcal{L}$-essential word u for a set of abstract suffixes V will in general not contain suffixes that extend u to $\mathcal{L}$-essential words, the following propositions hold.

1. For all sets V of abstract suffixes $mem_V(u) \subseteq mem_{\mathcal{L}}(u)$, i.e., we will never wrongly identify data values as memorable. Intuitively, a data value that is not memorable in u cannot influence behavior in any suffix.

2. If $u \equiv_{\mathcal{L}} u'$ then $C_u^{V(u)} \simeq C_{u'}^{V(u')}$ for all sets of abstract suffixes V. This can be shown by proving mutual inclusion of the domains.

3. If $u \not\equiv_{\mathcal{L}} u'$ then there exists a finite set V of abstract suffixes such that $C_u^{V(u)} \not\simeq C_{u'}^{V(u')}$. Since $\equiv_{\mathcal{L}}$ has finite index k, in the worst case V has to generate all suffixes up to length k (We will do better, actually).

We can thus use closures as basis for our observation table.

Definition 18 (Observation table)**.** An observation table is a tuple (U, V, T), of a prefix-closed set of $\mathcal{L}$-essential words U, a set of abstract suffixes V, and a function T, mapping each prefix $u \in U$ to the u-closure $C_u^{V(u)}$. □

The set U consists of a prefixed-closed subset $Sp(U)$ of *short prefixes*, and contains for every prefix $u \in Sp(U)$ at least the one-action extension ua where data values in a do not equal one another or data values in u. The u-closure $T(u)$ is constructed by asking membership queries for all suffixes in $V(u)$, following the approach from [23]. Our algorithm will initialize $Sp(U) = V = \{\varepsilon\}$, and maintain the invariants that $u \not\simeq u'$ for $u, u' \in U$ and $T(u) \not\simeq T(u')$ for $u, u' \in Sp(U)$.

In order to construct hypothesis automata from an observation table, we need two conditions to hold on the table.

Definition 19 (Closedness)**.** An observation table (U, V, T) is *closed* if for every prefix $u \in U \setminus Sp(U)$ there is a prefix $u' \in Sp(U)$ and a permutation π on D such that $\pi T(u) = T(u')$. □

Please note that in general there can be multiple effective permutations. This can be due to true symmetry of parameters, but also to the approximative nature of intermediate results in learning. Since the existence of effective permutations is transitive, there can never be two permutations proving the

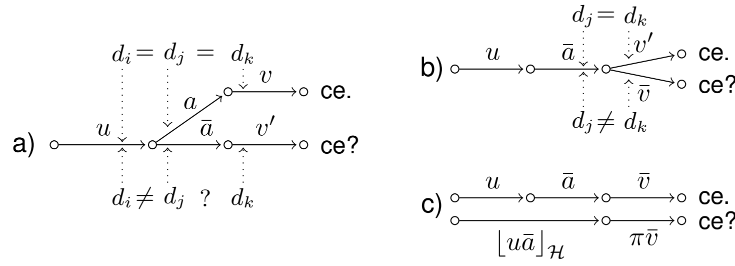

Figure 3.2: Counterexamples: a) new transition, b) new register, c) new permutation

same word from $U \setminus Sp(U)$ equivalent to different words from $Sp(U)$. The prefixes in $Sp(U)$ will become the locations of a hypothesis automaton. Closedness ensures that all transitions of the hypothesis, defined by prefixes in U, have a defined destination.

Definition 20 (Register-consistency)**.** An observation table (U, V, T) is *register-consistent* if for every prefix $ua \in U$, where a is of length one,

$$mem_V(ua) \cap ValSet(u) \subseteq mem_V(u). \qquad \square$$

When constructing a hypothesis from the table, we will store the parameters from $mem_V(u)$ in registers at the location corresponding to u. Register-consistency ensures that $mem_V(u)$ contains all parameters of u that are assumed to be stored in registers in continuations of u. This will guarantee that the assignments along transitions in the hypothesis are well-defined.

From a closed and register-consistent observation table we can construct a hypothesis automaton $\mathcal{H}$ along the lines of the approach presented in [23]. We will omit a detailed account of the automaton construction here, but simply give the key idea. The automaton is obtained from the observation table, using the set of prefixes and the permutations on D to determine locations and transitions. Registers are determined using the sets $mem_V(u)$ of closures $T(u)$. Guards and assignments can then be generated from the $\mathcal{L}$-essential words in U directly, and λ will be defined using values from the closures. We thus have:

Proposition 1. From a closed and register-consistent observation table (U, V, T) a well-defined hypothesis automaton $\mathcal{H}$ can be constructed, for which $\lambda_{\mathcal{H}}(u) = T(u)(\varepsilon)$ for $u \in U$. $\square$

Hypothesis validation Once we have generated a hypothesis automaton $\mathcal{H}$, an equivalence query will either signal success or return a counterexample, i.e., a data word w^c from the symmetric difference of $\mathcal{L}$ and $\mathcal{L}(\mathcal{H})$. We will process w^c from left to right in order to localize where precisely hypothesis and target system behave differently.

Starting with w^c, we will iteratively generate derived counterexamples, towards the word from $Sp(U)$ that leads to the same location in $\mathcal{H}$ as w^c. We refer to this word as the *access sequence* of w^c and denote it by $\lfloor w^c \rfloor_{\mathcal{H}}$. Key idea is that, since $w^c \in \mathcal{L} \Leftrightarrow \lfloor w^c \rfloor_{\mathcal{H}} \notin \mathcal{L}$, words generated in the process will at some point stop being counterexamples (cf. [63, 69]).

Technically, we will construct "triplet constrained words" uav, where $u \in Sp(U)$. We start with the triplet where u is the empty word ε, and av is w^c. We define the following three refinement steps, which will be iterated until we find a concrete discrepancy between $\mathcal{H}$ and the (unknown) canonical acceptor for $\mathcal{L}$. An example illustrating all steps will be given in Section 3.1.4.

A: Finding new transitions For ua of our triplet, let $u\bar{a}$ be a maximal (wrt. $<$) word from U.[1] Intuitively, $u\bar{a}$ corresponds to the trace of ua in $\mathcal{H}$. As shown (schematically) in Figure 3.2 a), we will try to transform the word $ua; v$ into a word $u\bar{a}; v'$ still being a counterexample. The problem here is deriving a suitable v' from v. If we cannot find such a word, we will find an $\mathcal{L}$-essential word $u\bar{a} \sqsubseteq ua' \sqsubseteq ua$ that we can

[1] Due to determinacy, there may be multiple such words of which we will pick one.

use as a new prefix in U. In this case we can continue with *hypothesis construction*. Otherwise, we continue with $u\bar{a}; v'$ and step B.

Technically, we will generate a sequence of counterexamples $ua; v = ua_1; v_1 > ua_2; v_2 > \ldots > ua_k; v_k$, by removing equalities between data values from ua_i that are not present in $u\bar{a}$. Removing equalities in ua_i may require refining the suffix v_i, too. For d_i, d_j, and d_k as shown in the Figure 3.2, we can try to make d_k equal to d_i, equal to d_j, or un-equal to both. For the at most $d = |Vals(a; v)|$ equal data values in the suffix there are $O(3^d)$ resulting candidate words $ua_{i+1}; v_{i+1}$ in each of the $k < |Vals(a)|$ steps. We continue until $ua_i \simeq u\bar{a}$ or no word $ua_{i+1}; v_{i+1}$ is a counterexample.

B: Finding new registers As shown in Figure 3.2 b), it may be that v' in $u\bar{a}; v'$ uses data values of $u\bar{a}$ not in $mem_V(u\bar{a})$, and thus are not stored in registers in $\mathcal{H}$ after processing $u\bar{a}$. Either the word $u\bar{a}; \bar{v}$ that is supported by the assignments in the hypothesis still is a counterexample and we continue with step C, or we will find a suffix v'' indicating a new register and continue with *hypothesis construction*.

The smallest sensible v'' results from a sequence of suffixes $v' = v'_1 > v'_2 > \ldots > v'_k = v''$ still yielding counterexamples, where $(ValSet(v'_{i+1}) \cap ValSet(a)) \subset (ValSet(v'_i) \cap ValSet(a))$. In each of the $k < |Vals(a)|$ steps we have to consider at most $|Vals(a)|$ candidate suffixes. A register will then be introduced by adding the abstract suffix $\langle v'' \rangle$ to V, which we generate from v'' by replacing all data values from D by placeholders.

C: Finding new locations Finally, let $\lfloor u\bar{a} \rfloor_{\mathcal{H}}$ be the access sequence of $u\bar{a}$, i.e., the word from $Sp(U)$ for which $\pi T(u\bar{a}) \simeq T(\lfloor u\bar{a} \rfloor_{\mathcal{H}})$ for some permutation π on D (used during hypothesis construction). In this step we will replace $u\bar{a}$ by its access sequence using π to replace data values in $\bar{v}$.

If $\lfloor u\bar{a} \rfloor_{\mathcal{H}}; \pi(\bar{v})$ is not a counterexample, as shown in Figure 3.2 c), either π is the wrong permutation from a set of potential ones, or both words are not equivalent wrt. $\equiv_{\mathcal{L}}$. In both cases, adding the abstract suffix $\langle \bar{v} \rangle$ to the table will make this explicit, and lead to a new permutation or, in case no effective permutation is left, to unclosedness, i.e., a new location. If $\lfloor u\bar{a} \rfloor_{\mathcal{H}}; \pi(\bar{v})$ still is a counterexample, we will start over with step A, using $\lfloor u\bar{a} \rfloor_{\mathcal{H}}$ as u and (misusing notation) $\pi(\bar{v})$ as $a; v$.

Since w^c is a counterexample, at some point one of the three steps will deliver a new prefix or suffix. Denoting the maximal length (i.e., $|Acts(w^c)|$) of a counterexample by m and the arity of the action with most parameters by p, we can estimate the number of membership queries we need to process a counterexample by $O(pm3^{pm})$. We thus have:

Proposition 2. Every counterexample delivers either *a new transition*, or an abstract suffix leading to *an increased number of locations* or *an increased sum of the number of register assignments*, or it leads to a reduced number of *symmetries between assigned registers* at a particular location. □

Correctness and Complexity Inferring an unknown data language over the set of actions A, the learning algorithm proceeds in rounds. In each round a well-defined hypothesis automaton can be constructed from the closed and consistent observations (Proposition 1). For initialization $Sp(U) = \{\varepsilon\}$, i.e., it contains the access sequence of the initial location, while $U \setminus Sp(U)$ contains a word with no equal data values for every $\alpha \in A$. The set of abstract suffixes is initialized as $V = \{\varepsilon\}$, distinguishing accepting and rejecting locations.

As usual, we will estimate the number of necessary membership and equivalence queries in terms of the size of the canonical DRA for the considered regular data language. Let the number of registers be denoted by r, the number of locations by n, the number of transitions by t, the arity of the action with most parameters by p, and the length of the longest counterexample by m.

Then, by construction, the number of prefixes in the final observation table is $t + 1$, i.e., in $O(t)$, and the number of suffixes lies in $O(nr)$: less than n to distinguish locations, less than nr to realize *register-consistency*, and less than nr to reduce the number of possible permutations.[2]

Each processing of a counterexample, which may require $O(pm3^{pm})$ membership queries, will lead to a refined observation table from which a new hypothesis automaton can be constructed. This automaton

[2] Reducing the number of permutations follows the same partition-refinement-pattern as automata learning does in general: With every new suffix a group of symmetric data values / registers is split (at a particular location).

Table 3.1: Observation Table (only showing a subset of all prefixes)

		ε	$\mathbf{in}(z_1, z_2)$	$\mathbf{out}()\mathbf{in}(z_1, z_2)$
ε	(l_0)	$-$	$\mathbf{in}(\mathbf{1}, \mathbf{2})$ $-$	$\mathbf{out}()\mathbf{in}(\mathbf{1}, \mathbf{2})$ $-$
$\mathbf{reg}(a, b)$	(l_1)	$-$	$\mathbf{in}(\mathbf{1}, \mathbf{2})$ $-$ $\mathbf{in}(a, b)$ $+$	$\mathbf{out}()\mathbf{in}(\mathbf{1}, \mathbf{2})$ $-$ $\mathbf{out}()\mathbf{in}(a, b)$ $+$
$\mathbf{reg}(a, b)\mathbf{in}(a, b)$	(l_2)	$+$	$\mathbf{in}(\mathbf{1}, \mathbf{2})$ $+$	$\mathbf{out}()\mathbf{in}(\mathbf{1}, \mathbf{2})$ $-$ $\mathbf{out}()\mathbf{in}(a, b)$ $+$
$\mathbf{reg}(a, b)\mathbf{in}(c, d)$		$-$	$\mathbf{in}(\mathbf{1}, \mathbf{2})$ $-$ $\mathbf{in}(a, b)$ $+$	$\mathbf{out}()\mathbf{in}(\mathbf{1}, \mathbf{2})$ $-$ $\mathbf{out}()\mathbf{in}(a, b)$ $+$
$\mathbf{reg}(a, b)\mathbf{in}(a, b)\mathbf{pw}(c)$		$+$	$\mathbf{in}(\mathbf{1}, \mathbf{2})$ $+$	$\mathbf{out}()\mathbf{in}(\mathbf{1}, \mathbf{2})$ $-$ $\mathbf{out}()\mathbf{in}(a, c)$ $+$
$\mathbf{reg}(a, b)\mathbf{in}(a, b)\mathbf{out}()$		$-$	$\mathbf{in}(\mathbf{1}, \mathbf{2})$ $-$ $\mathbf{in}(a, b)$ $+$	$\mathbf{out}()\mathbf{in}(\mathbf{1}, \mathbf{2})$ $-$ $\mathbf{out}()\mathbf{in}(a, b)$ $+$

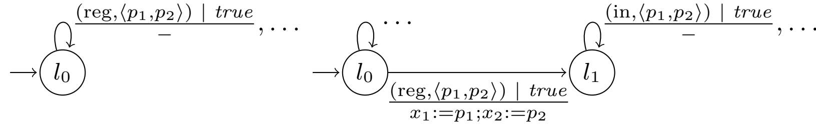

Figure 3.3: First and second hypothesis

will either have more transitions, more locations, or more registers than the previous one, or it uses a different permutation between prefixes reaching a location, where the number of possible permutations decreases strictly monotonically (Proposition 2). Due to the monotonicity of the refinement steps, chaotic fixpoint iteration is guaranteed to terminate after finitely many rounds with the greatest fixpoint, which resembles the canonical DRA for $\mathcal{L}$.

The number of membership queries needed to fill the observation table depends on the number of membership queries needed to produce all closures. An abstract suffix can have at most r abstract parameters, which can be instantiated by less than np parameters in the potential of a word in less than $(np)^r$ combinations. The number of membership queries needed to construct all closures lies therefore in $O(tnr \cdot (np)^r)$.

Theorem 5. Regular data languages can be learned with $O(t + nr)$ equivalence queries and $O(tnr \cdot (np)^r + (t + nr) \cdot pm3^{pm}))$ membership queries. □

Two factors for the number of membership queries look critical. (1) the "concatenation" of prefixes and abstract suffixes, which is responsible for the exponential term of the first summand, and (2) the transformation of arbitrary prefixes of counterexamples into corresponding $\mathcal{L}$-essential words, leading to the exponential term of the second summand. It should be noted, however, that both exponents are typically quite small in practice. In fact, p may well be considered a constant in many contexts, and pm estimates the worst case in which all data values of a counterexample are equal, which usually can be avoided when searching for counterexamples. Finally, the number of required registers r will typically grow much slower than the model size. This observation was also supported by our experiments.

3.1.4 Example application

In this section we give an example of a complete run of our algorithm, using the XMPP example from Figure 3.1, and present some performance data for our implementation of the algorithm.

The resulting (final) observation table for the example is shown (partly) in Table 3.1. The left column contains prefixes. Prefixes from $Sp(U)$ are shown in the upper part of the table. The three other columns are labeled with abstract suffixes. Table cells of a row labeled u contain suffixes from the domain of the u-closure $C_u^{V(u)}$ grouped per abstract suffix. The table was initialized as described in Section 3.1.3. The algorithm starts by constructing closures for all prefixes in U and the empty suffix. Since all prefixes are

Table 3.2: Experimental Results

Setup	# Loc.	# Trans.	MQs	EQs
RA learning algorithm	3	16	403	3
L^*, symmetry reduction, $\|D\| = 6$)	73	5,913	2,776	2
L^*, no optimization, $\|D\| = 6$)	73	5,913	415,333	72

not in $\mathcal{L}$, the table is immediately closed and consistent. In the constructed hypothesis, shown in the left of Figure 3.3, all prefixes lead to one non-accepting state.

An equivalence query returns the counterexample $\mathbf{reg}(a,a)\mathbf{in}(a,a)$ which is in $\mathcal{L}$ but rejected by the hypothesis. Performing step A of handling counterexamples results in a word $\mathbf{reg}(a,b)\mathbf{in}(a,b)$, which still is a counterexample. When refining the word to be supported by the (empty) assignment along the **reg**-transition in the hypothesis (step B), the words $\mathbf{reg}(a,b)\mathbf{in}(a,d)$ and $\mathbf{reg}(a,b)\mathbf{in}(c,b)$ are no longer counterexamples. In order to subsequently correct the yet empty assignment, we add the abstract suffix $\mathbf{in}(z_1,z_2)$ to the table.

When completing the table, the closure for the prefix $\mathbf{reg}(a,b)$ will be incompatible with the other closures, which can be seen in Table 3.1. In order to get a *closed* observation table, $\mathbf{reg}(a,b)$ will be added to $Sp(U)$, and $U \setminus Sp(U)$ will be extended accordingly. From the closed table we construct the hypothesis that is shown in the right of Figure 3.3.

We will get the same counterexample as in the first round. Analyzing it, we perform the refinement steps described in Section 3.1.3. We first perform the refinement steps for the empty prefix. First we transform $\mathbf{reg}(a,a)\mathbf{in}(a,a)$ to $\mathbf{reg}(a,b)\mathbf{in}(a,b)$ (step A). Steps B and C will not modify this counterexample since the equalities are supported already by the hypothesis and since $\mathbf{reg}(a,b)$ is its own access sequence. The second round starts with $\mathbf{reg}(a,b)$ as u, $\mathbf{in}(a,b)$ as a, and an empty suffix v. When refining $\mathbf{in}(a,b)$ to be supported by the corresponding guard of the **in**-transition from l_1 (step A), we discover that $\mathbf{reg}(a,b)\mathbf{in}(a,d)$ and $\mathbf{reg}(a,b)\mathbf{in}(c,b)$ are no counterexamples. Hence, $\mathbf{reg}(a,b)\mathbf{in}(a,b)$ must be $\mathcal{L}$-essential. We add it to $U \setminus Sp(U)$ in order to represent the guarded **in**-transition in the table.

To *close* the table, we have to move the new prefix to $Sp(U)$ as its closure is incompatible with the other closures. We extend $U \setminus Sp(U)$ accordingly. Now the resulting table is not *register-consistent*: It does not support any (re-)assignment along the new prefix as its closure does not have memorable data values. The closure of its continuation $\mathbf{reg}(a,b)\mathbf{in}(a,b)\mathbf{out}()$, however, has two memorable data values, namely a and b. We add $\mathbf{out}()\mathbf{in}(z_1,z_2)$ to the set of suffixes. From the closed and consistent observation table, shown in Table 3.1, we construct the final model: the canonical DRA from Figure 3.1.

We have implemented the outlined algorithm on top of LearnLib [53], and applied it to the discussed example. Counterexamples were found automatically by comparing DFAs, generated from hypothesis and target model for a small, concrete data domain. We compared our new learning algorithm for RAs with algorithms for learning DFAs utilizing abstraction, which to our knowledge would be the state-of-the-art approach to learning a system like the XMPP protocol. We have generated a DFA from the DRA in Figure 3.1 for the smallest sensible data domain of size 6 (the longest membership query has 6 distinct data values). This can be considered an optimal data abstraction. We have learned the model twice: once with no optimization, and once with a symmetry filter. The key figures of all experiments are shown in Table 3.2. The experiments show that learning register automata not only delivers much more expressive models, but (in this particular case) also is much more efficient than classic L^*-based learning.

3.1.5 Conclusions

In this section, we have presented an active learning algorithm for register automata, which allows capturing the flow of parameter values taken from arbitrary domains. The application of our algorithm to a small example indicates the impact of learning register automata models: The inferred models are much more expressive than finite state machines, allowing for capturing the semantics of networked systems in CONNECT on a more complex level. Moreover, the prototype implementation also drastically outperforms

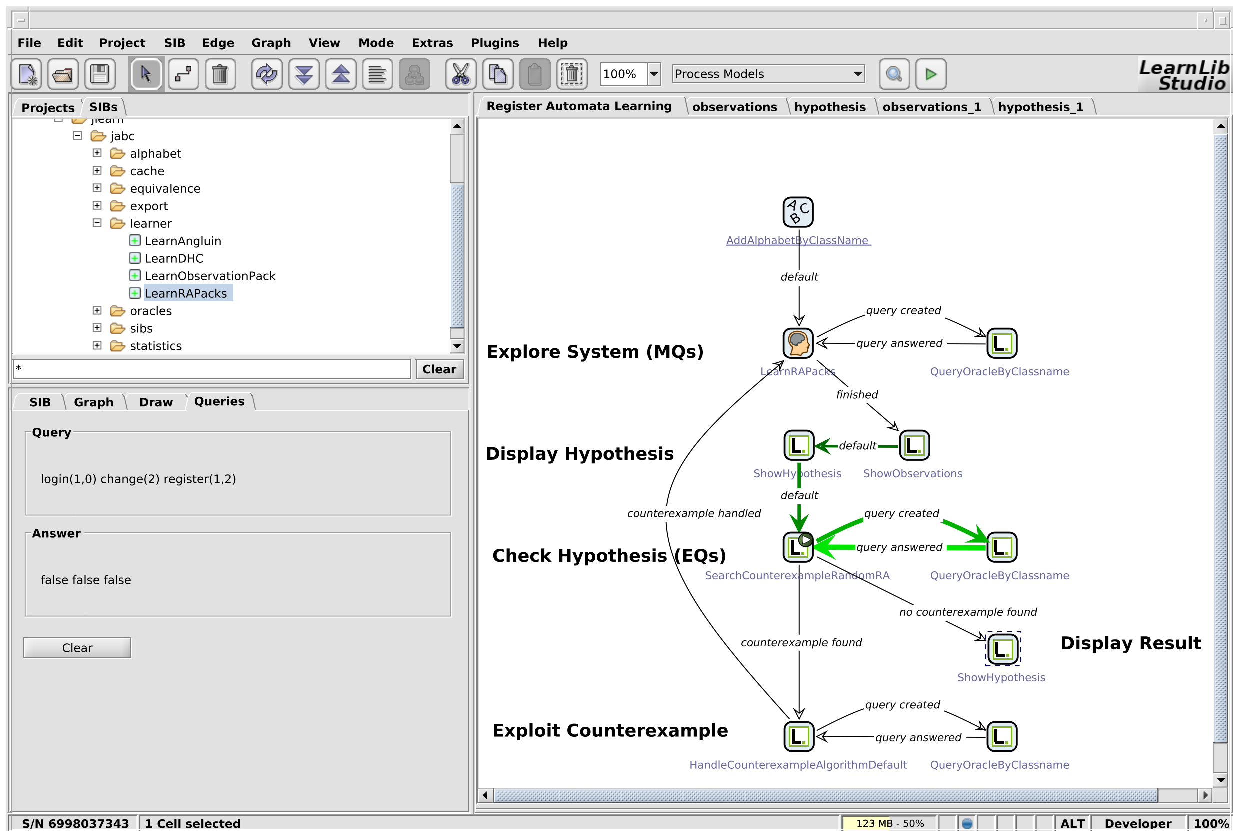

Figure 3.4: A modeled learning setup created in LearnLib Studio. The model is currently executed, with bold edges denoting the path of execution. The current query and its answer are made visible in the panel on the lower left side.

the classic L^* algorithm, even when exploiting optimal data abstraction and symmetry reduction. Due to the relatively high cost of learning, especially in networked environments, this imposes a significant improvement in terms of practical applicability. Currently, we are investigating the limits of our technology by considering generalizations, in particular concerning the transition structure, and by exploring scalability and potential optimizations.

3.2 Integration of Register Automata in LearnLib

Thanks to the flexible component-based approach of LearnLib, it was possible to integrate components for learning this new model type into the general framework without changes to the LearnLib architecture.

In the learning setup presented above, the only components that are specific for the RA machine model are the SIBs encapsulating the learning algorithm and the equivalence approximation. The overall infrastructure provided by the LearnLib is reused to a high degree when using RAs despite adapting a much richer automata model. The overall flavor of how learning setups can be created is completely unchanged compared to how setups are specified for other machine models, abstracting from the details of the underlying data structures and algorithms and thus shielding the user from additional complexity.

Due to being integrated into the component framework, learning setups for Register Automata can use all facilities LearnLib offers for debugging, visualization and statistics. Thus our extension provides a powerful and unique framework for learning data independent systems [47].

4 Type Analysis for Model Inference

As discussed in the previous chapters, one main aim of WP4 in CONNECT is the development of methods for the automated inference of models of networked systems. While the previous chapters concentrated on the modeling and inference aspects of the problem, the main focus of this chapter is the automation of the process.

When learning a black-box system, it is necessary to have some means to bridge between the concrete execution level (where actual messages are sent to the system) and the abstract model level. Usually, such a test harness is constructed manually, which can be time-consuming. Not only the manual implementation of an instrumentation of a system under learning can be error-prone and expensive. Also the manual design of interface to be inferred (i.e, the methods, the level of abstraction, the data values to be used, etc.) requires considerable effort. Luckily, in the context of CONNECT, these tasks can be automated quite well: We propose a method to automatically generate test-harnesses and interface alphabets from interface descriptions in WSDL that are capable of serving as a basis for inferring dataflow-sensitive behavioral models of black-box systems. WSDL is a subset of the xDL interface language, which is used in CONNECT and discussed in D1.2. Thus, in the following, any references to WSDL indeed refer to the corresponding xDL subset.

We use the StrawBerry tool to syntactically analyze a WSDL interface description, providing a type analysis that allows us to generate an (data aware) interface alphabet and a corresponding (data aware) test harness. The generated artifacts can be used by LearnLib directly to infer a model of the black-box system in question. In this way, we are also able to obtain global behavioral models that comprise the data-flow of the analyzed systems. Our method is unique in combining the general applicability of StrawBerry, which simply requires WSDL interfaces, with the ability of active automata learning to infer data-sensitive behavioral models.

An overview on related work is provided in Sect. 4.1. Preexisting tools that are facilitated in the presented approach are discussed in Sect. 4.2, followed by a running example in Sect. 4.3. The integration of syntactic interface analysis and automata learning is discussed in Sect. 4.4, for which results are provided and discussed in Sect. 4.5. Subsequently, Sect. 4.6 evaluates the applicability and feasibility of the presented approach by applying it to three web-services. Finally, we conclude in Sect. 4.7 and indicate some directions for future work.

4.1 Related Work

Inferring formal properties of software components has been a major research interest for the past decade. Most available approaches fall into one of two classes. One class generates extrinsic properties (e.g., invariants). The other class generates intrinsic properties, e.g., models describing the actual behavior of components. In both classes active and passive approaches, as well as black-box and white-box variants can be found. While StrawBerry falls into the first category, LearnLib is of the second kind.

The class of methods for generating properties can be further subdivided into methods that “mine” statistical models and methods that generate invariants. In the class of methods that generate statistical models, the approaches described in [72, 71] mine Java code to infer sequences of method calls. These sequences are then used to produce object usage patterns and operational preconditions, respectively, that serve to detect object usage violations in the code. StrawBerry shares with [71] the way an object usage pattern is represented, i.e., as a set of temporal dependencies between method calls (e.g., $m < n$ means “calls to m precede calls to n”).

The work of [60] presents a passive method for the automated generation of specifications of legal method call sequences on multiple related objects from from method traces of Java programs, [28] extends this method by active testing. As for StrawBerry, tests are used to refine the invariants that have been generated inductively from the provided information. However, in contrast to StrawBerry, none of these approaches focuses on data-flow invariants explicitly. A tool that infers invariants from data is Daikon [29].

In the class of methods that generate intrinsic properties, especially automata learning has been used

to generate behavioral models of systems. Active learning, as implemented in LearnLib, has been used to infer behavioral models of CTI systems as early as 2002 [35, 40]. It has since then been applied in a number of real-life case studies (e.g., [62, 61]). In these case studies, however, data has never been treated explicitly but was rather hidden from the learning algorithm. In [66], systems with data parameters are considered. However, this work does not consider relations between different parameters. Recently, automata learning has been extended to deal with data parameters and data dependencies explicitly by means of hand-crafted mappers [44, 6]. Our approach is unique in generating mappers automatically.

There are only few approaches that combine inference of behavioral models and invariants on data-flow. The authors of [16] present an approach for inferring state machines (by means of active learning) for systems with parameterized inputs. They first infer a behavioral model for a finite data domain, and afterwards abstract this model to a symbolic version, encoding extrapolated invariants on data parameters as guarded transitions.

The authors of [52, 49] demonstrate how behavioral models can be created with passive learning from observations gathered by means of monitoring. In addition, this approach tries to capture the behavioral influence of data values by applying an invariance detector [29]. This approach, however, is subject to the issue of all passive approaches: they are limited to the (possibly small) set of observed executions. If a piece of code or part of the application is not executed, it will not be considered in the generated model.

The work described in [31] (i.e., the SPY approach) aims for inferring a behavioral specification (in this case: graph transformation rules) of Java classes that behave as data containers components by first observing their run-time behavior on a small concrete data domain and then constructing the transformation rules inductively from the observations.

It is common to all these approaches that they work on a large basis of concrete information that by induction is condensed into symbolic behavioral models. Invariants on data values are obtained in a post-processing step after construction of behavioral models.

In [9] an approach is presented that generates behavioral interface specifications for Java classes by means of predicate abstraction and active learning. Here, predicate abstraction is used to generate an abstract version of the considered class. Afterwards a minimal interface for this abstract version in obtained by active learning. This is a white-box scenario, and learning is used only to circumvent more expensive ways of computing the minimal interface.

Our approach, in contrast, provides a solution for the black-box scenario. Similarly, however, we use StrawBerry to compute an interface alphabet, and mapper, which in combination work as an abstraction, and infer a model at the level of this abstraction, using LearnLib.

In summary, to our knowledge, the presented approach is unique in producing behavioral models from interface descriptions in the black-box scenario fully automatically and without further prerequisites (e.g., information about sensibles finite domain sizes).

4.2 The Constituent Tools

This section sketches the characteristics of the two employed tools, StrawBerry [17] and LearnLib [54]. We will later see that their combination allows for a fully automated inference of data-sensitive behavioral models.

4.2.1 StrawBerry

By taking as input a WSDL of a WS, StrawBerry (presented in D3.1) derives in an automated way a partial ordering relation among the invocations of the different WSDL operations. This partial ordering relation is represented as an automaton that we call *Behavior Protocol automaton*. It models the interaction protocol that a client has to follow in order to correctly interact with the WS. This automaton explicitly models also the information that has to be passed to the WS operations. The behavior protocol is obtained through synthesis and testing stages. The synthesis stage is driven by syntactic interface analysis (aka data type analysis), through which we obtain a preliminary dependencies automaton that can be optimized by

means of heuristics. Once synthesized, this dependencies automaton is validated through testing against the WS to verify conformance, and finally transformed into an automaton defining the behavior protocol.

StrawBerry is a black-box and extra-procedural method. It is black-box since it takes into account only the WSDL of the WS. It is extra-procedural since it focuses on synthesizing a model of the behavior that is assumed when interacting with the WS from outside, as opposed to intra-procedural methods that synthesize a model of the implementation logic of the single WS operations [49, 71, 72]. Figure 4.1 graphically represents StrawBerry as a process split in five main activities.

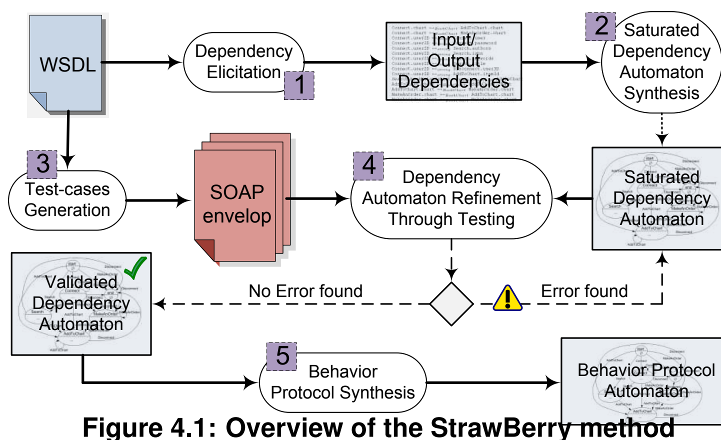

Figure 4.1: Overview of the StrawBerry method

The *Dependencies Elicitation* activity elicits data dependencies between the I/O parameters of the operations defined in the WSDL. A dependency is recorded whenever the type of the output of an operation matches with the type of the input of another operation. The match is syntactic. The elicited set of I/O dependencies may be optimized under some heuristics [17].

The elicited set of I/O dependencies (see the *Input/Output Dependencies* artifact shown in Figure 4.1) is used for constructing a data-flow model (see the *Saturated Dependencies Automaton Synthesis* activity and the *Saturated Dependencies Automaton* artifact shown in Figure 4.1) where each node stores data dependencies that concern the output parameters of a specific operation and directed arcs are used to model syntactic matches between output parameters of an operation and input parameters of another operation. This model is completed by applying a *saturation rule*. This rule adds new dependencies that model the possibility for a client to invoke a WS operation by directly providing its input parameters. The resulting automaton is then validated against the implementation of the WS through testing (see *Dependencies Automaton Refinement Through Testing* activity shown in Figure 4.1).

The testing phase takes as input the SOAP messages produced by the *Test-cases generation* activity. The latter, driven by coverage criteria, automatically derives a suite of test cases (i.e., SOAP envelop messages) for the operations to be tested, according to the WSDL of the WS. For this purpose, we use the WS-TAXI [14] tool that takes as input the WSDL of a WS and automatically produces the SOAP envelop messages ready for execution. In StrawBerry tests are generated from the WSDL and aim at validating whether the synthesized automaton is a correct abstraction of the service implementation. Intuitively, we can say that StrawBerry tests if the model conforms to the implementation. Testing is used to refine the syntactic dependencies by discovering those that are semantically wrong. By construction, the inferred set of dependencies is syntactically correct. However, it might not be correct semantically since it may contain false positives (e.g., a string parameter used as a generic attribute is matched with another string parameter that is a unique key). If during the testing phase an error is found, these false dependencies are eliminated from the automaton.

Once the testing phase is successfully terminated, the final automaton models, following a data-flow paradigm, the set of validated "chains" of data dependencies. StrawBerry terminates by transforming this data-flow model into a control-flow model (see the *Behavior Protocol Synthesis* activity in Figure 4.1). This is another kind of automaton whose nodes are WS execution states and whose transitions, labeled with operation names plus I/O data, model the possible operation invocations from the client to the WS.

The primary result of StrawBerry used in the subsequent learning phase is the set of validated "chains"

of data dependencies.

4.2.2 LearnLib and active automata learning

LearnLib is a framework for automata learning and experimentation (cf. D4.1). Active automata learning tries to automatically construct a finite automaton that matches the behavior of a given target automaton on the basis of active interrogation of target systems and observation of the produced behavior.

Active automata learning originally has been conceived for language acceptors in the form of deterministic finite automata (DFAs) (cf. Angluin's L^* algorithm [12]). It is possible, however, to apply automata learning to create models of reactive systems instead. A more suited formalism for this application are Mealy machines:

Definition 21. *A Mealy machine is defined as a tuple* $\langle Q, q_0, \Sigma, \Omega, \delta, \lambda \rangle$ *where*

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

In the context of reactive systems, the input alphabet contains actions which can be executed on the target system, while the output alphabet is determined by the output the system produces in response to the executed input actions.

When employed to create models in the form of Mealy machines, active automata learning employs two distinct types of queries to gather information on the System Under Learning (SUL):

- Membership Queries (MQs) retrieve behavioral information of the target system. Consisting of traces of system stimuli (each query $mq \in \Sigma^*$), MQs actively trigger behavioral outputs which are collected and analyzed by the learning algorithm. MQs are used to construct a hypothesis, which is subject of a verification by a second class of queries, the equivalence queries.
- Equivalence Queries (EQs) are used to determine if the learned hypothesis is a faithful representation of the target system. If the equivalence oracle handling the EQ finds diverging behavior between the learned hypothesis and the target system a counterexample $ex \in \Sigma^*$ will be produced, which is used to refine the hypothesis after restarting the learning process.

With those two query types, learning algorithms, such as $L^*_{i/o}$ [41], create minimal automata models, i.e., the learned result never contains more states than the minimized representation of the target system, and also guarantee termination with an accurate learned model.

LearnLib contains several learning algorithms fit for learning reactive systems, embedded in a flexible framework.

In practice, to learn concrete reactive systems, a *test-driver* has to translate the generated queries composed of abstract and parameterized symbols into concrete system interaction and conduct the actual invocations. In turn, the produced (concrete) system output has to be gathered and translated into abstract output symbols. Figure 4.2 shows the essential components of such a test-driver, embedded into a learning setup.

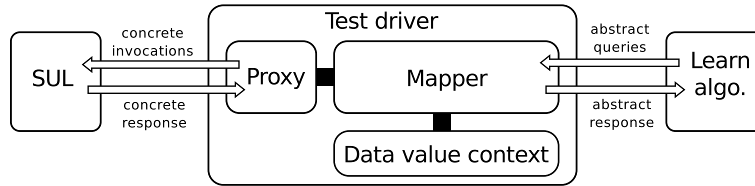

Figure 4.2: Schematic view of a test driver for learning a reactive system.

- A *mapper* is responsible for translating abstract queries generated by the learning algorithm into concrete queries comprised of actions that can be executed on the SUL. For parameterized actions, fitting valuations have to be inserted. Mappers are discussed, e.g., in [44].
- To fill in values for parameterized actions, a *data value context* maintains a set of value instances, that can be stored, retrieved and updated by the mapper.
- The *proxy* maintains a connection to the SUL and interacts with the SUL on behalf of the test-driver, using the concretized parameterized actions created by the mapper. Invocation results are gathered and returned to the mapper, which creates fitting abstract output symbols. For remote services which deliver an interface description in a standardized format (for instance, WSDL), such proxies can often be generated using specialized tools.

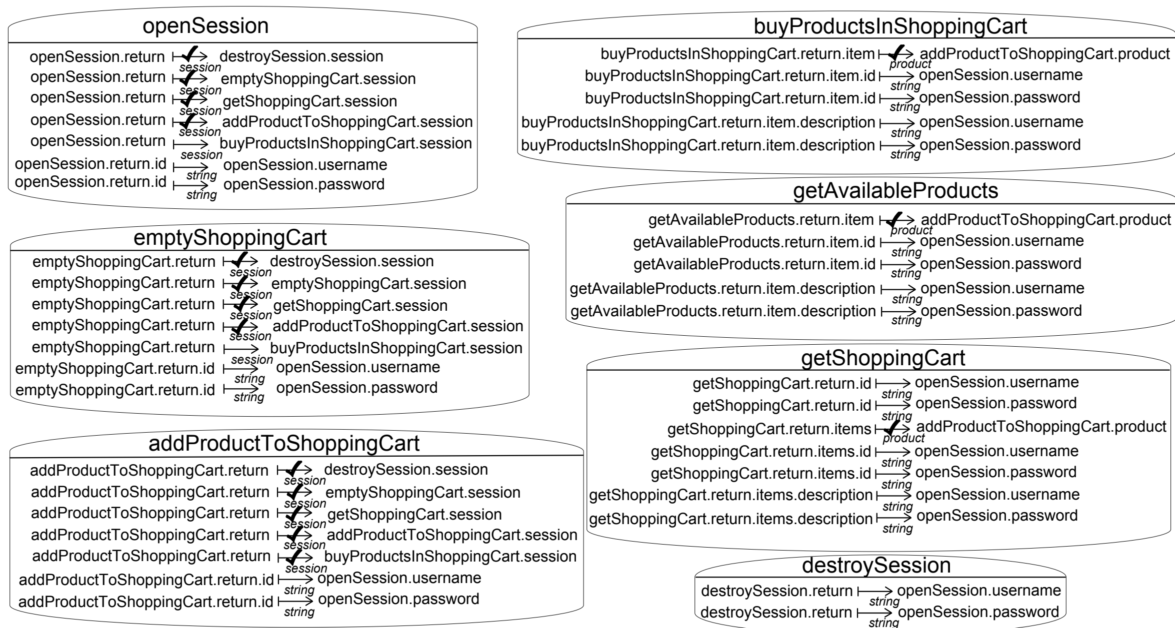

Figure 4.3: States of the dependencies automaton produced by StrawBerry

A central bottleneck of current practice is that the test-driver components such as the mapper must be constructed manually for any specific SUL. This is overcome by our approach which uses a generic mapper that is automatically instantiated with information derived from the syntactic interface analysis performed by StrawBerry.

4.3 Motivating Example

The explanatory example that we use in this chapter is a WS called `EcommerceImplService`[1]. This WS simulates a small e-commerce service, where clients can open a session, retrieve a list of products, add

[1] http://vulpis.cs.tu-dortmund.de:9000/ecommerceservice.

products to a shopping cart and finally conclude buying the items previously added to the cart. This example service is considered to be representative for a significant part of services that are targeted by the CONNECT project in term of complexity and technology and has the distinguished property of modeling a very familiar use case. The following operations are defined in the WSDL interface description:

• `openSession`: this operation is used by registered users to login into the WS. The operation gets the `username` and `password` as input and returns a `session`. `session` is a complex type composed of a session `id` and `creationTime`.

Input data	Output data
`user: string;` `password: string;`	`return: session;`

• `destroySession`: this operation gets as input a `session`, destroys this session, and returns a string denoting success.

Input data	Output data
`session: session;`	`return: string;`

• `getAvailableProducts`: this operation gets no inputs and returns `productArray`, i.e., a list of products, where a `product` is a complex type composed of the product `id`, its `description`, and its `price`.

Input data	Output data
	`return: productArray;`

• `emptyShoppingCart`: this operation gets as input a `session`, empties the shopping cart, and returns the current `session`.

Input data	Output data
`session: session;`	`return: session;`

• `getShoppingCart`: this operation gets as input a `session` and returns the current `shoppingCart`. `shoppingCart` is a complex type composed of a cart `id`, a list of `products`, and the `price`.

Input data	Output data
`session: session;`	`return: shoppingCart;`

• `addProductToShoppingCart`: this operation gets as input a `session` and a `product`, adds the product to the shopping cart, and returns the current `session`.

Input data	Output data
`session: session;` `product: product;`	`return: session;`

• `buyProductsInShoppingCart`: this operation gets as input a `session`, buys the array of products contained into the shopping cart and returns this array.

Input data	Output data
`session: session;`	`return: productArray;`

The particular implementation of this service has the following three semantic properties, which we will use for the illustration of our method. We will see that the integrated approach finds them all.

- The operation `emptyShoppingCart` will return successfully even if the shopping cart was empty already, as long as a valid session is provided.

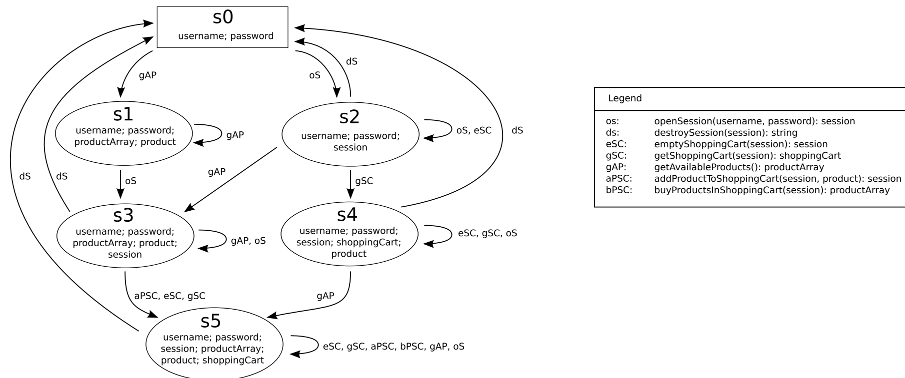

Figure 4.4: Model created by StrawBerry. The edge labels are abbreviated for improved readability.

- In contrast to this, the operation `buyProductsInShoppingCart` will only successfully conclude if the shopping cart connected to the current session is not empty. Otherwise an error will be raised.

- The shopping cart is emptied on successful invocations of `buyProductsInShoppingCart`.

This behavior was modeled to reflect actual web shops. That is, web shops usually do not allow for empty orders, as sending, e.g., empty packages to customers will nonetheless inflict costs. Performing a clearing operation on an empty shopping cart, however, is not hurtful. Upon concluding a purchase, customers will expect a "fresh" shopping cart, so they can resume shopping without having to worry about potentially shopping items twice.

There are several reasons why we chose to use a simulated e-commerce service over, e.g., an actual e-commerce service available on the Internet. First, public e-commerce services usually do not offer an experimental mode where orders will not actually result in costly deliveries and extensive test runs during the extrapolation of the service will not be interpreted as, e.g., a denial of service attack. Second, the simulated e-commerce service is comparatively small, which allows for easy comparison of the extrapolated models with the actual implementation.

4.3.1 StrawBerry at work

In this section we show StrawBerry at work with the `EcommerceImplService` WS. Figure 4.3 shows states of the dependencies automaton produced by StrawBerry. Each state contains dependencies that each operation has with other operations. Dependencies marked with ✓ represent dependencies that are validated by testing activities. Figure 4.4 shows the obtained behavioral automaton. In our approach, it is both necessary and reasonable to assume that, for some of the WSDL input parameters, a set of meaningful values, called an *instance pool* [31], is available. Nodes of the behavioral automaton contain the maturated "knowledge", i.e., the data that are provided with the instance pool or that are obtained as result of previously invoked operations. The `S0` state contains only information that comes from the instance pool, i.e., `username` and `password`. In `S0` only `openSession` and `getAvailableProducts` can be invoked. Once invoked the `openSession` operation, the service reaches the state `S2` in which `session` is available, since it is returned by the `openSession` operation. Similarly, by executing `getAvailableProducts` the service reaches the state `S1` in which both `productArray` and `product` are available since `productArray` is the return value of `getAvailableProducts` and `product` is nested into the complex type `productArray`.

Let us now focus on the state `S5`; in this state each operation can be invoked. Here it becomes apparent that the automaton is not a behavioral model of `EcommerceImplService`. An operation revealing this is, for instance, `buyProductsInShoppingCart`, which might fail when the shopping cart is empty. In other words, there exist a sequence of operations that might lead to `S5` with an empty cart. The lack of behavioral information in the produced model can be attributed to the fact that web service interfaces are not concerned with describing behavioral aspects and thus provide incomplete information to any analysis approach merely focusing on interfaces. Accordingly, StrawBerry was only able to find the first behavioral property described above since it can be described as an invariant on (i.e., partial order of) the legal

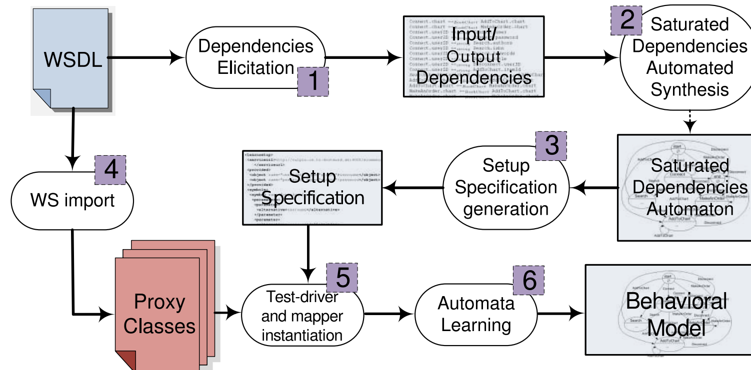

Figure 4.5: Integration of StrawBerry syntactic analysis (steps 1-2) and LearnLib (steps 5 and 6). Step 3 is a newly added feature. `wsimport` **provides proxy classes to interact with the target system.**

sequences of invocations: `getSession` has to be performed prior to `getShoppingCart`, which can only be performed prior to `destroySession`.

As discussed in the following sections, the approach that we present in this chapter overcomes this limitation.

4.3.2 LearnLib at work

LearnLib employs active automata learning algorithms that belong to the family of L^*-like algorithms. Models are inferred by actively gathering observations from test executions. In principle, this approach works without having any information about the syntactic structure of a system's interface: In the worst-case the alphabet of the learning algorithm could be comprised of arbitrary bit-strings. This, of course, does not bode well regarding the chances of creating an useful model. For one thing, experimentation will be much too expensive (in terms of tests). Also, the models will not be insightful at this level of abstraction. In fact, even if queries are assembled from an alphabet generated at the level of the WSDL without any notion of syntactic correctness, learning will still be to expensive and not lead to meaningful models.

To be able to learn models for systems on a sophistication level of the discussed example system, it is necessary to handle data dependencies of the actions to be invoked. This means that the abstract alphabet symbols in fact are parameterized, with fitting valuations being inserted at runtime and returned data values being retained as needed. This is done in the test-driver by the mapper component with data values being organized in a data value context, as discussed in Sect. 4.2.2.

In current practice, both the learning alphabet and the according mapper are constructed manually. This can be a time-consuming task, with, for example, more than a quarter of the total effort being attributed to theses tasks in [67]. This manual approach of creating automata learning setups induces clear limitations on where automata learning can be employed. For example, this is unsustainable in scenarios where behavioral models are to be learned automatically for a wide range of systems, which is a requirement for employing automata learning in the context of the CONNECT project.

4.4 The Integrated Approach

As outlined above, models created by means of syntactic analysis do not capture behavioral information well, while active learning observes actual system behavior and constructs models that include the observed behavioral aspects. However, the construction of mappers that enable learning setups to deal with data dependencies can be tedious.

In the integrated approach, to allow construction of an alphabet and a mapper accounting for data flow concerns, the interface analysis results produced by StrawBerry are handed over to LearnLib in form of an artifact called *setup specification* (an overview is given in Figure 4.5). The setup specification artifact is the main connection point between StrawBerry and LearnLib and contains the following information: (i)

the URL of the service to be learned; (ii) predetermined data values for an instance pool; (iii) alphabet symbols which refer to parameterized actions on the target system; (iv) parameters and return variables for each alphabet symbol.

The alphabet symbols and the corresponding parameters and return variables are generated from the result of StrawBerry's syntactic analysis. We recall that Figure 4.3 shows the states of the saturated dependencies automaton produced by StrawBerry and syntactic dependencies that each operation has with the other operations. Dependencies that are marked with ✓ identify dependencies that would be validated by StrawBerry if testing is performed (as described in Section 4.3.1). This information is used in the integrated approach to determine the data-flow between method invocations and to choose parameter and return variables for the setup specification.

This means that information on data dependencies between operations, as deduced by StrawBerry, are used to construct an alphabet of parameterized actions. This allows for carrying enough information so that the mapper can translate abstract alphabet symbols into concrete actions outfitted with live data values and manage return values of invocations.

To illustrate how an automated learning setup can be instantiated with the help of the generated setup descriptor (an activity represented as *Test-driver and mapper instantiation* in Figure 4.5), it is helpful to recall what concerns have to be addressed:

- Means for instrumentation of the SUL have to be provided, e.g., by means of a proxy that is accessible by the test-driver.
- An alphabet for the learner has to be constructed, as well as a mapping between the abstract alphabet and concrete system actions. This is where the dependency information provided by StrawBerry is essential.
- Facilities for handling communicated data-values have to be present and configured to account data-flow between operations.

In the following we will discuss these points in more detail:

Instrumentation Within a setup for active automata learning, the instrumentation layer is responsible for injecting system stimuli and gathering the target system's output for every invocation. For WSDL services, injecting system stimuli can be done in a straightforward way, e.g., by using automatically generated proxy classes that expose system functionalities while hiding the specifics of operating the target system through networked messages. For the experiments discussed in this chapter, proxy classes for the remote system are generated by the `wsimport` [56] utility, which can serve as instrumentation layer for the test driver (denoted as *WS import* activity in Figure 4.5).

Determining an alphabet and mapper configuration The interface description is a natural source for the alphabet employed for the learning process, as every message defined in the WSDL description usually has a direct mapping to system features intended for remote consumption. It appears most sensible to choose the names of the defined WSDL messages as abstract alphabet symbols for the learner, which the test-driver concretizes into actual operation invocations of the generated proxy classes. As such the mapping between the abstract learning alphabet and concrete system input is one from operation names to actual invocations of the corresponding operation.

For parameterized operations, abstract alphabet symbols also have to include information for the mapper on how to retrieve values from the data value context to enable actual invocation. Thus the abstract symbols for parameterized operation calls will contain references to this context in form of instructions on how to retrieve data values from it.

To populate the data value context, data values returned by the SUL will be stored as named variables. Thus the abstract symbols also have to contain information on the name of the variable the return value is assigned to. For each stored value the abstract output symbol forwarded to the learner will simply be the variable name in which the return value was stored, abstracting from the actual content of the return message that the system under test produced. A similar approach to abstraction is taken for error

messages: if the SUL produces an elaborate error message, the output returned to the learner usually will be a generic "error" symbol, abstracting from all the details related to this error instance. No data value will be stored in this case.

Storing and accessing data-values When concretizing learning queries into actual system input, fitting data values have to be inserted into parameterized system messages. Thus the system driver has to be able to store received data values and generate concrete system input by resorting to these stored values.

To accommodate data values, the data value context is realized as an embedded JavaScript environment. The reason for choosing a JavaScript environment over, e.g., a map of variable names and variable values, lies in the ability of a scripted context to access stored data with utmost flexibility. A scripted data value context is, e.g., able to access fields of complex data structures and provide those as parameter values.

Not every parameter can be filled with data values that are results of preceding system invocations. One notable example for this are login credentials, which have to be known beforehand. Such values have to be included in the setup specification and are copied into the data value context.

4.5 Application to the Example and Discussion

In the following, we will apply the presented approach to the running example.

Figure 4.6 shows an excerpt of the setup description created by StrawBerry as a result of the interface analysis. The `serviceurl` declaration in line 2 provides an URL to the SUL, which can be directly used as an input for `wsimport`. Predetermined values (credentials in this case) are provided in lines 3 to 6 and are used to populate the instance pool.

The remainder of the specification file defines a sequence of symbols. Each `symbol` includes a sequence of `parameter` declarations, which refer to named variables in the data value context. It can be seen that the `symbol` declarations include information on parameters and on the variables where return values are stored. Parameter values stored in the data value context are addressed by named keys that are specified by the `alternative` environment. The reason for having `alternative` declarations is that parameters may have several potential data sources. For example, the second parameter of the symbol `addProductToShoppingCart` may take data values from the variables `productArray` and `shoppingCart`. Each alternative induces the instantiation of additional abstract symbols, meaning that for the presented example the learning alphabet has in fact two `addProductToShoppingCart` symbols, one referring to `productArray` as parameter value, the other referring to `shoppingCart`.

The parameters of the symbol `addProduct- ToShoppingCart` illustrate why realizing the data value context as scriptable environment is advantageous: the `alternative` declaration in line 31 of Figure 4.6 includes attributes that specify how the data value for the corresponding parameter has to be extracted from the context. Instead of directly filling in the parameter value with the complete data structure that is pointed by the variable `productArray`, only the `field` "item" of this data structure should be considered. However, the field "item" references a set of products and not a single product. Thus, the `selector` "elementOf" is specified as well. From this information the JavaScript expression `elementOf(productArray.getItem())` is derived and evaluated on the data value context at run time, where the function `elementOf()` is predefined and simply returns the first value of any provided collection.

The result of learning a behavioral model with this setup specification is shown in Figure 4.7. Please note that this figure presents a view onto the learned Mealy machine that omits error transitions, only showing actions that do not raise an exception during execution. The impact of the experimental semantical analysis is already apparent from the fact that this model contains more states than those created by StrawBerry by means of syntactic analysis and test runs, with the effect that all the three properties mentioned in 4.3 are correctly revealed:

- As long as a session is open, it is possible to empty its associated shopping cart.
- When no products have previously been added to the shopping cart, the operation to purchase products does not conclude successfully.

Service Name	# of MQs	Syst. invoc. for MQs	# of EQs	Syst. invoc. for EQs	# states	Exec. Time (in sec.)
EcommerceImplService	1.259	9.558	3	21.157	8	660
DictionaryServiceImplService	6.893	104.310	3	22.041	16	1.891
PoliceWarnServiceImplService	187	1.171	3	21.030	5	601

Table 4.1: Experiment results

- After a purchase operation it is not possible to immediately trigger another purchase. Instead, it is necessary to put another item into the shopping cart. This means that the purchase operation does clear the shopping cart.

Apart from these facets even more subtle behavioral aspects are captured. For example, once a non-empty shopping cart is retrieved, its contents can be added to another session's shopping cart. This means that the data structure representing products in a shopping cart is not bound to session instances, which is another implementation detail influencing how the service can be operated that is not explicitly contained in the service's interface description.

4.6 Experiments

In CONNECT, learning technology has to be able to deal with very different services. To further evaluate the applicability and feasibility of the presented approach, we applied our approach on additional web-services. For each service we generated a behavioral model and compared the produced automata with the service implementation.

4.6.1 Case study subjects and experimental setup

We decided to create mock-up services to avoid negative side effects on the availability of production services. Moreover, the terms of use[2] of many services include restrictive invocation quotas, which makes experimentation aiming at complete system exploration difficult. For example, learning the e-commerce service requires approximately 30.000 system invocations including conformance testing. Also, some services employ a pay-to-use business model and thus are not available for experiments. Finally, not every productive service provides testing environments and thus active testing can trigger costly effects (like, e.g., purchasing actual products). On the other side of the medal, by employing mock-up services we can actually compare the created models with the service implementation, allowing for verification of the learned models against the actual implementations of the mock-ups. Threats to validity stemming from this approach are discussed in Section 4.6.3.

The performed experiments involve three services:
(i) `EcommerceImplService`[3]: this is the running example already introduced in this chapter. It represents a typical use-case of web services, yet intuitive.
(ii) `DictionaryServiceImplService`[4]: this service is an extended version a real-life dictionary web service. After login, a list of words can be retrieved, for which definitions are available. Similar words can be retrieved according to a strategy selected from a list of available strategies.
(iii) `PoliceWarnServiceImplService`[5]: the mock-up represents a hypothetical service where messages can be submitted for approval and delivered to police officers in a specified area.

The experiments have been conducted on a portable computer with an Intel Core i5-2520M CPU with two cores and a base frequency of 2.5 GHz. The running time of experiments, however, is influenced by the accumulated response time of the remote services. The services were hosted on a dedicated Pentium 4 PC with 3 GHz, connected via Gigabit Ethernet. The experiments have been conducted several times to ensure the provided numbers are representative.

[2]such as: http://services.aonaware.com/DictService/tos.htm
[3]http://vulpis.cs.tu-dortmund.de:9000/ecommerceservice?wsdl
[4]http://vulpis.cs.tu-dortmund.de:9500/dictionaryservice?wsdl
[5]http://vulpis.cs.tu-dortmund.de:9100/policewarnservice?wsdl

4.6.2 Results

Table 4.1 reports the results of the experiments, showing the number of membership queries (MQs), the number of system invocations caused MQs, the number of equivalence queries (EQs), the number of system invocations caused by EQs, the number of states of the produced behavioral automaton and the execution time in seconds. The produced behavioral automata have been inspected by the mock-up implementer and found to conform to the programmed behavior. As can be seen, a significant number of system invocations is necessary to create a model. Depending on system size, the number of system invocations caused by EQs can outnumber the number of invocations caused by MQs which are used for actual model construction. For all services, the time needed to produce a model was well below 35 minutes.

4.6.3 Threats to validity

The results of our experiments are subjects to the following threats of validity:

• As only a limited set of services have been evaluated, we cannot claim that the results of our experimental evaluation are generalizable. The evaluation should be considered as investigating the potential of the technique rather than providing a statement of effectiveness. However, the performed evaluation indicates that the approach is workable and produces accurate results.

• The experiments have been performed by considering mock-up services. Those services do not have business back-ends, but employ a complete WSDL technology stack and are behaviorally indistinguishable from deployed real-life services. Also, the mock-ups have been created taking into account similar services available in the market. For instance, the dictionary-service mock-up recreates a real-life service in extended form. Please recall that employing mock-ups was necessary as actual services come with restrictive terms of service or may mistake the active learning phase for a malicious attack due to the number of system invocations.

• The tools we have used or implemented could be defective. We employed, however, preexisting tools with a proven track-record: StrawBerry is a research tool existing for over two years and LearnLib is a stable and publicly available tool used by several independent research groups. The results obtained by the combined approach were subject of manual examination and checks.

• The results presented in Table 4.1 indicate that the combined approach does not produce ad-hoc results, but that results are produced within a time frame that is perfectly acceptable for off-line analysis. It is important to note that network latency does have an immediate impact on the concrete running time of learning setups.

4.7 Conclusions and Perspectives

We have presented a method and tool to *fully automatically* infer dataflow-sensitive behavioral models of black-box systems based on interface descriptions in WSDL by combining StrawBerry, a tool for syntactical analysis of the interface descriptions and the LearnLib, a flexible active automata learning framework. This combination allows us to overcome a central bottleneck, the manual construction of the mapper required for the learning tool to bridge between the model level and the concrete execution level. This is particularly important for CONNECT, where full automation of the analyses is a must, as manual construction of parts essential to the learning setup is clearly in contradiction to the overall project goal of automatically creating system connectors. Thus the presented approach is an important enabler for the applicability of learning technology within the CONNECT architecture.

Our method has been illustrated in detail along a concrete shop application and quantitatively evaluated on three web-services. The results are promising, but further case studies are required to fully explore the application profile of the approach. Scalability is certainly an issue here, and it has to be seen how stable the approach is concerning varying versions of WSDL-based interface specifications. Particularly interesting is here to investigate how our approach may profit from extra information provided e.g. through semantic annotations, a point explicitly addressed also in the CONNECT context. There, full

automation is not sufficient as CONNECT's support is meant to happen fully online. Finally, we are currently working on an extension of our technology to generate even more expressive models in terms of register automata [23]. These models are designed to make the currently only implicitly modeled dataflow information explicit by introducing transitions with explicit conditions and assignments.

```
 1 <learnsetup>
 2  <serviceurl>http://vulpis.cs.tu-dortmund.de:9000/ecommerceservice?wsdl
      </serviceurl>
 3  <provided>
 4   <object name="username" type="string">username</object>
 5   <object name="password" type="string">password</object>
 6  </provided>
 7  <symbols>
 8   <symbol name="openSession">
 9    <parameters>
10     <parameter>
11      <alternative>username</alternative>
12     </parameter>
13     <parameter>
14      <alternative>password</alternative>
15     </parameter>
16    </parameters>
17    <return>session</return>
18   </symbol>
19   ...
20   <symbol name="getAvailableProducts">
21    <parameters />
22    <return>productArray</return>
23   </symbol>
24   ...
25   <symbol name="addProductToShoppingCart">
26    <parameters>
27     <parameter>
28      <alternative>session</alternative>
29     </parameter>
30     <parameter>
31      <alternative selector="elementOf" field="item">productArray
         </alternative>
32      <alternative selector="elementOf" field="items">shoppingCart
         </alternative>
33     </parameter>
34    </parameters>
35    <return>session</return>
36   </symbol>
37  </symbols>
38 </learnsetup>
```

Figure 4.6: Excerpt of the setup specification for LearnLib generated by StrawBerry by means of syntactical analysis.

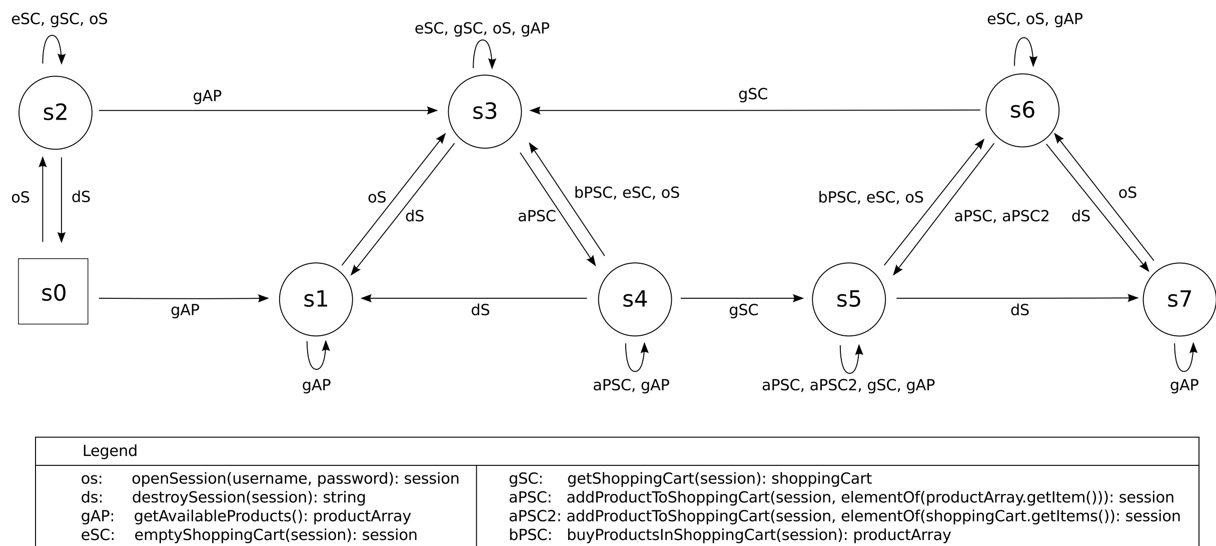

Figure 4.7: Model created by LearnLib using the setup description created by StrawBerry. The edge labels are abbreviated for improved readability.

5 Implementation and Integration of the Learning Enabler

In the previous chapters, we described techniques for inferring behavioral models of networked systems, along with an approach to adequately handle either incompletely learnt models or changes in the behavior of the system. Integrating these into the overall CONNECT architecture was one of the particularly important tasks in Y3.

Most of the tools and libraries developed in this WP were designed to operate on a stand-alone basis. While providing a high level of flexibility, this approach imposes hurdles when deploying these components in a tightly integrated environment as in CONNECT. One of the major problems that has to be dealt with is a mismatch in terms of data structures and formats.

In this chapter, we thus describe the development of the *Learning Enabler*, the component responsible for making the behavioral learning functionality available to the other CONNECT enablers, as well as bridging the gap between the individual tools and libraries, and the overall CONNECT setting. Hence, the Learning Enabler does only partially introduce new functionality, but rather acts as a "glue" to provide a homogeneous interface to the single components.

5.1 Conceptual Overview

A high level view of the CONNECT architecture is described in Section 5.1 of D1.1: The Learning Enabler takes *interface descriptions* of networked systems, 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).

Furthermore the Learning Enabler will take as input *information about metrics* that are measurable when interacting with the networked systems and guarantees the systems make concerning those metrics. From this and through measurements made during the course of learning, the Learning Enabler will produce refined or enriched data concerning the guarantees as a secondary output (cf. D4.2).

The Learning Enabler cooperates closely with the monitoring system, developed in WP5 (cf. D5.1, D4.2), in order to get information about the (in-) correctness of inferred models. The Learning Enabler will therefore provide the monitoring system with corresponding conditions, which are to be checked on the running system. In case such conditions are violated, the Learning Enabler will be notified and the (incremental) process of inference will be reissued.

The envisioned control-flow between the enablers is as follows. The discovery enabler will inform the Learning Enabler about a new (or updated) networked system, of which a new (or updated) behavioral model, and non-functional properties have to be inferred. The Learning Enabler will infer the desired information and forward it to the discovery enabler.

The discovery enabler will issue the generation of a CONNECTor using the information gathered about two networked systems to be connected. On deployment of the actual CONNECTor, the corresponding (deploying) enabler will notify the Learning Enabler about the new CONNECTor, and alongside provide information about the connected networked systems. From this information, the Learning Enabler will generate conditions to be monitored by the Monitoring Enabler, and instruct the Monitoring Enabler to do so.

The Monitoring Enabler will issue events, which the Learning Enabler will subscribe to. These events will be used retrace the steps in the models of the networked systems that are taken by the CONNECTor in order to find discrepancies between model and actual system. If such a discrepancy (a counterexample) is found, the learning enabler will start a new inference process for the networked system, using the counterexample as additional input. The updated model of the networked system will be sent to the discovery enabler.

The role of the Learning Enabler during the connection phase (and therefore omitting the Monitoring

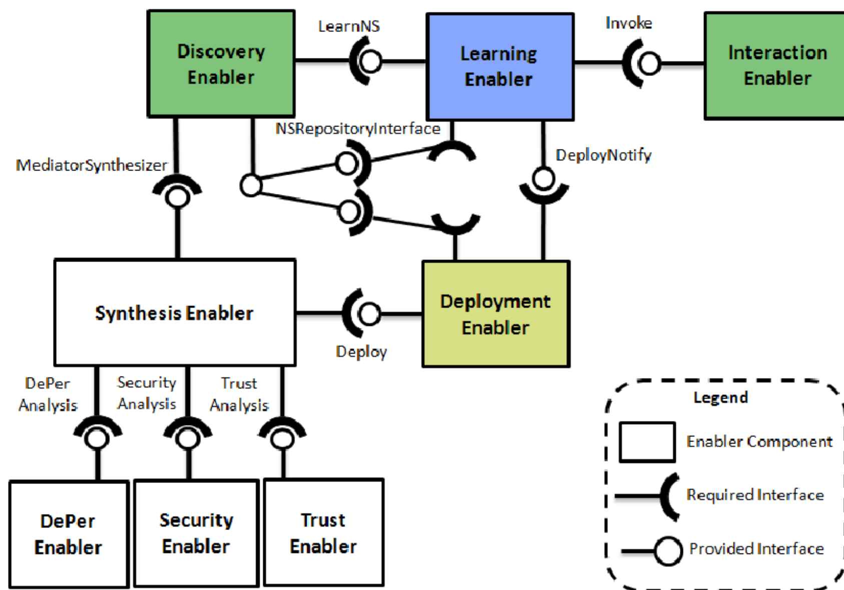

Figure 5.1: The Learning Enabler in the context of the CONNECT enabler architecture

Enabler) is displayed in Fig. 5.1. Enablers with which the Learning Enabler interacts directly are highlighted with green color, while the ones with which only an indirect interaction (i.e., via subscription to a message bus) takes place are highlighted in yellow. However, "direct" interaction does not necessarily mean that those enablers have to run on the same system (nor is this envisioned), but possible network transfer will be hidden by JMS wrapper classes (cf. D1.3), which implement the enabler's public interface and thus provide a transparent realization of the communication between the enablers.

5.2 Structure of the Learning Enabler

The task of the Learning Enabler is to aggregate the functionality of the following tools and libraries:

- **LearnLib** is an active automata learning framework. It includes algorithms for learning DFAs, Mealy machines and also register automata (cf. Chapter 2). It provides a set of interfaces for flexible interaction with arbitrary systems. LearnLibprovides its own formats for outputting the inferred automata.
- **StrawBerry** is a tool for analyzing interface descriptions from a WSDL and employs type analysis in order to produce a *Behavior Protocol Automaton*. A learning setup that can be used by an active learning algorithm is also a result of this analysis. This process is described in further detail in Chapter 4.
- Furthermore, it comprises various adapters and transformers for realizing the communication with other CONNECT enablers, the Discovery and Interaction enablers.

Fig. 5.2 depicts the data flow during the typical operation of the Learning Enabler, triggered by receiving new or updated information on a networked system's interface by the discovery enabler. By means of the StrawBerry tool, a learning setup description (see below) is generated, which will serve as a basis for the actual learning process. The learning algorithm interacts with the networked system using the abstraction provided by a test driver, which relies on the Interaction Enabler to invoke operations on the networked system. Eventually, the learning algorithm will output a Mealy machine, which is—besides being stored in an internal repository—sent back to the Discovery Enabler, after having been transformed to an LTS format compatible with other CONNECT enablers.

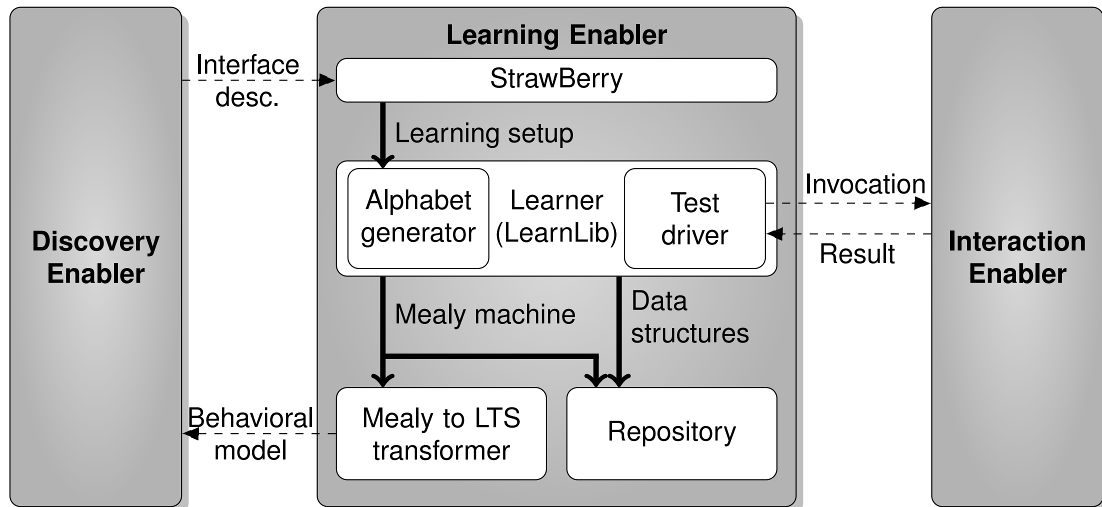

Figure 5.2: Data flow during operation of the Learning Enabler

Though the Learning Enabler also communicates with the Deployment and Monitoring Enablers, the use case described above is the only one in which other CONNECT enablers actively interact with the Learning Enabler. For this reason, the public interface exposed by the Learning Enabler, which is exclusively used by the Discovery Enabler, is very slim:

```
public interface LearnNS {
    public void learnCNSBehavior(String cnsId);
}
```

This single method is invoked by the Discovery Enabler and triggers the process sketched in the above Fig. 5.2. In the following subsections, we want to explain the components that are treated elsewhere in this deliverable in more detail.

5.2.1 Alphabet generation and test driver

The StrawBerry tool produces a learning setup, which serves as a basis for both the generated learning alphabet and the test driver. Since the latter is responsible for interpreting the symbols from the alphabet in order to communicate with the networked system, these two components are closely interconnected.

For an illustration of the process of alphabet generation, consider the XML fragment depicted in Fig. 5.3, which is part of the whole learning setup that was shown in Fig. 4.6 on p. 68. From this fragment, two *concrete* symbols will be generated: `addProductToShoppingCart(session, elementOf(productArray.getItem()))` being the first one, and `addProductToShoppingCart(session, elementOf(productArray.getItems()))` the second one. In this fashion, symbols are generated for each of the `<symbol>` tags contained in the learning setup.

To the generic learning algorithm, which is provided by LearnLib, these appear in a purely symbolic way: it neither recognizes that the above two symbols refer to the same operation on the networked system, or that an element is selected from the result of `productArray.getItem()`, nor has it to. This of course is not feasible at the point where those symbols have to be ‘executed’ in order to interact with the networked system: here, we need a *test driver* which is responsible for interpreting the symbols from the queries posed by the learning algorithm, as well as transforming the output received into plain symbols. The test driver implements a very generic interface provided by the LearnLib API, such that the learning algorithm does not need to know about *how* the symbols are being interpreted.

Considering the symbol `addProductToShoppingCart(session, elementOf(productArray.getItem()))`, execution concerns the following aspects:

```
25   <symbol name="addProductToShoppingCart">
26    <parameters>
27     <parameter>
28      <alternative>session</alternative>
29     </parameter>
30     <parameter>
31      <alternative selector="elementOf" field="item">productArray
            </alternative>
32      <alternative selector="elementOf" field="items">shoppingCart
            </alternative>
33     </parameter>
34    </parameters>
35    <return>session</return>
36   </symbol>
```

Figure 5.3: Fragment of the learning setup

- the operation `addProductToShoppingCart` has to be invoked on the *remote* networked system, but first
- the parameter values have to be calculated *locally*, from the expressions `session` and `elementOf(...)`.

In the course of Sec. 4.4, the tool `wsimport` was employed in order to generate proxy classes that allowed for abstraction from the middleware layer (SOAP in the case of web services). In a heterogeneous environment as in CONNECT, this however is not feasible: a middleware-independent way of invoking operations on a remote networked system is required. In the CONNECT architecture, this functionality is provided by the *Interaction Enabler* (cf. D1.3), which provides the following interface:

```
public interface Invoke {
  public Object[] invoke(URL serviceID, String operation, Object[] parameters,
      String[] returnTypes) throws Exception;
}
```

As this solves the former of the above two tasks, we now need to have a closer look at the latter, i.e., creating an adequate array to pass as `inputParameters`. As already stated in Sec. 4.5, the test driver needs to maintain a *variable context*, where it stores the values for, e.g., `session` and `productArray`. This is realized by means of a JavaScript engine, which also handles the evaluation of complex expressions such as `elementOf(productArray.getItem())`. The return values (if any) of the invocation are then stored into this context as well.

In a future version, among the responsibilities of the test driver will as well be the gathering of statistical information in order to infer non-functional properties, which is not yet implemented. Another obstacle is that, with the current interface offered by the Interaction Enabler, it is not yet possible to distinguish between an error which occured at the transport layer and an error intentionally issued by the remote networked system.

5.2.2 Result transformation

Upon the termination of the learning algorithm, it will return a Mealy machine. Since LearnLib was designed as a stand-alone library and uses its own data structures, this naturally is not the format other CONNECT enablers will understand. Hence, the Mealy machine needs to be transformed into an *Enhanced LTS* (cf. D3.3). This also comprises adding semantic annotations to the model (e.g., references to ontology concepts) for both actions and their parameters. This information is provided by the Discovery Enabler along with the interface description.

The resulting Enhanced LTS is then—along with the ID of the networked system—sent back to the networked system repository (which is part of the Discovery Enabler) by invoking the following interface method (cf. D1.3):

```
public interface NSRepositoryInterface {
  // ...
  public void updateNetworkedSystemBehavior(String nsID, EnhancedLTS lts);
  // ...
}
```

This method takes care of updating the networked system's behavioral model centrally, and might trigger further operations like CONNECTor synthesis. Once the learning of non-functional properties is implemented, those will be sent to the Discovery Enabler at this point as well.

5.2.3 Model and data structure repository

Besides sending the learnt models to the Discovery Enabler, the Learning Enabler also manages its own repository of inferred models (in their original form as Mealy machines). This is necessary to realize the "never-stop learning" approach: when the Deployment Enabler informs about the deployment of a synthesized CONNECTor, the Learning Enabler generates probes for the monitoring subsystem which allow observing the CONNECTed systems. During execution, a *state-tracking* (a further description is given in Sec. 6.3) is performed, until—in case of an incompletely learnt model—a counterexample is found, which triggers a re-learning of the system. Since it would be unnecessarily expensive to re-learn a networked system from scratch, the Learning Enabler also stores the data structures of learning algorithms. This allows for an incremental procedure of refining the behavioral models. Chapter 6 is dedicated to presenting the integration between Learning and Monitoring Enablers in more detail.

5.3 Current Status

All of the features described above—with the exception of inferring non-functional properties, the implementation of which will be the central task for Y4—are implemented in the current version of the enabler. This especially concerns the interoperability of StrawBerry, LearnLib and the other enablers, for which integration has been realized at an interface level: the current implementation uses stub classes implementing the interfaces of the Discovery respectively the Interaction Enabler (cf. D1.3). Hence, provided that these interfaces remain stable, replacing the stubs by the actual enablers can be done with very little effort and should not cause interoperability issues.

A remaining task is to set up the Learning Enabler for listening to GLIMPSE messages about CONNECTor deployment. Apart from that, the actual integration with the Monitoring Enabler—as described in the following Chapter 6—is completely established.

6 Integration of Learning and Monitoring

In this chapter we recall the architecture and functioning of the CONNECT monitoring infrastructure called GLIMPSE, which has been described in detailed way in the previous deliverable D4.2 [26]. Indeed, into the CONNECT project, monitoring constitutes a transversal Enabler with a central role for ensuring context-awareness and driving adaptation. For example, in Deliverable D5.2 [27] it has been already demonstrated for dynamic validation of dependability and performance analysis results. In this chapter we focus on the usage of monitoring for validation and possible adaptation of Learning results.

More in general, a monitoring process involves several activities, including the collection of raw observation data, the interpretation of such raw information to recognise higher-level events that are relevant at the business level, and the effective presentation of the results of the monitoring.

In real systems, the volume of collected raw data can be overwhelming. Moreover, such data needs to be filtered and aggregated to detect deviations from expected behavior and to derive measurements of interesting non-functional properties of the system. There is a gap to be filled when a high-level, business-relevant property is to be turned into a concrete setup of the monitoring infrastructure.

In order to be able to evaluate and keep under control any such property, it is necessary to instruct the monitor about what raw data to collect and how to make sense of it in order to infer whether or not a desired property is fulfilled. Unfortunately, not only this task is time-consuming, but it also requires a substantial human effort and specialised expertise in order to convert the high-level description of the property to observe into lower-level monitor configuration directives. Consequently, the outcome of such effort is very hard to generalise and to reuse, and, as a matter of fact, the resulting monitor configuration typically is only relevant in the specific situation at hand. This process needs to be iterated each time a different architecture needs to be monitored, or when the properties to be monitored change. The specification of Non-Functional properties, may also be exploited using the CONNECT Property Meta Model (CPMM). This model allows the user to define generic and specific models of a non-functional property [**?**]. In order to have all operations strictly connected, we are developing a tool to be able to convert the generated EMF model of the NF-property into a Drools Rule.

The rest of the Chapter is organized as follows: in the next section we introduce the motivation behind augmenting learning conducted before CONNECTor synthesis, with continuous observation of run-time behaviour of the CONNECTed system, after the CONNECTor is deployed. Then we report the status of current implementation of the proposed framework in section 6.2 and describe how the integration between learning and monitoring works 6.3. Conclusions and future work conclude the chapter.

6.1 Motivation

In real life software components that are released as black boxes do not generally provide models; this is particularly true with respect to behaviour. In previous chapters we have presented how active learning techniques are being leveraged to enable the automatic derivation of formal behavioral models for Networked Systems. However, these models are not guaranteed to be correct, which of course may have implications for the correctness of the CONNECTor. In CONNECT we do not have specification about the environment where the application is running and what and how many inputs will be provided by users, thus model-based approaches run the risk to interpret as faults, discrepancies between the estimated output and the measured output that are instead caused by modeling errors. Moreover, even if the constructed model was correct when it was built, the learned Networked Systems might subsequently evolve making the corresponding automata obsolete.

This is why the seamless interaction between the Learning Enabler and monitoring becomes essential to provide context about the run-time behaviour shown by the CONNECTed system, and to highlight potential discrepancy between the learned models and the actual functioning system. Therefore, in the third year our work on monitoring focussed mostly on enabling the interaction with Learning.

To address such goal, we propose a "Never-stop" Learning approach, where learning is not considered complete with the building of the model. Instead, we keep the NSs under observation by run-time monitor-

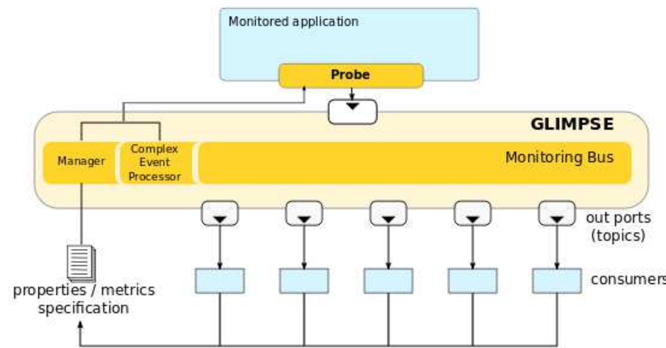

Figure 6.1: The GLIMPSE Architecture

ing in order to capture the event flow of components' interactions within the CONNECTed system, so that we can check whether the actual observed behaviour complies with the hypothesized model. We then attempt to retrace the sequence of monitored observations on the states of the current learned model. If this is not possible, i.e. there are no states in the hypothesis that can explain the observed interactions, this provides evidence of a mismatch between the learned hypothesis and the actual target systems.

6.1.1 GLIMPSE Infrastructure

For this chapter to be self-contained, we will briefly recap the main components of the Monitoring system used in this scenario. Its infrastructure is composed of several artifacts, structured to be more general and modular as possible and implemented to cover the main five core functions identified into a generic monitoring infrastructure, namely: Data collection, Local interpretation, Data transmission, Aggregation, Reporting.

The infrastructure, shown in figure 6.1, is totally generic and can be easily applied to different contexts.

Collector/Data Suppliers Probes intercept primitive events when they occur into the CONNECTor and send them on the GLIMPSE Monitoring Bus. Usually probes may be realized injecting code or instrumenting the SUT, in the CONNECT scenario, they are embedded into the CONNECTor. Probes may also be configured to use a primitive event filter in order to reduce the amount of generated raw data.

Communication Bus The Monitoring Bus is the communication backbone where all information (events, requests, answers) is sent by: Probes, Consumers, Complex Event Processor and by all the services querying information to GLIMPSE. Looking at the common infrastructure of the monitoring systems, the generated events, flowing through a bus, this bus is usually a middleware classified in Message Oriented Middleware, a complement of the event processing systems. MOM provides the transport layer for event processing and realizes the infrastructure to implement event channels. The events circulating on a MOM may be represented as messages. They have temporal semantics and some extra information about routing, expiration time and ordering.

The MB adopt a publish-subscribe paradigm and all the communications are managed by the Manager component; In the current GLIMPSE implementation, the system backbone is implemented by means of ServiceMix4 [4], an open source Enterprise Service Bus, used to combine advantages of event-driven architecture and service-oriented architecture functionality.

Complex Event Processor The Complex Event Processor (CEP) is the rule engine instrumented through enablers evaluation requests. It infers complex pattern analyzing primitive events generated from the probe inside the CONNECTor. There are several rule engines that can be used for this task (like Drools Fusion [1], RuleML [3]). In the current GLIMPSE implementation, we adopt the Drools Fusion rule language [1] that is open source and can be fully embedded in the realized Java architecture. The proposed

flexible and modular architecture allows for easily replacing this specific rule language with another one.

Consumer (CONNECT Enabler) It may be a learning engine, a dependability analyzer or a simple customer that requests some information to be monitored. It sends a request to the Manager using the Monitoring Bus and waits for the evaluation results on a dedicated answer channel provided by the Manager.

Manager The Manager component is the orchestrator of all the GLIMPSE architecture. It manages all the communications among the GLIMPSE components. Specifically, the Manager fetches requests coming from enablers, analyzes them and, if necessary, instructs the Probes. Then, it fill the knowledge base of the CEP Evaluator, with the request and creates and notifies to the enabler a dedicated channel on which it will provide results notified by the CEP evaluator.

6.2 GLIMPSE implementation

The communication paradigm adopted by Monitoring and Learning is publish-subscribe, to provide a better communication decoupling. There are also monitoring systems that interact using the request-response in- teraction pattern [42], usually used into security environment or when interacting with all REST-style web services [58], most SOAP-style and is the base for the Remote Procedure Call (RPC) [2] invocation pat- tern. The request-response pattern is often the natural way to design an application, when you need an additional information, you invoke the component or service to return it synchronously.

Examinating the five main component of the monitoring infrastructure, the most relevant may be noticed into the Complex Event Processor module, that allow to do more that just gather and collect information. This module, enables consumers (users of the monitoring) to instrument a knowledge base with inference rules and to define complex pattern to match.

As mentioned in Complex Event Processor description, the CEP may be replaced with another one. To allow the instrumentation of the CEP independently, we defined a generic XSD (ComplexEventRule) 6.1 that contains all the necessary information to interact with the specific knowledge base used. This rule will be loaded on the GLIMPSE knowledge base.

On a event-based monitoring infrastructure, as GLIMPSE is, the gathered information are provided in form of events. An event, is an atomic description, a smaller part of a more large and complex process at application level; in our scenario, an event represent a method invocation to a networked system, the invocation coming from the producer to the consumer, is captured when comes through the CONNECTor, incapsulated into an GlimpseBaseEvent object and sent through the Enterprise Service Bus, by means of ServiceMix.

The detailed structure of a GLIMPSE Event is described in figure 6.2

Our probes are a non intrusive data collector (proxies), they have no effect on the order and timing of events in the application and do not generate overload on the communication or on the services that are interacting with.

The interaction between Learning and Monitoring can be analyzed through a simple sequence diagram, as shown in Figure 6.3.

The interaction starts when the Learning Enabler sends a JMS message whose payload contains an XML object rule generated using ComplexEventRule classes (see Figure 6.1).

The Monitoring Enabler, upon receiving the request, properly manages the probes embedded in the CONNECTor.

Once the CONNECTor is deployed, data derived from real executions are sent on the Monitoring Bus and requested event pattern found notified to the Learning Enabler. The Learning Enabler, in turn, performs an analysis of the monitored observations and uses such information to check if the model analysed before deployment is still accurate.

If the model parameters are found to be inaccurate, or new behaviors emerge, Learning updates the

Listing 6.1: The Complex Event Rule XSD

```
<?xml version="1.0" encoding="UTF−8"?>
<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 name="Delete" type="tns:ComplexEventRuleType"
       maxOccurs="unbounded" minOccurs="0" />
      <element name="Start" type="tns:ComplexEventRuleType"
       maxOccurs="unbounded" minOccurs="0" />
      <element name="Stop" type="tns:ComplexEventRuleType"
       maxOccurs="unbounded" minOccurs="0" />
      <element name="Restart" type="tns:ComplexEventRuleType"
       maxOccurs="unbounded" minOccurs="0" />
     </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>
```

model with the new values.

If the new analysis evidences that the deployed CONNECTor needs adjustments, the Learning enabler may trigger an adaptation process through Synthesis enabler.

The interaction pattern between Learning and Monitoring is shown as a sequence diagram in Figure 6.3, where we intentionally left out system start-up operations. Whenever Monitoring receives a request message on the service channel, a new channel dedicated to the requesting Enabler is set up to communicate monitored values. Monitoring sends response messages to Learning as soon as the aspect of interest is available. The two Enablers exchange JMS messages whose payload is expressed in XML language.

Each *ComplexEventRule request message* received by the Monitoring Enabler, contains an XML generated through the `ComplexEventRule` XSD (See 6.1) that contains the following fields:

(i) `RuleName`, that specifies the name of the Rule to setup; (ii) `RuleBody`, that contains the code for instrumenting the ComplexEventProcessor that actually is Drools. To provide a more abstract generation of a rule for monitoring Non-Functional properties, we developed a MetaModel [13], for defining Non-Functional properties [**?**], from which, users may generate their own rule model and, using ATL or several transformation language like Acceleo [43] the generated model may be translated directly to the desidered/more performant/available Complex Event Processor language. (iii) `RuleType` that contains the name of the language on which the rule is expressed.

Each *ComplexEventResponse message* (see 6.2) sent by the Monitoring Enabler contains a `RuleName` field, which uniquely determines the matched rule. The field `NetworkedSystemSource` provides information about the networked system that generated the captured event. The field `ResponseKey` provides information about the method invoked on the NS. The field `ResponseValue` provides information about the response of the networked system to that method invocation.

6.3 Integration of GLIMPSE monitoring and the learning enabler

The overall structure of the monitoring-enabled learning setup is shown in Fig. 6.4. On the left hand side an instrumented CONNECTor observes the interaction between a client and the target system. These

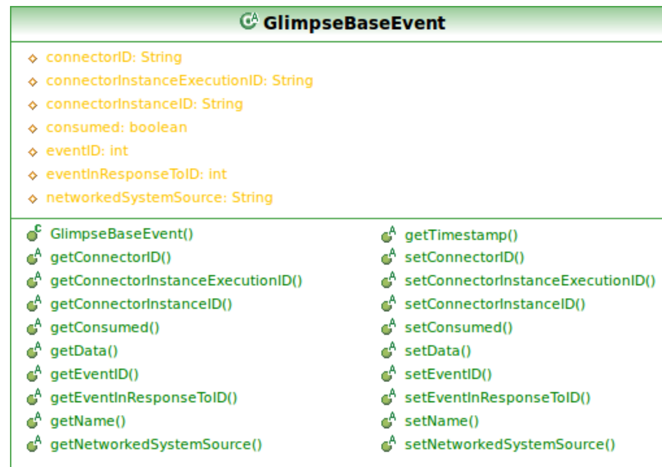

Figure 6.2: The GlimpseBaseEvent Interface

Listing 6.2: The Complex Event Response XSD

```
<?xml version=”1.0” encoding=”UTF−8”?>
<schema xmlns=”http://www.w3.org/2001/XMLSchema”
  xmlns:tns=”http :// www.example.org/ComplexEventResponse/”
  targetNamespace=”http://www.example.org/ComplexEventResponse/”
  attributeFormDefault=”qualified”>
  <element name=”ComplexEventResponseList” type=”tns:ComplexEventResponse” />
   <complexType name=”ComplexEventResponse”>
      <sequence>
       <element name=”RuleName” type=”string” maxOccurs=”1” minOccurs=”1” />
       <element name=”NetworkedSystemSource” type=”String” maxOccurs=”1” minOccurs=”1” />
       <element name=”ResponseKey” type=”string” maxOccurs=”1” minOccurs=”1” />
       <element name=”ResponseValue” type=”string” maxOccurs=”1” minOccurs=”1” />
      </sequence>
   </complexType>
</schema>
```

observations are delivered to the monitoring bus. On the right hand side a learning setup will retrieve observations from the monitoring bus.

It is at first unknown in what corresponding state in the hypothesis the target system is. Thus the so called “state tracker”, which relates incoming observations with the states in a given hypothesis, at first assumes that the target system is in any of the states of the hypothesis. With every observation the state tracker tries to find a fitting transition with matching output for the currently considered set of states. The next set of considered states consists of the successors of the previously considered states for which a transition with fitting output does exist. Usually the set of considered states will quickly converge to one state, which means that the state tracker has “a state lock”, i.e., the state tracker found a state in the hypothesis most likely relating to the current state of the target system.

If the state tracker loses the “state lock”, i.e., the set of considered state is empty, the trace of incoming observations since state lock and the last related state is sent as evidence to the evidence evaluator, which constructs a query comprised of an access sequence to the last known state and the delivered trace of observations. This query is executed on the target system and on the current hypothesis, upon which the execution results are compared. If the output does not match the constructed query is indeed a counterexample and thus delivered to the learning algorithm to construct a refined hypothesis.

6.4 Conclusions and perspectives

In this chapter we described the integration between two CONNECT enablers: Monitoring (GLIMPSE) and Learning (LearnLib) that are involved into one of the foremost objectives of the CONNECT architecture:

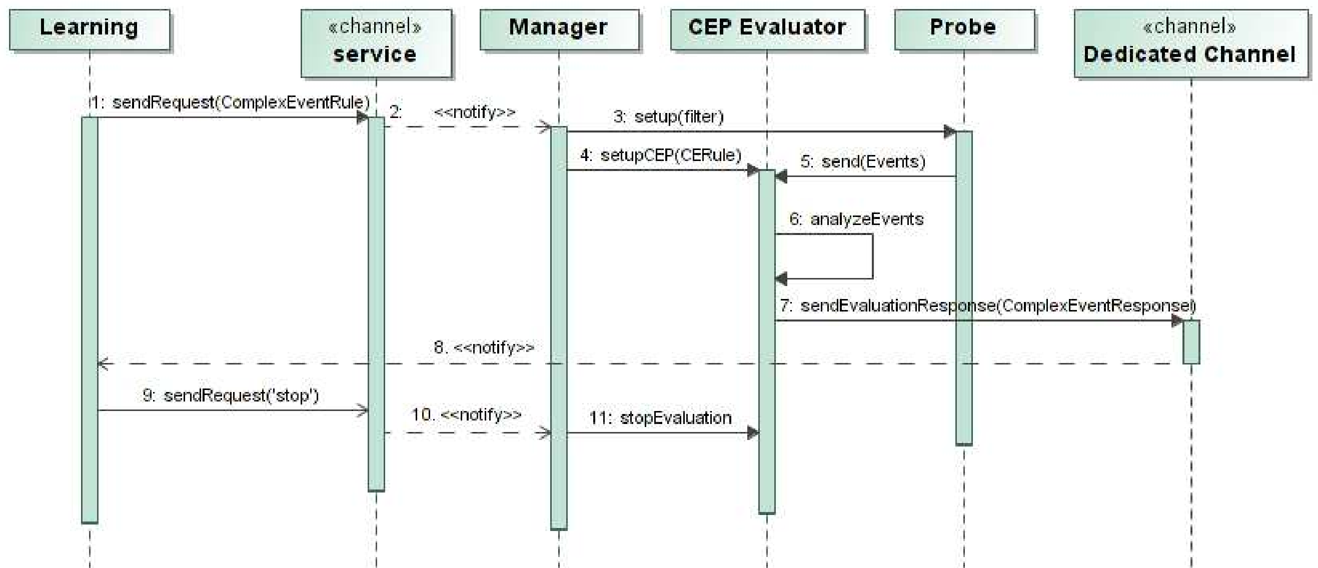

Figure 6.3: The interaction between Learning and Monitoring

Adaptation.

The simple yet powerful idea that we implemented is that after a CONNECTor is deployed, the Learned models for the Networked Systems undergo continuous validation, by keeping relevant events under observation through the publish-subscribe monitoring mechanism. If some behaviour is detected that is in contrast with the learned models, GLIMPSE notify the Learning enabler, which in turn will restart active learning tests to possibly construct a refined hypothesis and update the models.

The learned model can now be enriched using realtime monitored data and then given in input to the Synthesis Enabler that in charge to manage emergent behaviors of the CONNECTed system.

In Y4 we aim to combine into the CONNECT architecture Monitoring Enabler, Learning Enabler and Dependability Enabler with Synthesis Enabler in order to provide an eternal interoperability among heterogeneous and evolving networked systems.

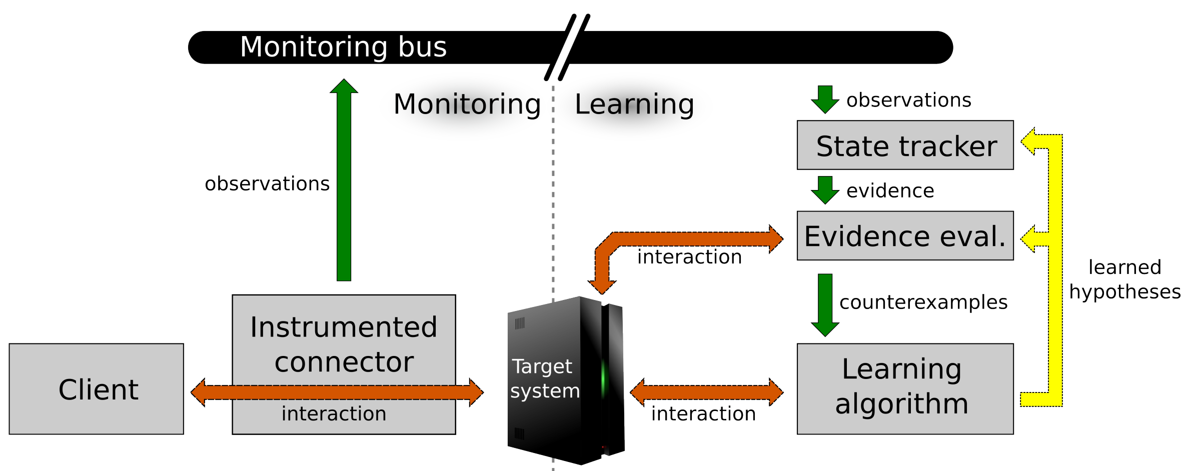

Figure 6.4: Integration of monitoring and learning

7 Conclusion and Future Work

This document has detailed the work done by WP4 during Y3 of CONNECT. We have extended the state-of-the-art on learning techniques, so that they are able to handle richer models including data of various forms occurring in systems within the scope of CONNECT, such as communication protocols and networked services. These extensions build on the work initiated during Y2, and have now been implemented in LearnLib.

During Y3, we have also demonstrated how information available in interface descriptions can be exploited to create learning setups, and also make behavioral learning more efficient. We described how type analysis can infer dataflow properties of a system, which is then exploited by the learning algorithm.

We have also developed an integration of the GLIMPSE monitoring infrastructure with LearnLib, using a “Never-stop Learning” approach. In this approach, GLIMPSE we can observe during execution whether or not the inferred model really suffices as an explanation for the real system’s behavior. If not, information collected by GLIMPSE is subsequently analyzed by LearnLib, leading to a refined model.

Finally, in Y3, we have realized the Learning Enabler, and integrated it as a part of the overall CONNECT architecture.

Future work

During Y4, we will focus on developing our past lines of work towards further maturation and integration within the CONNECT architecture. The work on learning of richer automata models will be further developed by developing and implementing learning techniques, extending the work of Section 3 to that it is parameterized on a data domain, on which an arbitrary set of relations and operations is provided. The extension will be evaluated on realistic communication procotols and services.

We will also extend learning techniques to cover non-functional properties. Our plan is to work along two directions. In one, we will add nonfunctional properties to a learned model; this is in some sense a continuation of the work reported in Chapter 8 of D4.2. In the other, we will also capture the influence of nonfunctional properties on functional ones, using techniques inspired by those developed in Chapter 2 of this deliverable.

We will further integrate the Learning Enabler with the real Interaction Enabler as well as with the JMS infrastructure to enable remote communication between the CONNECT enablers. As these modifications only concern changes to the implementation of a few classes, but not to their interfaces, we do not expect this task to require a significant amount of time. Moreover, the Learning Enabler needs to be evaluated in the context of the scenario developed by WP6.

Bibliography

[1] Drools Fusion: Complex Event Processor. http://www.jboss.org/drools/drools-fusion.html.

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

[3] RuleML: The Rule Markup Initiative. http://ruleml.org.

[4] ServiceMix: an open source ESB. http://servicemix.apache.org/home.html.

[5] Extensible messaging and presence protocol. http://xmpp.org/rfcs/, 2011. version of 07.02.2011.

[6] F. Aarts, B. Jonsson, and J. Uijen. Generating models of infinite-state communication protocols using regular inference with abstraction. In A. Petrenko, A. da Silva Simão, and J. C. Maldonado, editors, *ICTSS*, volume 6435 of *Lecture Notes in Computer Science*, pages 188–204. Springer, 2010.

[7] F. Aarts, J. Schmaltz, and F. W. Vaandrager. Inference and Abstraction of the Biometric Passport. In Margaria and Steffen [51], pages 673–686.

[8] F. Aarts and F. Vaandrager. Learning I/O automata. In P. Gastin and F. Laroussinie, editors, *CONCUR 2010 - Concurrency Theory, 21th International Conference*, volume 6269 of *Lecture Notes in Computer Science*, pages 71–85. Springer, 2010.

[9] R. Alur, P. Cerný, P. Madhusudan, and W. Nam. Synthesis of interface specifications for Java classes. In J. Palsberg and M. Abadi, editors, *POPL*, pages 98–109. ACM, 2005.

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

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

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

[13] Antonia Bertolino and Antonello Calabrò and Francesca Lonetti and Antinisca Di Marco and Antonino Sabetta. Towards a model-driven infrastructure for runtime monitoring, 2011.

[14] C. Bartolini, A. Bertolino, E. Marchetti, and A. Polini. WS-TAXI: a WSDL-based testing tool for Web Services. In *ICST 2009, Denver, Colorado - USA*. IEEE, 2009.

[15] 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*, 2010.

[16] 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, *FASE*, volume 4961 of *Lecture Notes in Computer Science*, pages 317–331. Springer, 2008.

[17] A. Bertolino, P. Inverardi, P. Pelliccione, and M. Tivoli. Automatic synthesis of behavior protocols for composable web-services. In H. van Vliet and V. Issarny, editors, *ESEC/SIGSOFT FSE*, pages 141–150. ACM, 2009.

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

[19] T. Bohlin, B. Jonsson, and S. Soleimanifard. Inferring compact models of communication protocol entities. In Margaria and Steffen [51], pages 658–672.

[20] M. Bojanczyk, C. David, A. Muscholl, T. Schwentick, and L. Segoufin. Two-variable logic on data words, 2011. ACM Transactions on Computational Logic, to appear.

[21] M. Bojanczyk, B. Klin, and S. Lasota. Automata with group actions. In *LICS*, pages 355–364. IEEE Computer Society, 2011.

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

[23] S. Cassel, F. Howar, B. Jonsson, M. Merten, and B. Steffen. A succinct canonical register automaton model. In *ATVA*, volume 6996 of *Lecture Notes in Computer Science*, pages 366–380. Springer Verlag, 2011.

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

[25] 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.

[26] CONNECT Consortium. Deliverable 4.2 – Further development of learning techniques, 2011.

[27] CONNECT Consortium. Deliverable 5.2 – Design of approaches for dependability and initial prototypes, 2011.

[28] V. Dallmeier, N. Knopp, C. Mallon, S. Hack, and A. Zeller. Generating test cases for specification mining. In *Proceedings of the 19th international symposium on Software testing and analysis*, ISSTA '10, pages 85–96, New York, NY, USA, 2010. ACM.

[29] 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.

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

[31] C. Ghezzi, A. Mocci, and M. Monga. Synthesizing Intentional Behavior Models by Graph Transformation. In *ICSE 2009, Vancouver, Canada*, 2009.

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

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

[34] O. Grumberg, O. Kupferman, and S. Sheinvald. Variable automata over infinite alphabets. In A. Dediu, H. Fernau, and C. Martín-Vide, editors, *LATA*, volume 6031 of *Lecture Notes in Computer Science*, pages 561–572. Springer, 2010.

[35] 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,* 5th *Int. Conf. on Fundamental Approaches to Software Engineering*, volume 2306 of *Lecture Notes in Computer Science*, pages 80–95. Springer Verlag, 2002.

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

[37] F. Howar, B. Steffen, B. Jonsson, and S. Cassel. Inferring Canonical Register Automata. In *VMCAI 2012*, to appear.

[38] 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.

[39] 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.

[40] H. Hungar, T. Margaria, and B. Steffen. Test-based model generation for legacy systems. In *Test Conference, 2003. Proceedings. ITC 2003. International*, volume 1, pages 971–980, 30-Oct. 2, 2003.

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

[42] X. Jinyu, X. Xiaorong, W. Jingtao, and H. Yingduo. Framework of synchronous dynamic security monitoring system based on pmu. In *Power Tech Conference Proceedings, 2003 IEEE Bologna*, volume 2, page 5 pp. Vol.2, june 2003.

[43] Jonathan MUSSET and Étienne JULIOT and Stéphane LACRAMPE. ACCELEO Reference Guide, .

[44] B. Jonsson. Learning of Automata Models Extended with Data. In M. Bernardo and V. Issarny, editors, *SFM*, volume 6659 of *Lecture Notes in Computer Science*, pages 327–349. Springer, 2011.

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

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

[47] R. Lazic and D. Nowak. A unifying approach to data-independence. In C. Palamidessi, editor, *Proc. CONCUR 2000, 11th Int. Conf. on Concurrency Theory*, volume 1877 of *Lecture Notes in Computer Science*, pages 581–595, 2000.

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

[49] D. Lorenzoli, L. Mariani, and M. Pezzè. Automatic Generation of Software Behavioral Models. In *ICSE 2008*, pages 501–510, NY, USA, 2008. ACM.

[50] O. Maler and A. Pnueli. On recognizable timed languages. In *Proc. FOSSACS04, Conf. on Foundations of Software Science and Computation Structures*, Lecture Notes in Computer Science. Springer Verlag, 2004.

[51] 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.

[52] L. Mariani and M. Pezzè. Dynamic Detection of COTS Component Incompatibility. *IEEE Software*, 24(5):76–85, 2007.

[53] M. Merten, B. Steffen, F. Howar, and T. Margaria. Next Generation LearnLib. In *TACAS 2011*, volume 6605 of *Lecture Notes in Computer Science*, pages 220–223. Springer Verlag, 2011.

[54] M. Merten, B. Steffen, F. Howar, and T. Margaria. Next Generation LearnLib. In P. A. Abdulla and K. R. M. Leino, editors, *TACAS*, volume 6605 of *Lecture Notes in Computer Science*, pages 220–223. Springer, 2011.

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

[56] Oracle.com. JAX-WS RI 2.1.1 – wsimport. `http://download.oracle.com/javase/6/docs/technotes/tools/share/wsimport.html`, 2011. [Online; accessed 13-September-2011].

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

[58] C. Pautasso, O. Zimmermann, and F. Leymann. Restful web services vs. big web services: Making the right architectural decision. In *17th International World Wide Web Conference (WWW2008)*, pages 805–814, Beijing, China, April 2008 2008.

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

[60] M. Pradel and T. Gross. Automatic generation of object usage specifications from large method traces. In *Automated Software Engineering, 2009. ASE '09. 24th IEEE/ACM International Conference on*, pages 371 –382, nov. 2009.

[61] H. Raffelt, T. Margaria, B. Steffen, and M. Merten. Hybrid test of web applications with webtest. In *TAV-WEB '08: Proceedings of the 2008 workshop on Testing, analysis, and verification of web services and applications*, pages 1–7, New York, NY, USA, 2008. ACM.

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

[63] R. Rivest and R. Schapire. Inference of finite automata using homing sequences. *Information and Computation*, 103(2):299–347, 1993.

[64] 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.

[65] 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.

[66] M. Shahbaz, K. Li, and R. Groz. Learning Parameterized State Machine Model for Integration Testing. In *Proc. COMPSAC 2007, 31st Annual International Computer Software and Applications Conference, Beijing, China*, volume 2, pages 755–760, Washington, DC, USA, 2007. IEEE Computer Society.

[67] M. Shahbaz, K. C. Shashidhar, and R. Eschbach. Iterative refinement of specification for component based embedded systems. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, pages 276–286, New York, NY, USA, 2011. ACM.

[68] 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.

[69] B. Steffen, F. Howar, and M. Merten. Introduction to active automata learning from a practical perspective. In *SFM*, volume 6659 of *Lecture Notes in Computer Science*, pages 256–296. Springer, 2011.

[70] J. Tretmans. Model-based testing and some steps towards test-based modelling. In *SFM*, volume 6659 of *Lecture Notes in Computer Science*, pages 297–326. Springer Verlag, 2011.

[71] A. Wasylkowski and A. Zeller. Mining Operational Preconditions. http://www.st.cs.uni-saarland.de/models/
papers/wasylkowski-2008-preconditions.pdf (Tech. Rep.).

[72] A. Wasylkowski, A. Zeller, and C. Lindig. Detecting Object Usage Anomalies. In *ESEC-FSE '07, pp. 35-44*. ACM, 2007.

[73] T. Wilke. Specifying timed state sequences in powerful decidable logics and timed automata. In H. Langmaack, W. P. de Roever, and J. Vytopil, editors, *FTRTFT*, volume 863 of *Lecture Notes in Computer Science*, pages 694–715. Springer, 1994.