Integration into the CONNECT Architecture

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

▶ **To cite this version:**

Antonia Bertolino, Antonello Calabro, Sofia Cassel, Yu-Fang Chen, Falk Howar, et al.. Integration into the CONNECT Architecture. [Research Report] 2012. hal-00805623

HAL Id: hal-00805623
https://inria.hal.science/hal-00805623

Submitted on 28 Mar 2013

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.

CONNECT

Emergent Connectors for

Eternal Software Intensive Networked Systems

ICT FET IP Project

Deliverable D4.4

Integration into the CONNECT Architecture

SEVENTH FRAMEWORK PROGRAMME

http://www.connect-forever.eu

Inria · DOCOMO Euro-Labs · Ambientic · Lancaster University · Thales · Università dell'Aquila · technische universität dortmund · University of Oxford · Uppsala Universitet

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

Deliverable Number	:	D4.4
Title of Deliverable	:	Integration into the CONNECT Architecture
Nature of Deliverable	:	R
Dissemination Level	:	Public
Internal Version Number	:	1.2
Contractual Delivery Date	:	1 Dec. 2012
Actual Delivery Date	:	13 Dec. 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), Bernhard Steffen (TUDo)
Reviewer(s)	:	Massimo Tivoli (UDA)

Abstract

The CONNECT *Integrated Project aims at enabling continuous composition of networked systems, using a revolutionary approach, based based on on-the-fly synthesis of* CONNECT*ors. The Role of Work Package 4 is to develop techniques for learning representative models of the connector-related behavior of networked peers and middleware through exploratory interaction, and for monitoring the runtime behaviour of the connected system.*

This document provides an overview of WP4 achievements during the final year of CONNECT*, as well as a summary of WP4 achievements and remaining challenges for the entire period of* CONNECT*.*

During Y4, WP4 has further increased the power and efficiency of learning techniques, developed and implemented techniques for handling non-functional properties in learning, and finalized the integration of the learning and monitoring enablers into the CONNECT *architecture.*

Over the 46 months of CONNECT *operation, WP4 has significantly advanced the state-of-the-art in active automata learning. Prior to the* CONNECT *project, active learning had been developed only for finite-state component models, utilizing a finite set of interaction primitives. In* CONNECT*, we have lifted this technology to rich and infinite-state techniques by novel symbolic and abstraction-based techniques, thereby providing a break-through in the state-of-the-art, which will have long lasting impact also after* CONNECT*. During* CONNECT*, we have also thoroughly re-engineered our framework for learning, LearnLib, making learning functionality available as reusable components. Further, we have developed a generic monitoring infrastructure that offers great flexibility and adaptability, which is model-driven: it can thus be adapted to a rich set of domain-specific languages, expressed as metamodels, and exploit the support to automation offered by model-driven engineering techniques. Our assessment of the learning and monitoring enablers on* CONNECT *scenarios shows that the developed technology can cope very well and very efficiently with the challenges imposed by the* CONNECT *approach.*

Document History

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

Document Review

Date	Version	Reviewer	Comment
Nov. 29, 2012 Dec. 7, 2012	1.0 1.1	Massimo Tivoli Valérie Issarny	approving significant restructurings

Table of Contents

List of Figures

1 Introduction

The CONNECT Integrated Project aims at enabling continuous composition of evolving networked systems. It adopts a revolutionary approach to the seamless networking of digital systems, based on on-the-fly synthesis of CONNECTors via which networked systems communicate. CONNECTors are implemented through a comprehensive dynamic process based on (i) extracting knowledge from, (ii) learning about, and (iii) reasoning about, the interaction behavior of networked systems, together with (iv) synthesizing new interaction behaviors out of the ones exhibited by the systems to be made interoperable, and further (v) generating and deploying corresponding connector implementations.

The Role of Work Package 4 is to develop techniques for realizing step (ii) in this process, i.e., for learning representative models of the connector-related behavior of networked peers and middleware through exploratory interaction. Work Package 4 also develops techniques for monitoring the runtime behaviour of the connected system, since this is a prerequisite for enabling continuous composition.

The objectives, as stated in the Description of Work (DoW)[1], 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.'*

Work Package 4 is structured into three tasks.

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

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

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

The work in WP4 is based on existing techniques for learning the temporal ordering between a finite set of interaction primitives. Such techniques have been developed for the problem of regular inference (i.e., automata learning), in which a regular set, represented as a finite automaton, is to be constructed from a set of observations of accepted and unaccepted strings. The most efficient such techniques use the setup of *active* learning, where the automaton is learned by actively posing two kinds of queries: a *membership query* asks whether a string is in the regular set, and an *equivalence query* compares a hypothesis automaton with the target regular set for equivalence, in order to determine whether the learning procedure was (already) successfully completed. The typical behavior of a learning algorithm is to start by asking a sequence of membership queries, and gradually build a hypothesized automaton using the obtained answers. When a "stable" hypothesis has been constructed, an equivalence query finds out whether it is equivalent to the target language. If the query is successful, learning has succeeded; otherwise it is resumed by more membership queries until converging at a new hypothesis, etc.

[1] CONNECT Grant Agreement, Annex I

Overview of this Deliverable In the next chapter, we report on the work that was performed during Y4. In Chapter 3, we summarize the overall contributions of WP4 during CONNECT, highlighting the challenges that have been solved and the challenges that remain. An assessment of the achievements of WP4 is performed in Chapter 4. Conclusions are collected in Chapter 5.

Two technical journal papers are appended to this deliverable. They represent central achievements of WP4 that have been obtained through collaborative efforts.

- S. Cassel, F. Howar, B. Jonsson, M. Merten, and B. Steffen. A succinct canonical register automaton model. Invited for publication in *Journal of Logic and Algebraic Programming*, 2012.

  This paper is an extended and more elaborate version of the first paper on this topic [6].

- F. Howar, B. Steffen, B. Jonsson, S. Cassel, and M. Isberner. Inferring canonical register automata. Submitted for publication, 2012.

  This paper is an extended description of our approach for learning automata with data, which supersedes both the papers [13] and [12].

2 Challenges and Achievements During Y4

2.1 Review Recommendations

We here list comments and recommendations from the Y3 review, together with our responses.

> *Overall, this is excellent progress.*

WP4 is happy about this feedback.

> *Thus, the remaining challenge is efficient handling of non-functional properties.*

The WP4 framework is prepared for dealing with non-functional properties. More specifically, the LearnLib framework can be (and has been) instrumented to monitor non-functional properties. For instance, learned models can be annotated with latencies to form (a simple form of) timed automata. Other possibilities include failure frequences. We elaborate on this topic in Section 2.5.

> *How do the methods for monitoring of non-functional properties developed in WP2 interfere with learning techniques developed in WP4?*

The above comment probably contains a typo: we read it as "How do the methods for monitoring of non-functional properties interfere with learning techniques developed?", in fact monitoring is not developed in WP2 but in WP4 (as also learning is). If in the above question the term *interfere* has to be taken literally under the negative meaning of acting as an obstacle, we do not think there is any interference between the two enablers. Learning happens before deployment of the CONNECTor and monitoring happens during the execution of the CONNECTed system. On the contrary, the two activities can usefully cooperate, following the process described in [4]. Indeed, the monitor can report to the Learning Enabler, similarly to how it does with any other enabler, if some relevant property happens, and learning could in this way be informed that some learned model should be revised. To have this cooperation, the Learning Enabler needs to subscribe to GLIMPSE for the desired properties, again similarly to what is done by any other Enabler.

2.2 Challenges During Y4

During Y4, the challenges for WP4 have been to:

- further refine the work on automata models and learning algorithms,
- make the learning algorithms more efficient,
- extend learning techniques to handle non-functional properties,
- finalize the integration of the learning enabler into the CONNECT architecture,
- finalize the integration of the monitoring enabler into the CONNECT architecture, and
- (in connection with the two previous items) integrate learning and monitoring.

These challenges include those planned for D4.4 in the Description of Work, from which we quote

- *Final integration of the learning and monitoring functionality into the* CONNECT *architecture*
- *Overall Evaluation*

In this chapter, we elaborate on the achievements with respect to these challenges.

2.3 Extension of Learning to Richer Languages

Prior to the CONNECT project, state of the art learning algorithms had been developed for finite-state models of components, that utilized a finite set of primitives for interaction. To support learning of realistic networked components, one challenge of CONNECT has been to extend learning techniques to handle richer models, that include data of various form. In CONNECT, we have addressed this challenge by developing a theory of canonical *register automata*. This theory has served as the basis for development of new classes of learning algorithms that can generate rich behavioral models. During Y2 and Y3, we developed this line of work for a setting where data values were taken from an unbounded domain, and could be compared for equality. During Y3, we initiated a generalization of this work to settings where a range of different predicates can be applied to data values. As an illustrating example, 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 essential test between a fourth value d_4 and these three is whether $d_4 \leq d_1$ or not (i.e., all other comparisons not essential for determining 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 essential or not, resulting in only 2 cases.

During Y4, the foundations for and presentation of the original register automaton model, for the setting where data values can be compared for equality, has been stabilised. A paper has been invited to a journal [7], and is also enclosed as an appendix. During Y4, we have also fully worked out the generalization of this work to the setting where a range of predicates can be applied on data values, and published the resulting proposal at an international conference [8].

2.4 Efficient Learning Algorithms

In Y3, on the basis of our canonical automaton model, we have developed a learning algorithm for behaviors that combine control and data. This learning algorithm can be derived systematically, by exploiting the fact that the automaton model is based on a Nerode congruence. 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 [28] for learning regular languages. We have transferred it to Mealy machine learning in [31], and to learning with alphabet abstraction refinement in [15]. A first presentation of this learning algorithm appeared in [13].

In Y4, we have substantially extended both the scope and the efficiency of the learning algorithm presented in [13], resulting in a better algorithm presented in [12].

- First, the new algorithm applies to a richer component model that considers both reception of and production of messages by a component. The algorithm can infer any equality between a received or produced data value and any data value in any future message, and how such an equality may affect subsequent behavior. The resulting model is called Input/Output Register Automata (IORA for short).

- Second, the new realization of the algorithm is significantly more efficient than the previous one. We have applied the new implementation to a range of examples. 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. Thus, this work represents a breakthrough in the area of automata learning, and outperforms previous approaches to learning models that combine control and data.

An extended presentation of the learning algorithm, combining the contributions of [13] and [12], is provided as an appendix.

2.5 Learning Non-functional Properties

In the CONNECT framework, alongside with inferring functional models, the learning enabler is envisioned to gather information about non-functional properties to be used for connector synthesis and subsequent validation of dependability properties. Partly inspired by the reviewers' comments about adopting a lightweight approach to this problem, we have developed, implemented, and evaluated an extension to our learning technology [27].

In this section we show how active learning and obtained models can be extended to also provide non-functional information about inferred systems. We present two strategies, one for recording non-functional data in the course of active learning, and one for augmenting inferred models with data obtained from a system in actual use by means of live logfile analysis.

2.5.1 Generating QoS Data Points During Learning

Let us briefly recapitulate the experimental setup of a learning process: The learning algorithm will formulate queries to the system on an abstract level. The queries will be passed to a test-driver that translates queries into sequences of single actions on the component that is subject to learning, here also called the system under test (SUT). The test-driver executes these actions one by one and records the respective results. We can extend the test-driver easily to collect some data during the active interrogation of the target system, which is part of the employed active learning methods. The data collected this way, however, might be strongly biased for three reasons:

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

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

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

Issue (3) has to be addressed by different means. To produce a complete picture of the non-functional properties of a system, exposure to realistic load is needed. The only way to do this is to conduct tests in parallel. It is easy to see how the setup used for equivalence approximation can be parallelized: one can simply select multiple transitions at a time and conduct multiple tests per transition in parallel.

In order to produce realistic load in the learning phase, the learning algorithms and equivalence tests in LearnLib can be modified to produce batches of queries instead of single queries. Being constructed during learning, the size of these batches usually varies along the learning process. The resulting setup is shown in Figure 2.1. The learning algorithm produces batches of queries and passes these to an oracle, which distributes the queries to a number of test-drivers. The latter consist of (1) a *TestOracle* serving as interface to the learning algorithm, (2) a *TestDriver* that translates abstract symbols to executable actions on the system, and finally (3) the *SUTInstrumentation* that actually performs the actions on the SUT. We can easily instrument this setting by adding probes at different points that record data points useful for quality of service measurement, especially of non-functional aspects. The *TestDriver* logs data about the execution of single actions (e.g., latencies), while the *TestOracle* logs data concerning single symbols and complete queries. The *TestOracle* can, e.g., record failure rates for symbols. This is not possible in the

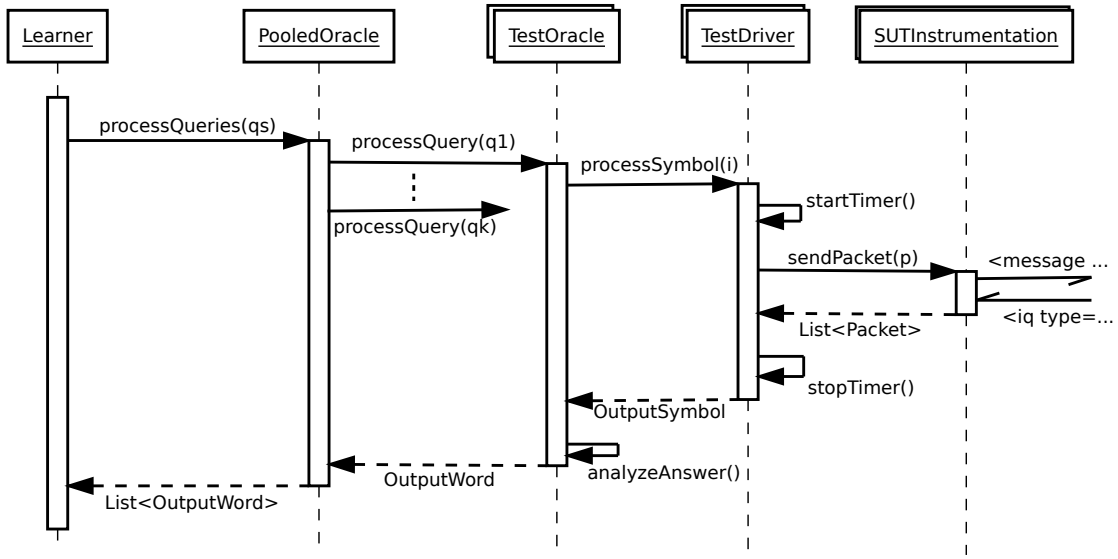


Figure 2.1: Sequence chart probe integration

TestDriver as it requires an interpretation of the obtained (or omitted) answers. This architecture, i.e., the combination of probes in the test process and parallel execution of tests in a two-step approach, allows us to provide data for a broad range of non-functional properties relevant in the scope of CONNECT.

2.5.2 Recording Latencies

We have applied our enhanced learning framework to measure non-functional properties in learning experiments on the OCS. This experiment was conducted as part of a case study on the use of learning technology employed during development, validation, operation and use of the Online Conference Service (OCS), Springer Verlag's online manuscript submission and review system [17].

The OCS is an online manuscript submission and review service that is part of a product line for the Springer Verlag that started in 1999 [26]. It evolved over time to include also journal and volume production preparation services. The main purpose of the OCS is the adequate handling of a wealth of independent but often indirectly related user interactions. Hence, completing a task may have an impact on other participants or shared objects. From this point of view the OCS is a user and context driven reactive system with a web interface. Users can decide when they execute their tasks, which typically consist of small workflows, and in case of multiple tasks the order in which they process or perhaps reject them.

We first addressed latencies by measuring execution times of actions. Initially we observed and learned them at the aggregation level of actions, as one would do in a customary testing setting: one observes the executions and aggregates the data by action across all the executions (i.e. not at the level of single transitions: this is a model agnostic approach). We tracked the variation of execution time with increasing load, i.e., the execution times for creating new conferences in an increasingly populated system. All conferences run in parallel on the same server. The obtained results shown in Figure 2.2 proved very useful for profiling purposes. On the left we see the diagram of the measures for the initial implementation: a linear increase of execution times from two seconds on an empty systems to almost 8 seconds on a system with 700 conferences.

The discovered linear growth of execution times led us to carefully inspect the involved business methods and objects and to introduce refinements of two kinds:

1. optimizations of the business logic to reduce the number of statements to be executed on the database, and

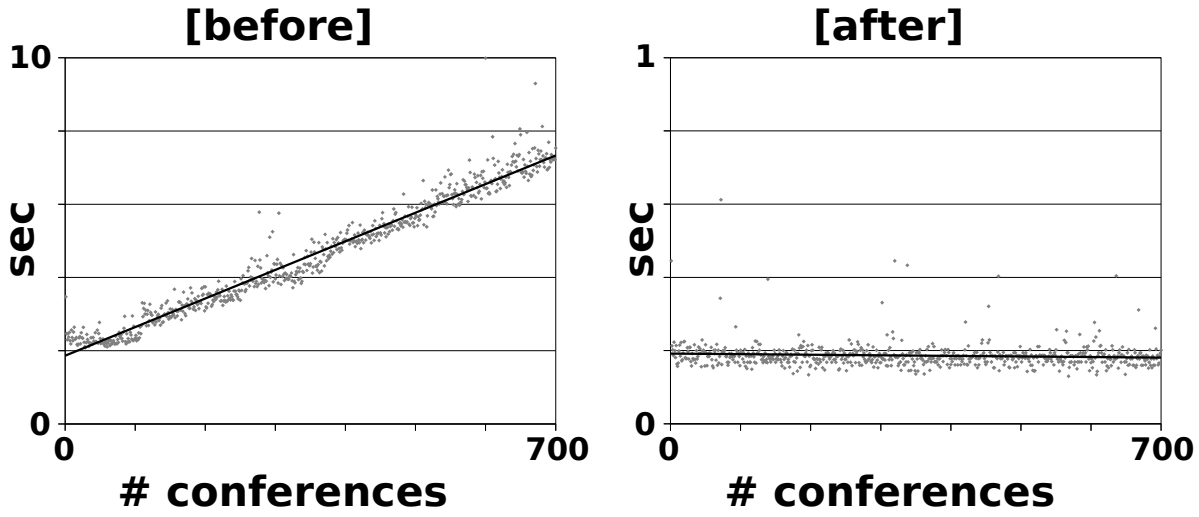


Figure 2.2: Measured latencies for creating conferences

2. effective refinements concerning the O/R-mapping[1] of the business model. In particular the (lazy-)loading strategies for relations were refined.

The right graph in Figure 2.2 shows the execution times once these optimizations were implemented. Now the execution times stay constant even in a densely populated system; the slight decrease in the measured data is not significant and can be attributed to caching facilities in the enterprise architecture.

Since we did not observe the timing behavior at the level of single transitions in the model but rather per business method, the results from profiling indicated only a set of promising methods, worth being considered and possibly refined/optimized. Without (behavioral) context information, considerable effort was required in order to determine effective refinements. We therefore conjecture that timing details combined with behavioral models will further ease the effort of locating needed refinements more precisely in the source code. Also, to overcome the problem of unrealistic data in the learning environment, we have decided to enrich learned behavioral models with information on non-functional properties obtained through monitoring of a system in actual use. We outline this in the next section, again by means of the OCS system.

2.5.3 Capturing Actual Usage

The OCS implements a form of trace logging in that it intercepts and logs every state-changing call of the business logic, which can be facilitated to reconstruct the user's behavior. We used this technique already a long time ago to discover specific profiles of use of the system across different user communities [18]. Back then the underlying question was the decision of whether a certain consolidation of the infrastructure was feasible, economically advantageous, and beneficial to the users. The logging was correspondingly abstract.

Detailed log information is useful to track the history of an object – like a submission – or to trace possible errors in the system. This tracking goes well beyond the logging level we had in previous versions, and it is important for transparency and liability reasons, to solve potential disputes on claims like e.g. that a paper was submitted and has been removed later from the conference service while the PC Chairs insist that they did not delete it. An evaluation of all data entries for the paper in question will provide evidence on the actions that really took place.

Additionally, all entries are secured against fraud: a combination of timestamp, counter, and signature

[1] An O/R-mapper introduces a layer of indirection for connecting objects in an object oriented language to tables in a relational database.

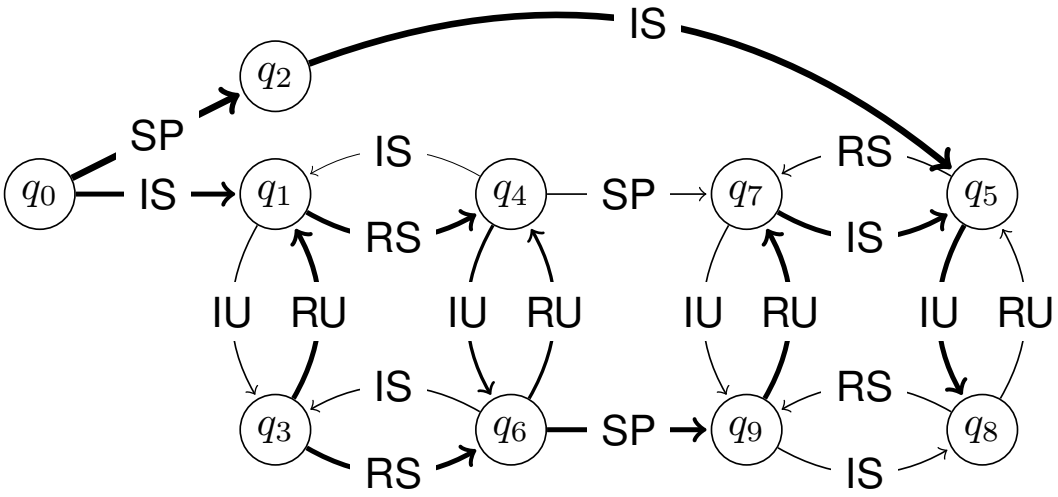


Figure 2.3: System model enriched with information from log data

for each entry makes it hard even for the system administrators to modify or delete entries of the log file undetected.

As soon as enough log data has been gathered, it can be used not only to observe the actions of single users but also to get a picture of the overall usage in the system. In combination with the model generated by active automata learning, we can also project the data onto the model and obtain an enriched model that contains not only all possible behaviors, but also for example the usage frequency of each state and of each path. In practice, during this model enrichment all instance-specific parameters in the log file are anonymized so that all papers and all users become indistinguishable, and their histories are aggregated into one workflow. Figure 2.3 shows an example of such a model that is enriched with frequency information: the thicker the line of a transition edge, the more often it has been visited.

2.5.4 Conclusions

By means of a case study, we have illustrated the power of active learning for enabling the automated quality assurance of complex and distributed evolving systems. In particular, it shows the impact of coninuity in the learning and monitoring activities, in several ways.

- Frequently building the system and applying our learning-based testing technology continuously immediately revealed misconceptions, and the incrementally improved learned models better and better reflected the overall system behavior. Applied carefully, learning nicely adapts to changes and is therefore an ideal means for dealing with evolving systems.

- By careful adaptation, learning technology can infer a number of quality metrics, including various non-functional properties, that enrich the generated models, and also guide subsequent development of the system under learning.

We are therefore convinced that this technology will have quite a practical impact in the future of quality assurance for evolving systems.

2.6 Integrating the Learning Enabler

While the applications in the CONNECT project have been the driving force for many of the developments reported on in the previous sections, they have resulted in stand-alone tools or libraries, such as LearnLib. This yields a high amount of flexibility and vastly broadens the target audience. On the other hand, it creates the necessity of adapting those very general techniques to the specifics of the CONNECT scenarios, such as data structures, communication protocols etc. The *Learning Enabler* is the component in the CONNECT architecture intended to accomplish this task, providing a smooth interoperability with the various other CONNECT enablers.

A prototype of the Learning Enabler was presented in D4.3. In Y4, its integration into the overall CONNECT architecture has been finished. Furthermore, its functionality was enhanced, e.g. by integrating the learning of non-functional properties as presented in Section 2.5.

2.6.1 Finalized Integration into CONNECT Architecture

In the CONNECT architecture, the Learning Enabler interacts with the Discovery Enabler and the Interaction Enabler (for an overview of the architecture, see D1.3 and D1.4). In Y3, this integration was established at an interface level, using stub implementations for testing purposes. Particularly, the stub Interaction Enabler relied on the tool `wsimport`, limiting its application to networked systems communicating via the SOAP protocol.

In Y4, the integration with both the actual Discovery and Interaction Enabler has been established. The integration with the Interaction Enabler leverages the applicability of automata learning in CONNECT from SOAP-based services only to the full bandwidth of middleware protocols supported in CONNECT, as now a single component is used for both the networked system invocations during learning and during CONNECTor execution.

As the CONNECT Enabler architecture is envisioned to be deployed in a distributed environment, the functionality of the Learning Enabler is made accessible via JMS using a corresponding wrapper (cf. D1.3). Consequently, communication with the other Enablers is realized via JMS. The complete setup has been successfully tested and run in a distributed networked environment.

2.6.2 Enhancements of the Learning Enabler

While the aforementioned aspects concern the interoperation with external components only, also the internal functionality was enhanced. A data-flow diagram depicting the internal operations of the Learning Enabler in the CONNECTion phase is shown in Figure 2.4. This diagram is a refined version of the corresponding diagram shown in Deliverable D4.3. We will therefore concentrate on the differing aspects.

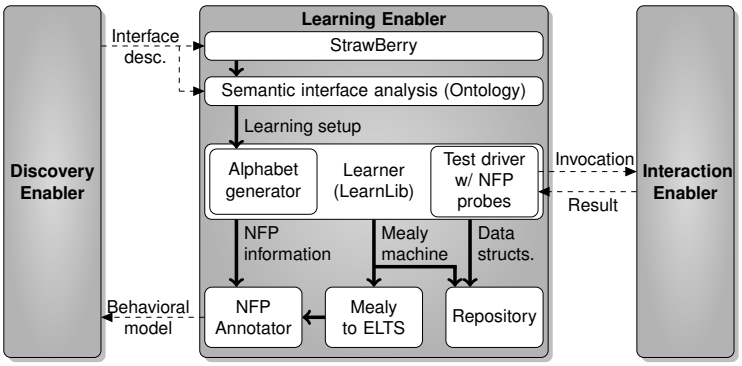


Figure 2.4: Refined data-flow of the Learning Enabler in the CONNECTion phase.

When the Discovery Enabler requests a networked system's behavior to be learned, it provides the Learning Enabler with an xDL interface description. From this interface description, an adequate learning

alphabet has to be constructed (cf. D4.3). For this, the Learning Enabler uses the StrawBerrytool [5, 23], which is based on type analysis. This yields many ambiguities, as most web services, for example, rely on strings as their only datatype. However, in CONNECT, interface description are assumed to be semantically annotated with references to a common domain ontology. Considering for instance a login operation, this allows a clear distinction between the parameters for username and password—which are both of type string—by means of references to different concepts in the ontology.

The Learning Enabler thus further refines the learning alphabet, performing an ontology-aware analysis of the data dependencies between the single invocations. In order to capture the expressiveness of ontologies, this step is supported by the Pellet OWL-DL reasoner [29], which allows to reason about the various concepts referred to by the parameter instances.

The second enhancement of the Learning Enabler's operation concerns the learning of non-functional properties, as described in Section 2.5. For the purpose of generating QoS data points during learning (cf. Section 2.5.1), the automatically generated mapper has to be implemented with probes recording, e.g., information about latencies. At the end of the learning process, an Enhanced LTS (cf. D3.3) is produced, which is subsequently annotated with non-functional properties derived from the data points generated during learning.

2.6.3 Conclusion

The Learning Enabler shows the strength of LearnLib's flexible, module-oriented approach. External components—such as the Interaction Enabler—can easily be integrated as test drivers. This enables the use of automata learning techniques in a large number of environments, such as in the CONNECT scenario for learning behavioral specifications of networked systems. The successful integration tests underline the ability of the developed prototype to serve as an integral part of the CONNECT architecture, working in a fully distributed environment.

2.7 Integrating the Monitoring Enabler

During Y4, the overall monitoring system architecture has been updated in order to more effectively cooperate with the enablers within the CONNECT architecture. In particular, the resources consumption has been reduced and some testing concerning the intrusiveness of the GLIMPSE probes has been started.

Concerning the integration of the GLIMPSE Monitoring Enabler, the monitoring infrastructure is now able to cooperate with all the CONNECT enablers involved in the off-line and on-line analysis of the CONNECTor, namely:

- Deployment Enabler: to be informed about the deployment/runtime status of the CONNECTor;
- Dependability Enabler: in order to improve and ensure CONNECTability;
- Security Enabler: in order to notify exceptions occurred into the CONNECTor;
- Learning Enabler: to improve the learned models of the networked systems.

A fully working version of GLIMPSE has been released on the CONNECT website.

Finally, the translator of CPMM (Connect Property MetaModel) models to GLIMPSE rules (Drools) is now completed (this activity involves Monitoring on WP4 and CPMM on WP5), so that the monitor can be integrated in a fully model-driven approach. The detailed description of CPMM can be found in Deliverables D5.3 and D5.4.

In the following subsections, we provide a more in-depth description of the Y4 advancements.

2.7.1 Final Implementation of the Model-driven Monitoring Infrastructure

The released monitoring infrastructure GLIMPSE is a generic and flexible instrument that aims to provide the developer and the user with a set of tools to monitor functional and non-functional properties without

the need to manage low-level coding details, but just using a meta-model for defining the non-functional property to be monitored.

GLIMPSE is event-based. In CONNECT, an event represents a method invocation on a remote web service: the invocation, from the producer to the consumer, is captured when it comes through the CONNECTor, encapsulated into a specific object, and sent through the CONNECT bus.

We report below in Figure 2.5 the main GLIMPSE components, including the new one that has been added in the fourth year of the project, the CPMM Model Converter. Details of the other monitor components have been included in Deliverables D4.2 and D4.3.

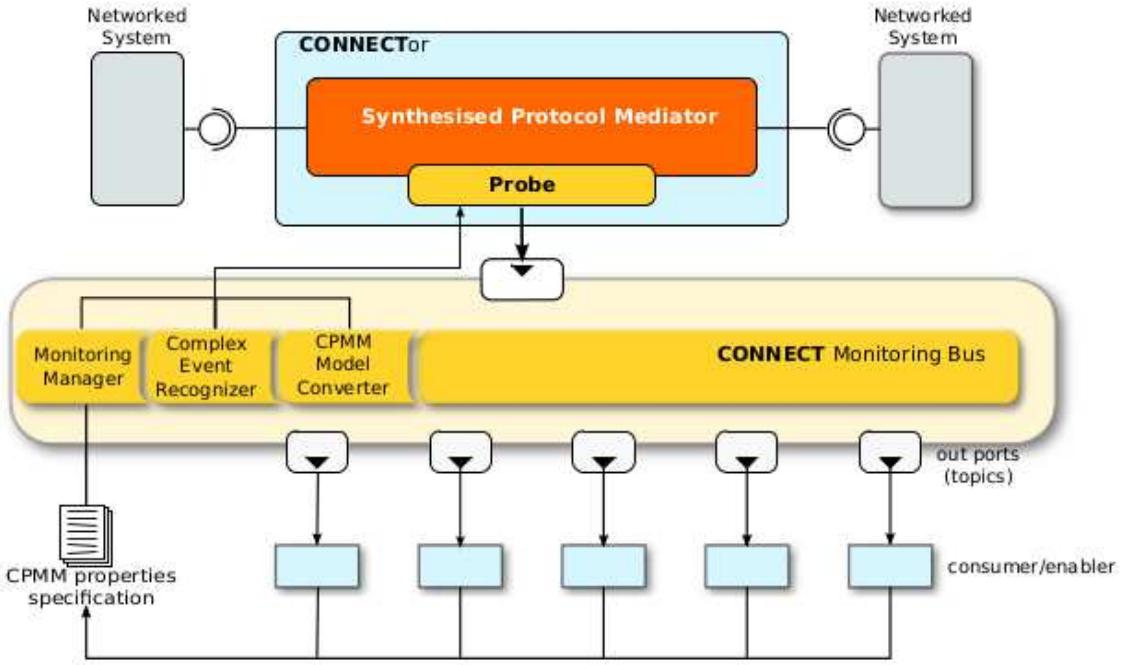


Figure 2.5: The GLIMPSE infrastructure providing the monitor enabler behaviour

2.7.2 Properties Specification

The properties object provided to GLIMPSE contains a JMS ObjectMessage that must respect the following three prerequisites:

1. the contained object must be a CPM (CONNECT property model) object;
2. the ObjectMessage must have a property ”DESTINATION” sets to ”monitor”;
3. the ObjectMessage must have a property ”SENDER” sets with the consumerName.

The object stored into the JMS ObjectMessage sent from the enabler (consumer of the monitoring) to GLIMPSE is the model of the non-functional property that the enabler wants to be monitored expressed using the CPMM.

Once received this message and relative payload, the Manager component will forward it to the CPMM Model Converter that is described below.

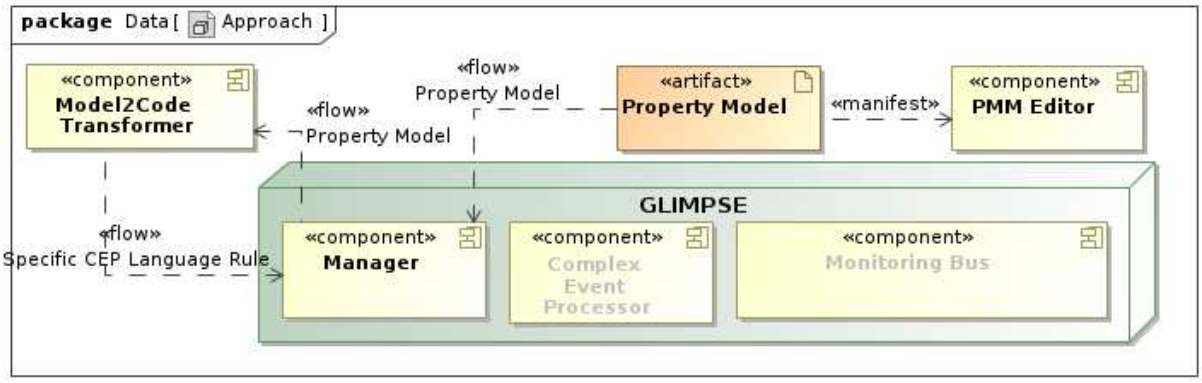


Figure 2.6: CPMM Model Converter infrastructure

Listing 2.1: Main Template of Model2Code Transformer

```
[comment encoding = UTF-8 /]
[module generate('cpmm/model/Core.ecore', 'cpmm/model/EventType.ecore', '
    cpmm/model/EventSet.ecore', 'cpmm/model/Metrics.ecore', 'cpmm/model/
    MetricsTemplate.ecore', 'cpmm/model/Property.ecore')]
[import cpmm::acceleo::utilities::utility]
[template public generateElement(p : Property)]
[comment @main/]
[file (p.name, false, 'Cp1252')]
 [if p.oclIsTypeOf(QuantitativeProperty)]
  [processMainQuantitativeProperty(p)/]
 [elseif (p.oclIsTypeOf(QualitativeProperty))]
  [processQualitativeProperty(p)/]
 [/if]
[/file]
[/template]
```

2.7.3 CPMM Model Converter

This component is the new one added into the GLIMPSE architecture. This component takes as input an XML file contained in the payload of the JMS ObjectMessage sent through the GLIMPSE bus to the manager component and generated using the CPMM Eclipse editor. This XML file is provided as input to the CPMM Model Converter on which the Model2Code Transformer component implemented using Acceleo code generator IDE is running.

The output of the transformation is a XML file containing a Drools rule that is provided to the GLIMPSE Complex Event Processor to load it into the Knowledge Base.

More in detail, the implemented Model2Code Transformer, shown in figure 2.6 takes as inputs the CPMM ecore models (specifically Core.ecore, EventSet.ecore, Event-Type.ecore, Metrics.ecore, MetricsTemplate.ecore, Property.ecore) and the model (compliant to CPMM) of the property to be monitored, and produces one or more Drools rules. Since this activity is shared between WP4 and WP5, which is where the CPMM model has been defined, we report here for the sake of self-completeness an example of Model2CodeTransformer template. More details about the converter can be found in Deliverable D5.4.

The Model2Code Transformer we developed consists of a main generation rule or template that calls a certain number of other templates, each one is applied on a CPMM model element (or object) to produce some text. In each template there are static areas, that will be included as they are defined in the generated file, and dynamic areas, that correspond to the expression evaluation on the current object.

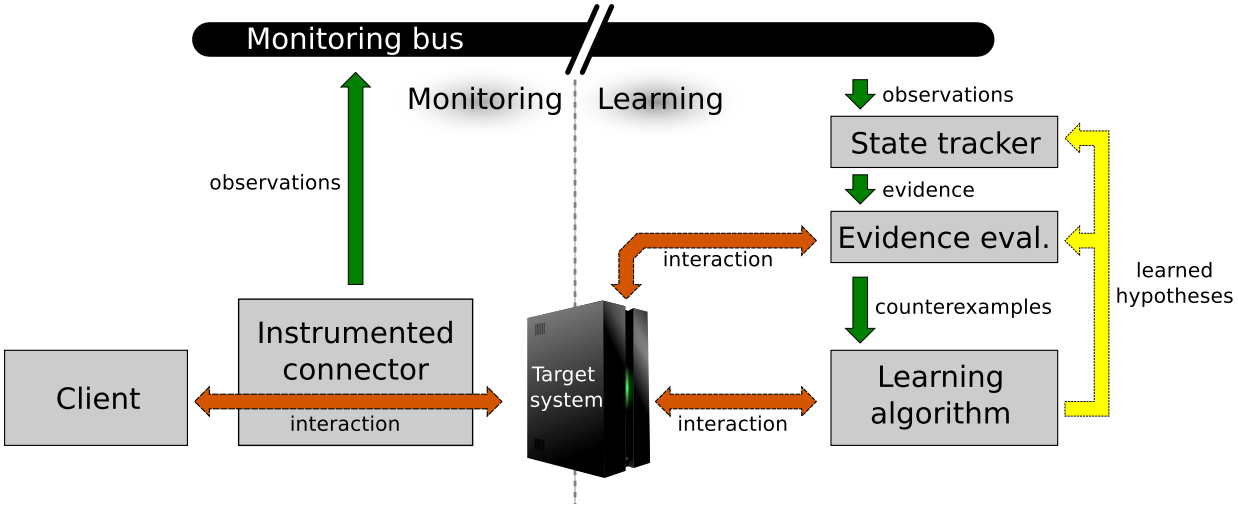


Figure 2.7: Overall architecture of the monitoring setup

2.8 Integration of Monitoring and Learning

The correctness of active automata learning depends on the availability of tests that can reject incomplete models and provide counterexamples that serve as witness for divergence between the learned automaton model and the target system. Thus the learning procedure is outfitted with testing facilities that try to identify diverging model behavior.

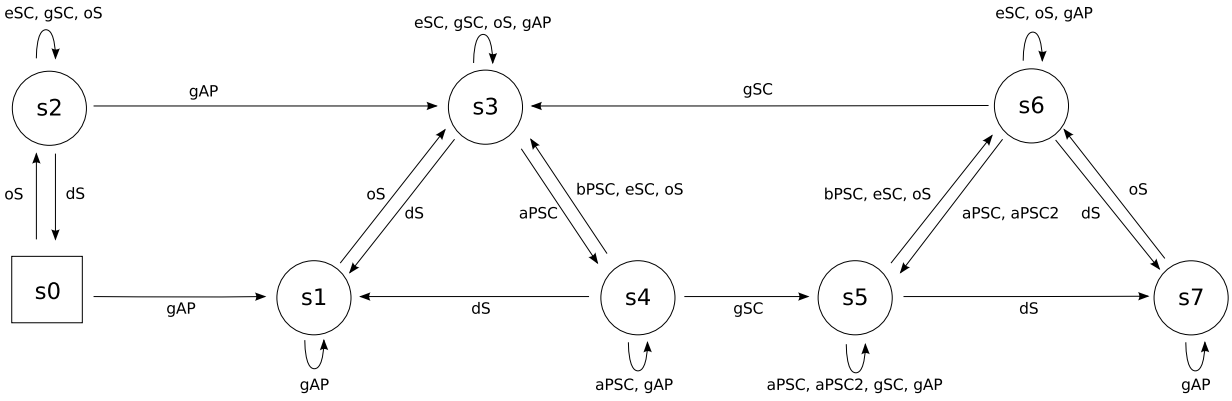
These tests, however, are in practice only approximate, meaning that (depending on the thoroughness of testing) errors may not be detected in all cases. In the CONNECT context it is desirable that learned models are provided in a timely fashion so that CONNECTors can be generated and deployed with little delay to cater for on-demand scenarios. Thus only little time should be spent on testing models, limiting the thoroughness of pre-synthesis testing procedures.

2.8.1 The Architecture of Monitoring Integration

Searching for model inadequacies at runtime (i.e., post-synthesis) can help to close this gap and paves the way for continuous evolution of learned models. In the CONNECT architecture the generated CONNECTor is outfitted with monitoring probes that can observe runtime interactions between connected systems. These observations can then be compared to the models that were provided by the learning enabler, triggering relearning in case of detected mismatches.

The overall anatomy of such a setup is illustrated in Figure 2.7. On the left hand side the synthesized CONNECTor, instrumented with monitoring probes, collects observations on the reactions of the target system to requests issued by, e.g., a client. These observations are delivered to a rule-driven monitoring bus provided in the form of GLIMPSE, ensuring that these messages arrive only at components that are part of the relearning process. On the right hand side, the learning enabler will compare the incoming observations with the current automata model of the target system, generating evidence of noncompliance if the observations do not match the current model hypothesis. These traces of evidence are subsequently checked against actual system behavior by means of system interaction and processed to constitute counterexamples that can be used to learn a refined hypothesis by resuming the learning process, generating an evolved model. This evolved model is delivered to CONNECTor synthesis and subsequently is the new subject of monitoring.

In Y4, we have finished integration between the respective components of the proposed monitoring/learning architecture. In order to assess its usefulness in a connect context, we have evaluated it on a connect-related e-commerce service. The results of this evaluation are presented in the following.



Legend	
os: openSession(username, password): session ds: destroySession(session): string gAP: getAvailableProducts(): productArray eSC: emptyShoppingCart(session): session	gSC: getShoppingCart(session): shoppingCart aPSC: addProductToShoppingCart(session, elementOf(productArray.getItem())): session aPSC2: addProductToShoppingCart(session, elementOf(shoppingCart.getItems()): session bPSC: buyProductsInShoppingCart(session): productArray

Figure 2.8: The target system of the integration case study

2.8.2 The Integration Evaluation

The soundness of the approach has been examined in a scenario where the first hypothesis provided by learning indeed is incomplete and is subsequently refined via monitoring. Regarding the architecture shown in Figure 2.7, the respective components were realized as follows:

- The *target system* in the examined scenario is a simple e-commerce service displayed in Figure 2.8. The model describes a shop application, where a client can log in, retrieve a list of products, add products to a shopping cart, and finally can purchase these products. Albeit rather compact, this model shows a certain degree of behavioral sophistication that makes it a compelling example for behavioral learning. For instance, as indicated in the model, the service only permits a purchase operation whenever the shopping cart is not empty. Also, every purchase operation will clear the shopping cart, meaning that it is not possible to issue two purchase commands in direct succession. This is essential information for being able to properly interact with the service and is not simply deducible by analyzing, e.g., the system interface alone.
- The *client* was simulated by a component that would randomly issue requests to the target system, eventually reaching every part of the target system and thus triggering observations that would cover the whole system. To get an estimate on how quickly relevant observations could be expected in real-life, this component was throttled so that in practice no more than six requests per second would be generated.
- GLIMPSE is the base technology for the *monitoring bus*. Given that many systems may be monitored concurrently, Drools rules are provided to ensure that observations are reliably separated so that per-system observations can be observed on the receiving end of the bus.
- The CONNECT*or* funneled all request issued by the client to the target system in an unfiltered fashion, i.e., without introducing additional behavior. Instead its mere role in the examined setup was to observe the traffic between the client and the target system and issue messages to the monitoring bus accordingly.
- Messages from the monitoring bus were received by the *state tracker*, which is a primary means to detect noncompliance of the learned model with incoming runtime observations. Given that the monitored hypothesis may be inadequate and that the internal state of the target system may not known when the first observations of client/system interaction arrive, the state tracker's initial task is to identify the state in the hypothesis model that corresponds to the internal state of the monitored

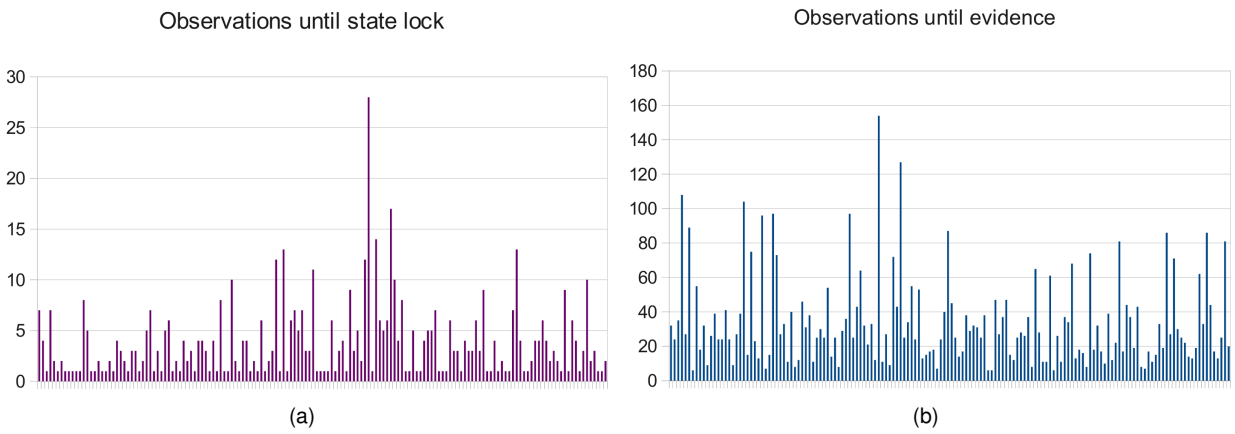


Figure 2.9: Number of consumed observations until (a) a lock onto a single state in the hypothesis is achieved and (b) evidence for non-compliance is generated.

system. This can be achieved by stepwise execution of incoming observations on all states of the hypothesis model and successive elimination of those states that could not produce results matching the actual observations. In practice incoming observations will rapidly "narrow down" the set of states that have to be considered, ideally converging just one single state, i.e., a "model lock" is achieved. If, however, the set of possible states becomes empty (i.e., the model lock is subsequently lost), this means that the hypothesis model could not support the sequence incoming observations. Thus the sequence of observations that lead to a lost model lock constitutes *evidence* for non-compliance. In the experimental setup only traces would be collected as evidence following a lock onto a single-state that subsequently was lost.

- Evidence for non-compliance may contain false positive errors ("false alarms") as, for example, spurious errors in the client/system communication or disruptions of the monitoring bus by network errors may lead to corrupt traces that, of course, cannot be explained by the learned hypothesis. Thus the *evidence evaluator* verifies incoming evidence by directly interacting with the target system and comparing system reactions to those produced by the current hypothesis. Evidence that passes this test (i.e., evidence that indeed exposes a hypothesis fault) is emitted as a *counterexample*.

Whenever counterexamples are generated the learning enabler resumes its learning procedure to produce a refined model. Some experimental results are presented in Figure 2.9. The following observations can be made:

- A state lock in the learned hypothesis was achieved within 4.01 observations on average. This means that identification of the assumed system state within the learned model is achieved within few seconds. This means that it is not a big issue that the system state is not known beforehand.

- For the initial (inadequate) learned hypothesis, evidence for noncompliance with the target system was generated on average after 33.65 observations. Again, this means that in the examined scenario counterexamples would usually be found well within one minute. Some traces of evidence, however, exceed 100 observations, meaning that counterexamples on occasion are vastly longer than minimal counterexamples that could be conceived (minimal counterexample lengths do not exceed the state count of the target system). This is, however, expected given the randomized nature of the client triggering the observations, which has no notion of orderly exploration. It is important to note, however, that real-life clients cannot be expected to employ an exploration strategy either.

2.8.3 Conclusion

The evaluation experiments showed that runtime monitoring architecture can successfully provide information to the learning component so that a refined model matching the system behavior could be generated. Moreover, the conducted experiments show that the monitoring architecture and the integration into the learning framework finalized during Y4 are able to support the goal of model evolution within CONNECT-related scenarios.

3 Overall Contributions by WP4 During CONNECT

In this chapter, we summarize the overall contributions achieved during the CONNECT project, discuss their impact and some remaining challenges. We structure the discussion into the following topics:

- extending the power and efficiency of learning algorithms,
- implementation of learning,
- monitoring infrastructure, and
- integration of the learning enabler.

3.1 Extending the Power and Efficiency of Learning Algorithms

Background Synthesis of networked components, as envisaged in the CONNECT scenario, is inherently limited by the quality and descriptive power of the models of network peers that are provided by the learning enabler. Therefore, it has been the ambition of WP4 to develop learning algorithms that can generate rich behavioral models of component behavior. In WP2, it has been agreed to use transition-system models, more specifically a variant of interface automata, which have been developed both for the finite-state case and the case with data (see D2.3). It has thus been natural for WP4 to aim at learning such models, both with and without explicit representation of the communicated data objects.

Prior to the CONNECT project, active learning algorithms had been developed for finite-state models of components that utizilized a finite set of primitives for interaction. To support learning of realistic networked systems, one challenge of CONNECT has been to extend learning techniques to handle richer models, primarily models including data of various form. Such techniques should allow to learn models of many different system classes, including communication protocols, which require a proper treatment of parameters, and web services, where associated metadata descriptions should be exploited by suitable techniques.

Contribution To address this challenge, WP4 has developed techniques for extending active automata learning to the generation of rich models, in particular automata models extended with data. These techniques build on a number of new fundamental ideas that WP4 has introduced into the field of learning.

The first idea is to apply *symbolic abstraction techniques*, that had previously been developed for program analysis and formal verification, to the setting of active automata learning. Abstractions transform the problem of learning a rich, possibly infinite-state model, to the problem of learning a finite-state model. The latter problem can be addressed by existing standard techniques. In Y1, we showed how manually supplied abstractions could be applied to rich transition sytems models with data, on several case studies [1]. We also showed how such abstractions could be refined as a response to information about the inadequacy of the current abstraction [15].

The idea of applying symbolic abstraction techniques, however, was not sufficient for automated learning of rich models, since they relied on manual guidance, and could not be guaranteed to be efficient. We therefore developed *canonical automaton models*, based on Nerode-like congruences. Nerode congruences allow to guarantee that the learning of a component will succeed with a bounded number of test inputs, where the bound reflects the complexity of the component. We first developed canonical models for *register automata*, where data values could only be tested for equality [6]. In spite of the restriction to canonicity, our approach provided natural and succinct models on typical applications. In Y3 and Y4, this approach was generalized to a richer class of automata, covering also tests on sequence numbers, membership of objects in lists, etc. [6].

Finally, by developing a technique for *systematic search* for a least refined abstraction among the possible canonical automata models, we obtained a novel class of active learning algorithms, which can learn models that combine control and data. This learning algorithm can be systematically derived from the finite-state case by exploiting the fact that the automaton model is based on a Nerode congruence, which

induces a symbolic abstraction. It can thus be regarded as *the* natural generalization of active learning, from the finite-state case to the infinite-state case. 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. A first presentation of this learning algorithm appeared in [13] and is also discussed in [11]. In Y4, we substantially extended both the scope and the efficiency of the learning algorithm [12]. We have applied the new implementation to a range of examples. 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. Thus, this work represents a breakthrough in the area of automata learning, and outperforms previous approaches to learning models that combine control and data.

Along with the above fundamental conceptual development, WP4 has developed a large number of enhancements and improvements to active automata learning, including *effect-directed learning* [22] and the *use of type information* [24]. Most prominent is maybe the algorithmic improvements in the search for new test inputs that allowed to win the ZULU competition [14].

Finally, the work in WP4 has been validated in a large number of different case studies that have been reported in the published papers.

Impact During the course of the CONNECT project, the attention for the area of automata learning has grown significantly. It is now recognized as a potentially important part of test generation and automated formal verification. The related area of "specification mining", in which intended specifications of software components are inferred by observing their behavior, has grown in importance, and provides many opportunities for application of our automata learning techniques.

Our introduction of symbolic abstraction techniques into automata learning [1] is already being referenced by other groups. The line of work has been continued, e.g., by the group of Frits Vaandrager at the University of Nijmegen (see, e.g., http://www.italia.cs.ru.nl/).

Remaining Challenges During CONNECT, we have established a new line of research in the area of active automata learning. Two lines of continued work suggest themselves.

- One line of work consists in further refining and optimization the approach for different classes of models in different application areas. One could, e.g., develop a protocol learning engine that would generate models of implemented communication protocol, or a corresponding engine for networked services. These engines would be equipped with techniques for anticipated data domains and application-dependent optimizations.
- Another line of work consists in optimizing the approach towards situations where the budget for available test sequences is restrictied. For instance, a scenario could be to learn a model of a component using at most 100 test sequences. We expect that our generated theory of learning rich models is a suitable basis also for solutions to contexts like this one. Solutions can be developed by leveraging the experience gained from our implementation for the ZULU competition [9, 14], and from the field of passive learning.

3.2 Implementation of Learning

During the CONNECT project, our framework for active automata learning, LearnLib, has been thoroughly reengineered in order to make the learning functionality available as reusable components [22]. The resulting modularized framework is structured according to the basic principles of the MAT (Minimally Adequate Teacher) paradigm, accounting for structures such as learning algorithms, test drivers, equivalence checks and application-specific filters. This ensures that a multitude of learning algorithms following that paradigm can be implemented within a rich supporting infrastructure, enabling the composition of application-fit learning setups, utilizing interchangeable components provided by LearnLib. This flexibility enabled the creation and refinement of the learning setup that eventually won the ZULU competition [9, 14]. The overall structure is also fit to support the arrival of new automata models such as register

automata [6] and their respective learning algorithm, witnessed by implementations within the framework provided by LearnLib. Being publicly available for free, LearnLib has seen adoption by groups that are not part of the CONNECT project. It is intended to release LearnLib as open source project, creating further incentive for adoption and open development.

To enable unsupervised learning in the context of on-demand instantiation of connectors, a reusable test driver was developed that can be configured to drive interaction with a wide spectrum of networked services. The per-system configuration is obtained by means of interface analysis, backed by ontologies. This enables the creation of automata models for networked systems in a fully automated way, which is an important step forward compared to the pre-CONNECT state, where test drivers and their configurations were created manually.

Remaining Challenges To enable accelerated evolution and adoption of the learning technology developed during CONNECT, LearnLib should be made available as open source project, coupled with a community-building effort to encourage 3rd parties to contribute.

3.3 Monitoring Infrastructure

Timely and effective run-time adaptation can only be ensured by continuously observing the interactions among the networked systems. To this purpose, in CONNECT we have developed a modular, flexible and lightweight monitoring infrastructure, called GLIMPSE. GLIMPSE is a comprehensive event-based publish-subscribe infrastructure for monitoring, which is the central instrument within the CONNECT architecture to provide self-awareness and to drive runtime adaptation.

The major technical contribution of this monitoring solution is its *model-driven orientation*, which is based on two key elements:

1. a generic monitoring infrastructure that offers the greatest flexibility and adaptability; and
2. a coherent set of domain-specific languages, expressed as metamodels, that enable us to exploit the support to automation offered by model-driven engineering techniques.

Concerning 1), GLIMPSE is totally generic and can be easily applied to different contexts. Such infrastructure supports all the main functionalities required to a monitoring infrastructure, including data collection, local interpretation, data transmission, aggregation, and reporting. It has been fully integrated within the CONNECT architecture, and is now able to cooperate with all the CONNECT enablers involved in the off-line and on-line analysis of the CONNECTor: namely, the Deployment Enabler, DePer, the Security Enabler, and Learning Enabler. A fully working version of GLIMPSE has been released on the CONNECT website.

Concerning 2), the monitor fully implements a Model-driven approach, so that the functional and non-functional properties to be observed can be specified by models conforming to the CPMM meta-model defined in WP5. In this respect, GLIMPSE is thus an advancement of state-of-art in runtime monitoring infrastructures, as no similar approaches with the maturity and comprehensiveness of CPMM exist. A translator from CPMM (Connect Property MetaModel) models to GLIMPSE rules (Drools) has also been released. A detailed description of CPMM can be found in D5.3 and D5.4.

Remaining Challenges While the integration of the monitoring infrastructure within CONNECT has been finalized, showing promising results, the overall impact of the monitoring approach, e.g., in terms of performance feasibility, has still to be determined by means of more extensive experimentation.

Concerning the design of GLIMPSE, it was conceived to be generic and extensible, and in fact we plan to further refine and experiment this infrastructure by exploiting it in future research projects in the scope of service-oriented architecture. We have made GLIMPSE freely available for use from the research community.

One important direction for future work is to extend the infrastructure to deal with events that can happen at different layers. In fact, the emphasis of the service-oriented paradigm natively drives the

building of software systems as multi-layered. Consequently, monitoring has often to be able to deal with layer-specific events, and addressing layer-specific issues. Work in this direction has already been started, as described in [10], where we describe a Multi-source Monitoring Framework that can correlate the messages monitored at business-service level, with the observations captured by the infrastructure monitoring the low level resources.

3.4 Integration of the Learning Enabler

In Sections 3.1 and 3.2, we have described the technical developments on learning, driven by the requirements in the CONNECT project, which have resulted in stand-alone tools or libraries, such as LearnLib. On the other hand, for the specifics of the CONNECT scenarios, we have also developed techniques specifically for the realization of the Learning Enabler as a component in the CONNECT architecture, providing a smooth interoperability with the various other CONNECT enablers. The Learning Enabler builds on the integration of a number of tasks, including:

- receiving interface descriptions, and generating corresponding test harnesses,
- applying semantic and syntactic analyses of interface descriptions, in order to tailor and adapt test harnesses and learning algorithms,
- equipping test harnesses with probes that gather additional information, including non-functional properties,
- storing results,
- producing models in suitable formats: for CONNECT this means in particular the Enhanced LTS format, and
- annotating the generated models.

From the outside perspective, these operations constitute a single learning operation for a networked system, to be invoked solely by the Discovery Enabler.

Remaining Challenges As of now, the Learning Enabler has only been evaluated in CONNECT scenarios with a comparatively low number of states. The learning technology itself, however, should be able to support systems with more advanced complexity, which should be examined. Also, investigating mechanisms to cover additional non-functional properties may present interesting results.

4 Assessment of WP4 Results

In this chapter, we provide an overall evaluation of the WP4 results with respect to the objectives and assessment criteria discussed in D6.3, hence refining the preliminary evaluation contained in that deliverable. We recall that the objective of WP4 is to support a bootstrapping mechanism which can start from minimal information about networked components and generate usable behavioral models. The approach taken for learning behaviours is assessed according to whether and how well it can take account of previous knowledge, according to the quality of learned models, and the range of aspects (data, non-functional properties) that can be covered. It is also assessed whether and how well learning can support evolution by means of techniques for incremental learning, involving to detect deviations from learned behavior and trigger appropriate learning action.

As criteria for evaluation, we use exactly the same ones as in the preliminary assessment reported in Section 5.2.4 of Deliverable 6.3. These criteria are structured according to three objective. For each criterion, we provide on the one hand a general assessment concerning the general capabilities of WP4 results, on the other hand (under Objective 2.1) a particular assessment against the GMES scenarios and demonstrator provided by WP6. In these scenarios, learning technology was successfully used to generate behavioral models of components such as a service providing weather data and a service that provides access to a flying drone. In each case, the system could be accurately described as a finite state machine that models the complete input/output-behavior. These behavioral models can be obtained without much delay, including the time inherent to networked invocation. The accuracy of the learned models was verified by manual inspection. In summary, the learning technology proved its ability to generate accurate models fit for connector synthesis in a time frame that is suitable for ad-hoc CONNECT scenarios.

Objective 1: Learning should be able to exploit available information about a network component, and should minimally need only a description of interfaces

Assessment Criteria:

1. How much prior description is needed for learning.
2. How well learning can exploit additional information

Assessment: concerning

How much prior description is needed for learning: API descriptions are enough, and there are some translation tools that transform API descriptions automatically into the required alphabet format. Thus no special description is required in order to start the learning process at the API-level [24, 25]. This analysis of interface descriptions can be backed up by ontologies, providing, e.g., information on data dependencies and preset values. However, in practice it is often important to optimize the alphabet description for a particular learning goal, e.g. to establish a particular level of abstraction. Such optimizations cannot be automated in general, but one may apply certain common patterns, e.g., for data treatment that also easily covers a lot of these more specific scenarios [30]. A chosen abstraction can automatically be refined to resolve nondeterminism introduced in case the abstraction is “too coarse” [15]. One of the most elaborate techniques for e.g., dealing with infinite data structures is the learning of register automata developed in the context of CONNECT, where the required structural restriction (data indepence) is typically given. Several case studies (see, e.g., Chapter 5 of D4.3) have demonstrated how the learning proceeds automatically from the availability of interface descriptions and information about where to direct invocations.

How well learning can exploit additional information: Additional knowledge can favorably be used for constructing tailored abstractions (e.g., exploiting ontological information), or to steer the learning process itself (e.g., knowledge about the intent behind the system). In case this information is formally provided in an adequate format, both exploitations can be achieved automatically. e.g., grouping API-level symbols to ontology-level symbols is very easy. It might,

however, introduce unwanted non-determinism, which can then automatically be taken care of by means of our Automated Alphabet Abstraction technology [15]. Steering the learning process on the basis of intent expressed in terms of temporal logics has also been implemented [16]. Type information in interface descriptions can be used to enhance learning setups, and make learning more efficient. This is demonstrated in Chapter 4 of D4.3. The overall performance of automata learning can be dramatically improved by employing application-specific information to answer learning queries, as is demonstrated in several case studies [19, 2, 32]. Ontologies can be used during alphabet construction [25].

Objective 2: To generate accurate behavior models of networked components, covering a range of aspects

Assessment Criteria:

1. How accurate are the learned models
2. What range of aspects can be covered by learned models.

Assessment: concerning

How accurate are the learned models: The accuracy of the learned models depends on the chosen level of abstraction, and the intensity of learning (learners would say 'on the quality of the equivalence query', which signalizes the success of the learning process). It lies in the nature of (active) automata learning that it is not possible to state concrete quality results beyond: the obtained models are the most concise representation of all the observed knowledge. The problem always remains to decided, when one can stop with enforcing more observations. This situation calls for a 'life-long learning' process (discussed, e.g., in [27]), which we enforce by monitoring: Learned models are put into the application context and validated while the system is running. This allows us to detect any deviations from the modeled behavior and, in turn, the corresponding update of the models. This guarantees a continuous improvement of the learned models. A significant set of case studies has demonstrated that learning can produce sizable and detailed models of components.

In the GMES scenario developed by WP6 with the goal of showing the interplay of all the developed technologies, learning technology was successfully used to generate behavioral models of components such as a service providing weather data and a service that provides access to a flying drone. In each case, the system could be accurately described as a finite state machine that models the complete input/output-behavior. These behavioral models can be obtained without much delay, including the time inherent to networked invocation. The accuracy of the learned models was verified by manual inspection. In summary, the learning technology proved its ability to generate accurate models fit for connector synthesis in a time frame that is suitable for ad-hoc CONNECT scenarios.

What range of aspects can be covered by learned models: Learning has been originally designed to infer languages, or in our case, potential runtime traces (of reactive systems). Typically, this comprises functional behavior (at least as long as it is deterministic), but it may well also comprise certain non-functional properties. In case these are deterministic, active learning would work straightforwardly. Otherwise special care has to be taken. Experience with combining active learning with passive learning to detect non-functional properties like performance, dependability and the like is very promising. Data parameters can also be handled in learning: for simple examples this is is demonstrated in Chapter 3 of D4.3. Apart from generating models, the learning procedure can also assemble non-functional data, as is discussed, e.g., in Section 2.5 and in extended form in [27].

Objective 3: To support evolution

Assessment Criteria:

1. Whether and how well evolution can be detected, and how well learning updates models.

Assessment: Evolution is detected whenever some (monitored) observation concretely contradicts previous observations. The easiest way to deal with such situations is to simply replace the old by the new observation. This has, however, a drawback: the central invariant for learning, namely ”the obtained models are the most concise representation of all the observed knowledge” can no longer be guaranteed. If this invariant is important, one should re-validate the observations the current hypothesis is based upon and adapt the model accordingly. As this does not require any equivalence queries this can often been done quite efficiently. This re-validation step can also be applied at any time one wants to check for system evolutions currently not adequately coved by the model. The infrastructure to support evolution has been developed, integrating GLIMPSE [3] and learning technology, and experimented with on simple examples, but not yet assessed on a full scenario.

The evolution of models is not a notion that is only relevant to the immediate CONNECT context, but has already been employed on real-life systems [26, 27].

5 Conclusion

Let us again recall the challenges of WP4, as stated in the DoW.

> *'... 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.'*

During its 46 months of operation in CONNECT, WP4 has addressed these challenges, by advancing the scientific state-of-the-art, implementing tools that are integrated into the CONNECT architecture, and evaluating the techniques and tools on a range of case studies.

Scientific State-of-the-Art In CONNECT, WP4 has significantly advanced the state-of-the-art on active automata learning, by extending active automata learning to the generation of rich models, in particular automata models extended with data. By developing a number of fundamentally new techniques and ideas, WP4 has developed a natural generalization of active learning, from the finite-state case to the infinite-state case (with data). 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. On the way, WP4 has also made significant contributions to automata theory for languages over infinite alphabets. In addition, WP4 has developed a large number of enhancements and improvements to active automata learning, e.g., witnessed by winning the ZULU competition.

WP4 has also devised and implemented a comprehensive event-based publish-subscribe infrastructure for monitoring, which is the central instrument within the CONNECT architecture to provide self-awareness and to drive runtime adaptation. The monitor fully implements a model-driven approach, so that the functional and non-functional properties to be observed can be specified by models conforming to the CPMM meta-model defined in WP5. GLIMPSE is thus an advancement of state-of-art in runtime monitoring infrastructures, as no similar approaches with the maturity and comprehensiveness of CPMM exist. CPMM models are automatically translated into Drools rules, which is GLIMPSE's native language.

Software Tools The work in CONNECT has resulted in further development and creation of software tools for learning and for monitoring.

- **LearnLib** is the leading tool that implements active automata learning. During CONNECT, the LearnLib software has been extended and reengineered into a flexible and versatile framework for active automata learning that is based on a clear component structure. Based on LearnLib, CONNECT-enabled components such as reconfigurable test-drivers have been implemented and integrated into the learning enabler. The learning technology has demonstrated the ability to create models for actual systems both within the scope of CONNECT and in scenarios with preexisting systems.
  LearnLib is available for download from https://www.connect-forever.eu/software.html, including documentation and a tutorial.
- **GLIMPSE** is an advancement of state-of-art in runtime monitoring infrastructures, through its model-driven approach. Functional and non-functional properties to be observed can be specified by models conforming to the CPMM meta-model defined in WP5. No similar approaches with the maturity and comprehensiveness of CPMM exist.
  GLIMPSE is available for download from https://www.connect-forever.eu/software.html, including usage information and relevant publications.

Contributions to Overall CONNECT Challenges Both LearnLib and GLIMPSE have been integrated into the CONNECT architecture, in the form of the learning and monitoring enablers. Through a number of case studies, it has been demonstrated that they can respond to the initial challenge outlined in the DoW: starting from a minimal description of a syntactic interface, a model of the dynamic behavior of a networked system can be learned, also with data and non-functional aspects. Furthermore, using the monitoring enabler, the model can be continuousl checked for conformance with the actual system and for evolution. Deviations and evolution will trigger relearning activity.

A number of case studies and integration experiments underline the ability of the developed learning and monitoring enablers to fulfill their roles as an integral part of the CONNECT architecture, working in a fully distributed environment, and also support the goal of model evolution within CONNECT-related scenarios.

Bibliography

[1] 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.

[2] O. Bauer, J. Neubauer, B. Steffen, and F. Howar. Reusing system states by active learning algorithms. In *EternalS'11*, volume 255 of *Communications in Computer and Information Science*, pages 61–78. Springer, 2011.

[3] A. Bertolino, A. Calabrò, F. Lonetti, and A. Sabetta. Glimpse: a generic and flexible monitoring infrastructure. In *Proceedings of the 13th European Workshop on Dependable Computing*, EWDC '11, pages 73–78, New York, NY, USA, 2011. ACM.

[4] A. Bertolino, A. Calabrò, M. Merten, and B. Steffen. Never-stop learning: Continuous validation of learned models for evolving systems through monitoring. *ERCIM News*, 2012(88), 2012.

[5] 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.

[6] 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.

[7] S. Cassel, F. Howar, B. Jonsson, M. Merten, and B. Steffen. A succinct canonical register automaton model, 2012. Invited for publication in *Journal of Logic and Algebraic Programming*.

[8] S. Cassel, F. Howar, B. Jonsson, and B. Steffen. A succinct canonical register automaton model for data domains with binary relations. In *Automated Technology for Verification and Analysis - 10th International Symposium, ATVA 2012, Thiruvananthapuram, India, October 3-6, 2012. Proceedings*, volume 7561 of *Lecture Notes in Computer Science*, pages 57–71. Springer, 2012.

[9] 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.

[10] A. B. Hamida, A. Bertolino, A. Calabrò, G. D. Angelis, N. Lago, and J. Lesbegueries. Monitoring service choreographies from multiple sources. In *Software Engineering for Resilient Systems - 4th International Workshop, SERENE 2012, Pisa, Italy, September 27-28, 2012. Proceedings*, volume 7527 of *Lecture Notes in Computer Science*, pages 134–149. Springer, 2012.

[11] F. Howar. *Active learning of interface programs*. PhD thesis, TU Dortmund, Department of Computer Science, Chair of Programming Systems, 2012.

[12] F. Howar, M. Isberner, B. Steffen, O. Bauer, and B. Jonsson. Inferring semantic interfaces of data structures. In *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change - 5th International Symposium, ISoLA 2012, Heraklion, Crete, Greece, October 15-18, 2012, Proceedings, Part I*, volume 7609 of *Lecture Notes in Computer Science*, pages 554–571. Springer, 2012.

[13] F. Howar, B. Steffen, B. Jonsson, and S. Cassel. Inferring Canonical Register Automata. In *Proc. VMCAI 2012*, volume 7148 of *Lecture Notes in Computer Science*, pages 251–266. Springer Verlag, 2012.

[14] F. Howar, B. Steffen, and M. Merten. From ZULU to RERS - Lessons Learned in the ZULU Challenge. In Margaria and Steffen [20], pages 687–704.

[15] 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*, volume 6538 of *Lecture Notes in Computer Science*, pages 263–277. Springer, 2011.

[16] M. Isberner. Untersuchung der Optimierbarkeit regulaerer Extrapolationsverfahren durch Ausnutzung vorhandenen Wissens. Master's thesis, TU Dortmund, Department of Computer Science, Chair of Programming Systems, 2011.

[17] M. Karusseit and T. Margaria. Feature-based modelling of a complex, online-reconfigurable decision support service. In *WWV '05, 1st Int. Worksh. Automated Specif. and Verification of Web Sites*, March 2005. ENTCS 1132.

[18] T. Margaria and M. Karusseit. Community usage of the online conference service: an experience report from three cs conferences. In *Proceedings of the IFIP Conference on Towards The Knowledge Society: E-Commerce, E-Business, E-Government*, pages 497–511, Deventer, The Netherlands, The Netherlands, 2002. Kluwer, B.V.

[19] T. Margaria, H. Raffelt, and B. Steffen. Knowledge-based relevance filtering for efficient system-level test-based model generation. *ISSE*, 1(2):147–156, 2005.

[20] 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.

[21] T. Margaria and B. Steffen, editors. *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change - 5th International Symposium, ISoLA 2012, Heraklion, Crete, Greece, October 15-18, 2012, Proceedings, Part I*, volume 7609 of *Lecture Notes in Computer Science*. Springer, 2012.

[22] M. Merten. *Active automata learning for real-life applications*. PhD thesis, TU Dortmund, Department of Computer Science, Chair of Programming Systems, 2012.

[23] M. Merten, F. Howar, B. Steffen, P. Pelliccione, and M. Tivoli. Automated inference of models for black box systems based on interface descriptions. In Margaria and Steffen [21], pages 79–96.

[24] M. Merten, F. Howar, B. Steffen, P. Pelliccione, and M. Tivoli. Automated inference of models for black box systems based on interface descriptions. In Margaria and Steffen [21], pages 79–96.

[25] M. Merten, M. Isberner, F. Howar, B. Steffen, and T. Margaria. Automated learning setups in automata learning. In Margaria and Steffen [21], pages 591–607.

[26] J. Neubauer, T. Margaria, and B. Steffen. Design for Verifiability: The OCS Case Study. In *Formal Methods for Industrial Critical Systems: A Survey of Applications*. John Wiley & Sons, 2011. In print.

[27] J. Neubauer, B. Steffen, O. Bauer, S. Windmüller, M. Merten, T. Margaria, and F. Howar. Automated continuous quality assurance. In *Proc. FormSERA: 1st Int. Workshop on Formal Methods in Software Engineering: Rigorous and Agile Approaches*, pages 37–43. IEEE, 2012.

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

[29] E. Sirin, B. Parsia, B. C. Grau, A. Kalyanpur, and Y. Katz. Pellet: A practical owl-dl reasoner. *Web Semant.*, 5(2):51–53, June 2007.

[30] B. Steffen, F. Howar, and M. Isberner. Active Automata Learning: From DFAs to Interface Programs and Beyond (invited). In *ICGI 2012*, pages 21:195–209. JMLR W&CP, 2012.

[31] 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.

[32] B. Steffen and J. Neubauer. Simplified validation of emergent systems through automata learning-based testing. *SEW 2011*, 2011.

6 Appendix: Enclosed Publications

Enclosed with this deliverable, we provide two full journal papers, resulting from collaborative work in CONNECT, and presenting different key elements of the overall WP4 contribution. The reported papers have been attached in the order listed below.

- S. Cassel, F. Howar, B. Jonsson, M. Merten, and B. Steffen. A succinct canonical register automaton model. Invited for publication in *Journal of Logic and Algebraic Programming*, 2012.

  This paper is an extended and more elaborate version of the first paper on this topic [6].

- F. Howar, B. Steffen, B. Jonsson, S. Cassel, and M. Isberner. Inferring canonical register automata. Submitted for publication, 2012.

  This paper is an extended description of our approach for learning automata with data, which supersedes both the papers [13] and [12].

A Succinct Canonical Register Automaton Model[☆]

Sofia Cassel[a], Falk Howar[b], Bengt Jonsson[a], Maik Merten[b], Bernhard Steffen[b]

[a] *Dept. of Information Technology, Uppsala University, Sweden*
[b] *Chair of Programming Systems, University of Dortmund, Germany*

Abstract

We present a novel canonical automaton model, based on register automata, that can be used to specify protocol or program behavior. Register automata have a finite control structure and a finite number of registers (variables), and process sequences of terms that carry data values from an infinite domain. Here, we consider register automata that compare data values 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. This canonical form is well suited for modeling, e.g., protocols or program behavior. Since it captures only essential relations between data values, it can be exponentially more succinct than previous proposals, and opens the way to new practical applications, e.g, in automata learning.

Keywords: register automata, data languages, canonical model, Myhill-Nerode, automata theory

1. Introduction

When modeling a system, it is crucial to be able to capture not only control aspects but also data aspects of its behavior. Often, systems are

[☆]Supported in part by the European Commission FP7 project CONNECT (IST 231167).

Email addresses: sofia.cassel@it.uu.se (Sofia Cassel), falk.howar@cs.tu-dortmund.de (Falk Howar), bengt.jonsson@it.uu.se (Bengt Jonsson), maik.merten@cs.tu-dortmund.de (Maik Merten), bernhard.steffen@cs.tu-dortmund.de (Bernhard Steffen)

modeled as finite automata. Finite automata can be and have been extended with data, for example as timed automata [1], counter automata, data-independent transition systems [16], and different kinds of data automata and register automata. Many of these types of automata have long been used for specification, verification, and testing (e.g., [19]).

In our context, *register automata* [3, 14, 8] is a very interesting formalism. A register automaton (RA) has a finite set of registers (a.k.a. variables) and processes sequences of terms that carry data values from an infinite domain. It can compare data values, e.g., for equality, and assing data values to registers. Register automata can thus can be regarded as a simple programming language, with variables, parallel assignments, and conditions.

Modeling and reasoning about systems becomes significantly easier if we can use canonical automata. This is exploited in equivalence and refinement checking, e.g., through (bi)simulation based criteria [15, 18], and in automata learning (a.k.a. regular inference) [2, 11, 20]. There are standard algorithms for minimization of finite automata, based on the Myhill-Nerode theorem [12, 17], but it has proven difficult to carry over similar constructions to automata models over infinite alphabets, including timed automata [23].

Recently, canonical automata based on extensions of the Myhill-Nerode theorem have been proposed for languages where data values can be compared for equality [10, 3, 6], and also for inequality, when the data domain is equipped with a total order [3, 6]. These canonical models are, however, obtained at the price of rather strict restrictions on how data is stored in variables, and on which guards may be used in transitions. Examples of such restrictions are uniqueness (two variables may not store identical data values) and order (a fixed ordering is enforced between variables). Both of these restrictions may cause a blow-up in the number of states in the canonical automaton.

Another cause of unwanted blow-ups is the encoding of relations between data values regardless of whether they are essential[1] or not. For example, a cross-product of two independent automata, representing the interleaving of two independent languages, may result in a blow-up due to encoding non-essential relations between data values of the two languages.

Motivated by the above restrictions, we propose a novel form of register automata for languages where data is compared for equality. Our model

[1]By *essential* relation, we mean a test which is necessary for recognizing the language.

avoids unnecessary blow-ups in the number of states, while preserving canonicity. Thus two variables may store identical data values, and we do not restrict access to variables to a specific order or pattern. This supports a much more intuitive modeling of data languages, while leaving the expressiveness untouched. The idea behind our approach is to filter out non-essential relations between data values, resulting in register automata that are minimal in a certain class.

By a non-technical analogy, we could compare the difference between the automata of [10, 3] and our canonical form to the difference between the region graph and zone graph constructions for timed automata. The region graph considers all possible combinations between constraints on clock values, be they relevant to acceptance of the input word or not, whereas the zone graph construction aims to consider only relevant constraints. The analogy is not perfect, however, since our automata are always more succinct than those of [10, 3].

In summary, our main contribution is a Nerode congruence for a form of register automata. This formalism can easily be used to specify protocol or program behavior since it provides a canonical representation of any (deterministic) RA-recognizable data language.

Related Work. Among the first to generalize regular languages to infinite alphabets were Kaminski and Francez [14] 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 [22, 5, 4] for an overview). Most of these formalisms recognize data languages that are invariant under permutations on the data domain. In [7] a logical characterization of data languages is given plus a transformation from logical descriptions to automata.

While most of the above mentioned work focuses on non-deterministic automata and is concerned with closedness properties and expressiveness results of data languages, we are interested in a framework for (deterministic) register automata that can be used to model the behavior of protocols or (restricted) programs. This includes, in particular, the development of canonical models on the basis of a Myhill-Nerode-like theorem. Kaminski and Francez [10], Benedikt et al. [3], and Bojanczyk et al. [6] all present Myhill-Nerode theorems for data languages with equality tests, which prompted us to propose the more succinct construction described in this paper.

Organization. In the next section, we illustrate our main ideas using a simple example. Then, we present our register automaton model as a basis for representing data languages. In Section 4, we introduce a succinct representation of data languages, allowing us to use a limited number of data words to represent an entire data language. Based on this representation, we define a Nerode congruence in Section 5 and prove that it characterizes minimal canonical forms of deterministic register automata, called (right-invariant) DRAs. In Section 6 we relate our canonical form to other formalisms, and establish some exponential succinctness results, before we conclude in Section 7.

2. An illustrative example

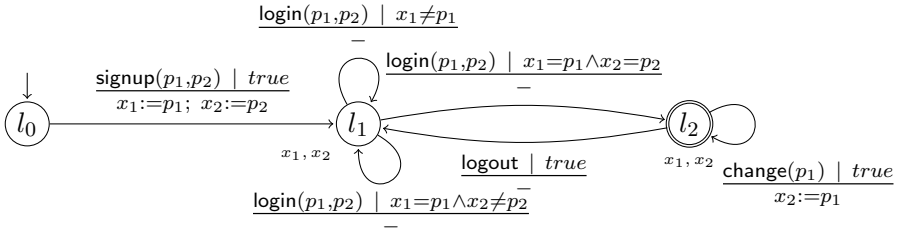


Figure 1: Partial model for a fragment of XMPP

In this section, we provide an overview of our main ideas using a simple running example. We have chosen to model the behavior of a fragment of the XMPP protocol [21], which is widely used in instant messaging. More specifically, we focus on the mechanism for signing up for an account, logging in and out, and changing the account password.

We model the behavior of XMPP as a *data language*, where each data word represents an execution of the protocol. A data word is thus a sequence of messages with parameters, for example $\mathsf{signup}(\mathtt{Bob}, \mathtt{lemon})\ \mathsf{login}(\mathtt{Bob}, \mathtt{lemon})$ where the parameter `Bob` is a username and the parameter `lemon` is a password. A data word is in the XMPP language if the corresponding execution of the protocol leads to a user being logged in. A user is logged in after executing a sequence of messages of the form $\mathsf{signup}(p_1, p_2)\ \mathsf{login}(p_1', p_2')$, where the password and username supplied in the signup message must match those in the login message.

Let us now consider the problem of succinctly representing languages such as the XMPP language using *register automata*. It seems reasonable that any such automaton would need to remember the password and username supplied in the $\mathsf{signup}(p_1, p_2)$ message in order to be able to compare them to those in any subsequent login message.

A naive way of doing this is to construct an automaton that performs all available tests on input parameters. This automaton will first test whether the username and password supplied in the signup message are equal. If they are, it will initialize one location variable; if they are not, it will initialize two location variables to store the username and password. Then the automaton will test whether the stored values match the ones supplied in the login message. It will follow a transition to an accepting state if they match and to a rejecting state if they do not match. Each possible valuation of tests available (here, only equality) will correspond to a separate location, resulting in a canonical automaton.

There are, however, other ways to achieve canonicity. In this paper, we represent a data language as a classification that states for any data word whether the word is in the language or not. Then we can find the minimal set of *essential* data words, and extend the classification of this set to any set of data words that includes the essential words. An essential data word has the property that two parameters are equal only if they absolutely must be, i.e., if the data word would be classified differently without this equality between parameters. For example, both words $\mathsf{signup}(\mathtt{Bob}, \mathtt{lemon})\ \mathsf{login}(\mathtt{Bob}, \mathtt{lemon})$ and $\mathsf{signup}(\mathtt{Bob}, \mathtt{Bob})\ \mathsf{login}(\mathtt{Bob}, \mathtt{Bob})$ are in the XMPP language. Obviously, whether the username and password are equal does not affect whether a word is in the XMPP language or not.

When we have determined the subset of essential data words for a data language, we can apply a Nerode-like congruence to obtain a canonical register automaton. Each essential word will then represent at most one location in the final register automaton.

A canonical automaton that recognizes our selected fragment of XMPP is shown in Figure 1. It is a register automaton with three control locations: one initial location (l_0), one location where the user has signed up for an account but not yet logged in (l_1), and one accepting location where the user is logged in (l_2). The initial location is marked by an arrow and the accepting location is denoted by two concentric circles. The automaton has two location variables, denoted x_1, x_2, which are initialized in location l_1.

The automaton reads one message at a time, and then follows a transition

to another location if the corresponding guard is satisfied, assigning new values to location variables. For example, if the automaton is in location l_0 and reads the message $\mathsf{signup}(\texttt{Bob}, \texttt{lemon})$, it will follow a transition to l_1, storing `Bob` in the variable x_1 and `lemon` in the variable x_2. Note that the automaton does not care whether the message parameters are equal or not; the message $\mathsf{signup}(\texttt{Bob}, \texttt{Bob})$ would store `Bob` in both x_1 and x_2.

When the next message is read, the automaton checks whether it is of the form $\mathsf{login}(p_1, p_2)$ and whether the guard $x_1 = p_1 \wedge x_2 = p_2$ is satisfied by the parameters in the message and the location variables. If this is the case, the automaton goes to the accepting location l_2. Thus, the word $\mathsf{signup}(\texttt{Bob}, \texttt{lemon})\ \mathsf{login}(\texttt{Bob}, \texttt{lemon})$ is accepted by the automaton. If the parameters and location variables do not satisfy the guard, the automaton simply loops in location l_1.

In the following sections, we will give formal definitions of the concepts in this example.

3. Data languages and register automata

Assume an unbounded domain D of data values and a set of *actions*. Each action has a certain *arity* that determines how many data values it carries 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. We will sometimes write $\bar{d}$ for $d_1, \ldots, d_n$. A *data word* is a sequence of data symbols. A *data language* is a set of data words that is closed under permutations on D.

Assume a set of formal parameters. Let a *parameterized symbol* be a term of form $\alpha(p_1, \ldots, p_n)$, where α is an action with arity n and $p_1, \ldots, p_n$ are formal parameters. We will sometimes write $\bar{p}$ for $p_1, \ldots, p_n$.

Assume a finite set of *variables* (a.k.a. registers), ranged over by $x_1, x_2, \ldots$.

A *guard* is a conjunction of equalities and negated equalities over formal parameters and variables, e.g., $p_i = p_j$ or $p_i \neq x_j$, .

Definition 1. *A register automaton 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 the* source location*,*
  - *l' is the* 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 variable $x \in X(l')$ is assigned the value of $\rho(x)$).*
- *$\lambda : L \mapsto \{+, -\}$ maps each location to either $+$ (accept) or $-$ (reject).* □

Semantics of a register automaton. A register automaton $\mathcal{A}$ classifies data words as either accepted or rejected. We 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.

We now describe the semantics of a register automaton more precisely. Consider the register automaton $\mathcal{A} = (L, l_0, X, \Gamma, \lambda)$. A *valuation*, denoted by ν, is a mapping from location variables to the domain D of data values. For a conjunction g over variables and data symbols, we write $\nu \models g$ if ν makes g evaluate to true.

A *state* of $\mathcal{A}$ is a pair $\langle l, \nu \rangle$ where l is a location in L and ν is a valuation over the variables in $X(l)$. The *initial state* of $\mathcal{A}$ is the pair $\langle l_0, \nu_0 \rangle$ where l_0 is the initial location and ν_0 is the empty valuation.

A *step* $\langle l, \nu \rangle \xrightarrow{\alpha(\bar{d})} \langle l', \nu' \rangle$ of a register automaton transfers the automaton from the state $\langle l, \nu \rangle$ to the state $\langle l', \nu' \rangle$ on the input $\alpha(\bar{d})$. Each step $\langle l, \nu \rangle \xrightarrow{\alpha(\bar{d})} \langle l', \nu' \rangle$ of $\mathcal{A}$ is derived from a transition $\langle l, (\alpha, \bar{p}), g, \rho, l' \rangle \in \Gamma$ such that

1. the valuation ν is such that $\nu \models g[\bar{d}/\bar{p}]$, i.e., ν makes g evaluate to true when $p_i \cdots p_n$ in g are replaced by $d_i \cdots d_n$, and
2. the valuation ν' is the updated valuation, where $\nu'(x_i) = \nu(x_j)$ whenever $\rho(x_i) = x_j$, and $\nu'(x_i) = d_j$ whenever $\rho(x_i) = p_j$.

A *run* of a register automaton $\mathcal{A}$ 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 $\lambda(l_k) = +$ and *rejecting* if $\lambda(l_k) = -$. The automaton $\mathcal{A}$ accepts a data word w if there is an accepting run over w. The data language recognized by $\mathcal{A}$, denoted $L(\mathcal{A})$ is the set of data words that it accepts.

A register automaton is *completely specified* if for any state $\langle l,\nu\rangle$ and any input $\alpha(\bar{d})$, there is a transition $\langle l,(\alpha,\bar{p}),g,\rho,l'\rangle \in \Gamma$ such that $\nu \models g[d/\bar{p}]$. A register automaton is *determinate* if the runs over any data word w are either all rejecting or all accepting runs. We refer to automata that are completely specified and determinate as DRAs.

In this paper, we only consider determinate, rather than deterministic, register automata. The reason for this is that our construction of canonical register automata (in Theorem 2) results in register automata that are determinate but not necessarily deterministic. By strengthening the transition guards, a determinate register automaton can easily be made deterministic.

In general, a DRA can have transition guards that imply $x_i{=}x_j$ or $x_i{\neq}x_j$ for two distinct location variables x_i, x_j. In a *right-invariant* DRA, transition guards may not imply any such relation between location variables. More precisely, a DRA is right-invariant if for each transition $\langle l,\alpha(\bar{p}),g,\rho,l'\rangle$,

- the guard g does not imply $x_i{=}x_j$ or $x_i{\neq}x_j$ for distinct $x_i,\ x_j \in X(l)$, and
- the combined effect of the guard g and the assignment ρ does not imply any equality between distinct variables $x_i,\ x_j \in X(l')$ (note that negated equalities may be implied).

In a right-invariant DRA, two data values can thus only be tested for equality if one of them is in the current input symbol. Furthermore, if the guard of a transition contains a test for equality between two data values, then at most one of them can be stored in a location variable of the target location. In this paper, we only consider right-invariant DRAs, and assume that any DRA mentioned is right-invariant unless otherwise stated.

The issue of regularity for data languages has been discussed, e.g., in [4]. One required property in order for a data language to be regular is the existence of an automaton model that recognizes it. In this paper, we focus

on that particular aspect of regularity and thus restrict our attention to data languages that are *DRA-recognizable*, i.e, accepted by a DRA.

4. A succinct representation of data languages

In this section, we will describe how to represent any data language (not necessarily DRA-recognizable) using a minimal set of data words.

For any set S, let a *classification* of S be a mapping from S to $\{+, -\}$. A data language can thus be seen as a classification of a set of data words which is closed under permutations on the domain of data values.

Consider a data word w. Let $Vals(w)$ be the (ordered) sequence of data values in w, and let $Acts(w)$ be the sequence of actions in w. Let n be the number of data values in w. For $k \leq n$, let $Vals(w)|_k$ denote the sequence of the first k data values in w, and let $Acts(w)|_k$ denote the prefix of $Acts(w)$ that has the first k data values as parameters. For example, if $w = a(1,2)b(3)$, then $Vals(w)|_2 = 1, 2$ and $Acts(w)|_1 = a$.

For two data words w, w', let $w \simeq w'$ denote that w' can be obtained from w by a permutation on the domain D. We will choose exactly one *representative* data word from each equivalence class of $\simeq$ in the following manner: without loss of generality, let the domain D be the set of positive integers. A data word w is representative if the data values that appear in any prefix of $Vals(w)$ is of form $\{1, 2, \ldots, k\}$ for some k. Thus the data word $a(1,1)b(2)$ is representative, but the data word $a(1,3)b(2)$ is not. Clearly, there is exactly one representative word in each equivalence class of $\simeq$.

We are now able to represent a data language by a classification of the set of representative data words. As we shall see, we can actually represent a data language by classifying only a subset of these representative data words, called the set of *essential* data words. In order to determine this subset, we will introduce two partial orders on the set of representative words. In the following, whenever a data word is mentioned, we will assume that it is a representative data word unless otherwise stated.

$\sqsubseteq$ Let $w \sqsubseteq w'$ denote that w' can be obtained from w by a mapping on D, i.e., whenever two data parameters are equal in w they are also equal in w'. For example, $a(1,2)b(1) \sqsubseteq a(1,1)b(1)$. The smallest elements w.r.t. $\sqsubseteq$ are data words where all data values are pairwise different; the largest elements are data words where all data values are equal.

Let $w = w' \sqcup w''$ denote that w is the least upper bound of w' and w'' w.r.t. $\sqsubseteq$, i.e., whenever two data parameters are equal in w' or in w'' they are also equal in w. This also implies that $w' \sqsubseteq w$ and $w'' \sqsubseteq w$. Note that the least upper bound w is unique since data words must be representative.

We call a set $\mathtt{W}$ of data words $\sqsubseteq$-*closed* if $w' \in \mathtt{W}$ and $w \sqsubseteq w'$ imply $w \in \mathtt{W}$.

$<$ Let $w < w'$ denote that for some $k > 0$,

- $Vals(w)|_{k-1} = Vals(w')|_{k-1}$ and $Acts(w)|_k = Acts(w')|_k$,
- the kth data value of $Vals(w)$ is different from any of the $k-1$ first data values of $Vals(w)$, but
- the kth data value of $Vals(w')$ is equal to some of the $k-1$ first data values of $Vals(w')$.

Note that we do not require $Acts(w) = Acts(w')$. Let $w \leq w'$ denote that $w = w'$ or $w < w$. Note that $w \sqsubseteq w'$ implies that $w \leq w$.

Intuitively, $w < w'$ means that w and w' share the same prefix up to some $k-1$. Then, the next data value in w' is equal to one of the previous data values, whereas the next data value in w is not equal to any of the previous data values. For example, $a(1,2)b(3) < b(1,2)c(2)) < a(1,1)b(2))$, while $b(1,2)c(2)$ and $a(1,2)b(1)$ are not related by $<$.

Let w be a data word such that $Vals(w) = d_1 \cdots d_n$ and let $0 \leq k \leq n$. Then w is *flat after* k if for all $i > k$, the ith data value in w is different from all other data values in w. Intuitively, this means that the data values $d_{k+1}, \ldots, d_n$ are different from each other and from all data values among $d_1, \ldots, d_k$. For example, the word $a(1,1)b(2,3)$ is flat after 2, and the word $a(1,2)b(2,3)$ is flat after 3.

A data word w is a k-*flattening* of a data word w' if there is some $k \geq 0$ such that $Acts(w)|_k = Acts(w')|_k$, $Vals(w)|_k = Vals(w')|_k$, and w is flat after k. Intuitively, a k-flattening w of w' is equal to w' up to and including the kth data value, after which the only restriction is that each subsequent data value in w is unique.

Let $\mathtt{W}$ and $\mathtt{W}'$ be sets of data words. Then $\mathtt{W}'$ is $\mathtt{W}$-*flattening-closed* if $w \in \mathtt{W}'$ implies that all flattenings of w which are in $\mathtt{W}$, are also in $\mathtt{W}'$. For example, let $w'' = a(1,1)b(2,3)$ and $w' = a(1,2)b(3,4)$ be data words in a set $\mathtt{W}$. Both

w'' and w' are flattenings of a data word $w = a(1,1)b(1,1)$. Then the set $\{w,\ w',\ w''\}$ is $\mathtt{W}$-flattening-closed.

Let $\mathtt{W}$ be a $\sqsubseteq$-closed set of data words and let $\Phi \subseteq \mathtt{W}$ be a $\mathtt{W}$-flattening-closed set of (representative) data words. We can now use the two partial orders $\sqsubseteq$ and $<$ to describe how a classification of Φ can be extended to a classification of $\mathtt{W}$ itself.

Define a *best-matching* relation $\preceq_\Phi \subseteq \Phi \times \mathtt{W}$ between Φ and $\mathtt{W}$, by letting $w \preceq_\Phi w'$ iff w is a maximal (w.r.t. $<$) data word in Φ such that $w \sqsubseteq w'$.

For example, let $w = a(1,2)b(3,4)$, let $w' = a(1,2)b(1,2)$, let $w'' = a(1,2)b(1,3)$ and let $\Phi = \{w,\ w''\}$. Then $w \sqsubseteq w'$ and $w'' \sqsubseteq w'$, but since $w'' < w$, we have that $w \preceq_\Phi w'$.

The following theorem slightly generalizes Theorem 1 in [8].

Theorem 1 Let $\mathtt{W}$ be a $\sqsubseteq$-closed set of data words[2]. Let λ be a classification of $\mathtt{W}$. Then there is a unique minimal non-empty $\mathtt{W}$-flattening-closed subset $\Phi \subseteq \mathtt{W}$ of data words such that $\lambda(w) = \lambda(w')$ whenever $w \preceq_\Phi w'$ for $w \in \Phi$ and $w' \in \mathtt{W}$. □

By minimal, we mean that if Φ' is any other non-empty $\mathtt{W}$-flattening-closed set of data words that satisfies the above condition, then $\Phi \subseteq \Phi'$. We call the elements of this minimal set λ-*essential words*.

Theorem 1 implies that any classification λ of a $\sqsubseteq$-closed set $\mathtt{W}$ can be succinctly represented by its restriction to the set of λ-essential words. The value of λ for a λ-essential word w will then generalize to all data words $w' \in \mathtt{W}$ with $w \preceq_\Phi w'$. Note that for each $w' \in \mathtt{W}$, any non-empty $\mathtt{W}$-flattening-closed subset $\Phi \subseteq \mathtt{W}$ of data words must contain at least one data word w such that $w \preceq_\Phi w'$.

It is also important to take the domain of λ into account. As we will show in the proof, if $Dom(\lambda) \subseteq Dom(\lambda')$ and λ' agrees with λ on $Dom(\lambda)$, then the set of λ-essential words is a subset of the set of λ'-essential words.

Proof. We prove Theorem 1 by defining how a minimal set Φ of λ-essential words can be constructed incrementally for any classification λ of $\mathtt{W}$.

We first assume that there is a bound on the length of words in $\mathtt{W}$, i.e., $\mathtt{W}$ has only a finite set of data words. We then construct the set Φ of λ-essential words incrementally, considering data words in increasing $<$-order.

[2]Recall that we assume data words to be representative.

This works because, as we will show, only the set of λ-essential words less than (wrp. to $<$) a data word w is needed in order to determine whether w is λ-essential. Let $\Phi_{\mathtt{W}}^{<w}$ denote this set.

Initially, the set Φ contains only the data words in $\mathtt{W}$ that are flat after 0. A larger data word w is added to Φ if it must be added in order to classify some data word correctly. More precisely, we add data words to Φ in the following manner:

Let w be a data word of length n and let $k < n$ be the smallest integer such that w is flat after k (if there is no such k, let $k = n$). We claim that w is λ-essential iff

- there is some $(k-1)$-flattening of w that is λ-essential, and
- there is a data word $w' \in \mathtt{W}$ such that w is a k-flattening of w', and a λ-essential data word $w'' \in \Phi_{\mathtt{W}}^{<w}$, such that $w'' \preceq_{\Phi_{\mathtt{W}}^{<w}} w'$ and $\lambda(w'') \neq \lambda(w')$. □

The first condition is ensures that Φ is $\mathtt{W}$-flattening-closed. The second condition ensures that data words are correctly classified. Since we add data words to Φ in increasing $<$-order, we have already established the set of λ-essential words that are $<$-smaller than w, none of which are able to classify w' correctly. Because Φ must be $\mathtt{W}$-flattening-closed, we cannot later add some data word that is $<$-larger than w and classifies w' correctly. Thus, if we do not add w to Φ now, we will never be able to correctly classify w'.

To complete the proof, we need to show that if $Dom(\lambda) = \mathtt{W}$ and $Dom(\lambda') = \mathtt{W}'$, where $\mathtt{W} \subseteq \mathtt{W}'$ and and λ' agrees with λ on $Dom(\lambda)$, then the set of λ'-essential words includes the set of λ-essential words.

We thus show that if a data word w is λ-essential, it must also be λ'-essential. Without loss of generality, let w be a $<$-smallest data word that is λ-essential but not λ'-essential. This means that any $<$-smaller λ-essential word will also be λ'-essential.

As in the above definition of essential words, let w' be such that w is a k-flattening of w' for some k. Let $w'' \preceq_{\Phi_{\mathtt{W}}^{<w}} w'$ and let $\lambda(w'') \neq \lambda(w')$. In other words, let w'' be the data word that best matches w' in $\Phi_{\mathtt{W}}^{<w}$, where the classification of w'' does not match that of w'.

Since w is not λ'-essential, there must be some data word w''' in $\Phi_{\mathtt{W}'}^{<w}$ such that $w'' < w''' < w$, such that $w''' \preceq_{\Phi_{\mathtt{W}'}^{<w}} w'$, and such that $\lambda(w''') = \lambda(w')$.

We now construct a data word $\tilde{w}$ as $\tilde{w} = w'' \sqcup w'''$. Since $\tilde{w}$ is the least upper bound of w'' and w''', we have that $w'' \sqsubseteq \tilde{w}$ and $w''' \sqsubseteq \tilde{w}$. Consequently, $\tilde{w} \sqsubseteq w'$. By construction, w'' must be a $(k-1)$-flattening of w. Then, since $w''' < w$, we have that $\tilde{w} < w$.

Because $w'' \preceq_{\Phi^{<w}_{\mathtt{W}}} w'$ and $w'' \sqsubseteq \tilde{w} \sqsubseteq w'$, we have that $w'' \preceq_{\Phi^{<w}_{\mathtt{W}}} \tilde{w}$. Because $w''' \preceq_{\Phi^{<w}_{\mathtt{W}'}} w'$ and $w''' \sqsubseteq \tilde{w} \sqsubseteq w'$, we have that $w''' \preceq_{\Phi^{<w}_{\mathtt{W}'}} \tilde{w}$. We also know that $\lambda(\tilde{w}) = \lambda(w'')$ and $\lambda(\tilde{w}) = \lambda(w''')$. Then we arrive at a contradiction, since we already know that $\lambda(w'') \neq \lambda(w''')$.

We can now finally construct the set of λ-essential words when $Dom(\lambda)$ is any $\sqsubseteq$-closed set $\mathtt{W}$. Let λ_n be the restriction of λ to words with length at most n, and let Φ_n be the set of λ_n-essential words. Since, by the preceding paragraphs, $\Phi_n \subseteq \Phi_{n+1}$ for all $n > 0$, we obtain the set of λ-essential words as $\bigcup_{n=1}^{\infty} \Phi_n$. □

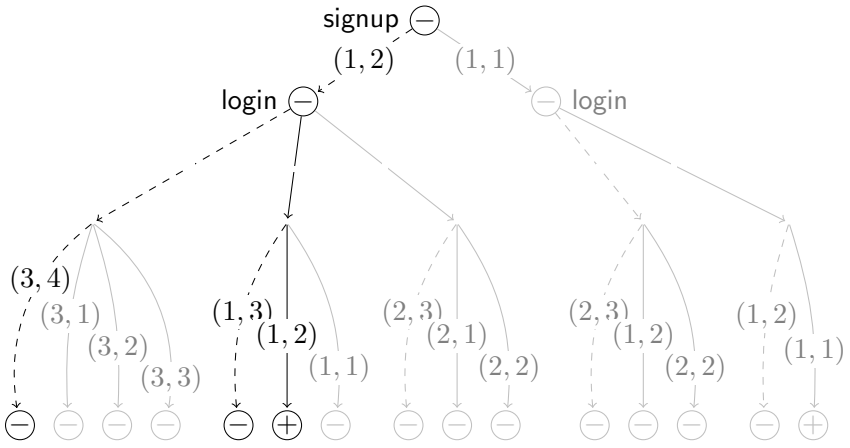


Figure 2: The set of λ-essential words for the XMPP language, as a tree

We can represent a $\mathtt{W}$-flattening-closed set Φ of data words visually as a tree. An example of a tree representing the XMPP language can be seen in Figure 2. The black branches and nodes represent λ-essential data words; non-essential data words are represented in gray. Words are read top-down in the tree: for example, the leftmost path represents $\mathsf{signup}(1,2)\ \mathsf{login}(3,4)$. Each node represents the data word that reaches it, and is marked $+$ or a $-$ to denote the classification of that word. The partial order $<$ runs from

left to right in the figure. Two nodes n_1, n_2 are incomparable by $<$ if they share a common predecessor with non-dashed paths to two children c_1, c_2, such that n_1 is a successor of c_1 and n_2 is a successor of c_2. For example, $\mathsf{signup}(1,2)\ \mathsf{login}(1,3)$ and $\mathsf{signup}(1,2)\ \mathsf{login}(2,2)$ are incomparable.

We can use this tree to describe how to inductively construct the set of λ-essential words for the XMPP language. First, all data words that are flat after 0 are essential (corresponding to the leftmost branch). We proceed with data words of length 1, i.e., with two parameters. The $<$-smallest data word is $\mathsf{signup}(1,1)$ (since $\mathsf{signup}(1,2)$ is flat after 0 and thus already processed) but it is not λ-essential, so we continue with words of length 2. We add the $<$-smallest data word $\mathsf{signup}(1,2)\ \mathsf{login}(1,3)$, because it is λ-essential. The data word $\mathsf{signup}(1,2)\ \mathsf{login}(2,3)$ is not λ-essential. Continuing from left to right according to the partial order $<$, we find that the data word $\mathsf{signup}(1,2)\ \mathsf{login}(1,2)$ is also λ-essential. Finally, the data word $\mathsf{signup}(1,1)\ \mathsf{login}(1,2)$ cannot be λ-essential, because $\mathsf{signup}(1,1)$ is not. The set of λ-essential words for the XMPP language now contains exactly the words represented by the black branches. Any other data word w can use the classification of a black branch, representing the word w', if $w' \preceq_w$, i.e., if the black branch is the rightmost (maximal w.r.t. $<$) branch such that $w \sqsubseteq w'$. For example, the data word $\mathsf{signup}(1,1)\ \mathsf{login}(1,1)$ is not λ-essential, and it is $\sqsubseteq$-larger than any word in the tree. The rightmost black branch is $\mathsf{signup}(1,2)\ \mathsf{login}(1,2)$ so we let $\mathsf{signup}(1,1)\ \mathsf{login}(1,1)$ use its classification.

5. Nerode congruence and canonical form

5.1. Suffixes

Our construction of the canonical DRA for a data language λ can, intuitively, be seen as 'folding' the set of λ-essential words so that runs of the canonical DRA correspond exactly to λ-essential words. Two λ-essential words w, w' will lead to the same location, if their possible 'continuations' into longer λ-essential words are equivalent. This equivalence corresponds to a Nerode-like congruence, which we will later define.

We assume an infinite ordered set $D_V = \{1, 2, 3, \ldots\}$, which is disjoint from D. Let a *suffix* be a data word whose data values are in $D \cup D_V$, such that the set of data values in D_V that appear in any prefix of its parameters is of form $\{1, 2, \ldots, k\}$ for some k. (This need not hold for the data values in D; e.g., $c(1,3)d(2,2)$ is a suffix.)

For a data word w, let a *w-suffix* be a suffix v such that $(Vals(v) \cap D) \subseteq Vals(w)$. For a data word w and a w-suffix v, let $w;v$ denote the unique representative data word of form wv' such that v' is obtained from v by an injective mapping $\sigma : D_V \mapsto (D \setminus Vals(w))$ mapping each element in D_V to a data value that does not occur in the word w. This will ensure that the data values in the suffix that are not equal to any data value in the prefix are mapped to 'fresh' data values when concatenating a prefix and a suffix. For example, let $w = a(1,2)$ be a data word and let $v = b(\mathtt{1},2)c(\mathtt{2},\mathtt{1})$ be a suffix. Then the unique representative data word $w;v = a(1,2)b(3,2)c(4,3)$ is obtained by letting $\sigma(\mathtt{1}) = 3$, and $\sigma(\mathtt{2}) = 4$.

Let $\mathtt{W}$ be a $\sqsubseteq$-closed set of data words, let λ be a classification of $\mathtt{W}$, and let w be a λ-essential word. Let a *λ-essential w-suffix* be a w-suffix v such that $w;v$ is a λ-essential word. Define the *λ-essential residual language* $\lfloor w^{-1}\lambda \rfloor$ as the classification of the set of λ-essential w-suffixes defined by $\lfloor w^{-1}\lambda \rfloor(v) = \lambda(w;v)$.

In order to compare, and determine equivalence between, continuations of two data words w and w', we must first provide a mapping between the data values of w and those of w'. If λ is a classification of a set of suffixes, and γ is a permutation on D, define $\gamma\langle\lambda\rangle$ as the classification of the set of suffixes $\{\gamma(v) \mid v \in Dom(\lambda)\}$, defined by $\gamma\langle\lambda\rangle(\gamma(v)) = \lambda(v)$. For example, let λ be a classification of a set $\{b(1,2),\ b(1,\mathtt{1}),\ b(\mathtt{1},\mathtt{2})\}$ of $a(1,2)$-suffixes, such that $\lambda(b(1,2)) = +$ and for any other suffix $v \in Dom(\lambda)$, $\lambda(v) = -$. Let γ be a permutation on the domain D of data values such that $\{\gamma(v) \mid v \in Dom(\lambda)\} = \{b(7,5),\ b(7,\mathtt{1}),\ b(\mathtt{1},\mathtt{2})\}$. Since the classification of a word or a suffix can only depend on the equalities and negated equalities between data values, we get that $\gamma\langle\lambda\rangle(\gamma(v)) = \lambda(v)$ for any suffix v in $Dom(\lambda)$.

Definition 2 (Memorable). Let $\mathtt{W}$ be a $\sqsubseteq$-closed set of data words, let λ be a classification of $\mathtt{W}$, and let w be a λ-essential word. Define the set of *λ-memorable* data values of w, denoted $\mathrm{mem}_\lambda(w)$, as the data values in $Vals(w)$ that also occur in some λ-essential w-suffix. □

Intuitively, $\mathrm{mem}_\lambda(w)$ is the set of data values that must be remembered after processing w in order to be able to correctly classify all continuations of w.

5.2. Nerode congruence

We are now ready to define a Nerode-like congruence on a set of λ-essential words. This congruence can then be used to construct a succinct DRA that recognizes the data language represented by the classification λ.

Definition 3 (Nerode congruence). Let $\lambda : \mathtt{W} \mapsto \{+, -\}$ be a classification of a set of representative data words. We define the equivalence $\equiv_\lambda$ on λ-essential words by $w \equiv_\lambda w'$ iff there is a permutation γ on D such that $\lfloor {w'}^{-1}\lambda \rfloor = \gamma\langle \lfloor w^{-1}\lambda \rfloor \rangle$. □

Intuitively, two representative data 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. The permutation γ used in Definition 3 need only relate memorable data values, i.e., it is enough to define it as $\gamma : \mathrm{mem}_\lambda(w) \mapsto \mathrm{mem}_\lambda(w')$. Then for any $\mathrm{mem}_\lambda(w)$-suffix v we have $w; v \equiv_\lambda w'; \gamma(v)$.

For example, consider $w = \mathsf{signup}(\mathtt{Bob}, \mathtt{lemon})\ \mathsf{login}(\mathtt{Bob}, \mathtt{lemon})$ and $w' = \mathsf{signup}(\mathtt{Bob}, \mathtt{lemon})\ \mathsf{login}(\mathtt{Bob}, \mathtt{lemon})\ \mathsf{change}(\mathtt{apple})$, such that $\mathrm{mem}_\lambda(w) = \{\mathtt{Bob}, \mathtt{lemon}\}$ and $\mathrm{mem}_\lambda(w') = \{\mathtt{Bob}, \mathtt{apple}\}$. Then we have $\gamma(\mathtt{Bob}) = \mathtt{Bob}$ and $\gamma(\mathtt{lemon}) = \mathtt{apple}$. Let $v = \mathsf{logout}\ \mathsf{login}(\mathtt{Bob}, \mathtt{lemon})$. Then $\gamma(v) = \mathsf{logout}\ \mathsf{login}(\mathtt{Bob}, \mathtt{apple})$ and

$$\mathsf{signup}(\mathtt{Bob}, \mathtt{lemon})\ \mathsf{login}(\mathtt{Bob}, \mathtt{lemon})\mathsf{logout}\ \mathsf{login}(\mathtt{Bob}, \mathtt{lemon})$$
$$\equiv_\lambda$$
$$\mathsf{signup}(\mathtt{Bob}, \mathtt{lemon})\ \mathsf{login}(\mathtt{Bob}, \mathtt{lemon})\ \mathsf{change}(\mathtt{apple})\mathsf{logout}\ \mathsf{login}(\mathtt{Bob}, \mathtt{apple}).$$

Guard transformation. Before we can use Definition 3 to construct a DRA, we must introduce a transformation from residual languages to constraints and transition guards.

Let Φ be a set of data words, let $w \in \Phi$ and let $Vals(w) = d_1 \cdots d_n$ be the ordered sequence of data values in w.

A conjunction $p_1 = p_{1'} \ \wedge\ p_2 = p_{2'} \ \wedge\ \cdots \ \wedge\ p_k = p_{k'}$ of equalities is on *normal form* if $1' < \cdots < k'$ and if all $p_1 \ldots p_{k'}$ are distinct.

- Let $eqs(w)$ be the conjunction (on normal form) of equalities of w, such that $p_i = p_j \in eqs(w)$ iff $d_i = d_j$.
- Let $neqs_\Phi(w)$ be the conjunction of negated equalities of w, such that $p_i \neq p_j \in neqs_\Phi(w)$ with $i < j$ iff $d_i \neq d_j$, and there is another data word $w' \in \Phi$ with $Vals(w)|_{j-1} = Vals(w')|_j$ and $Acts(w)|_j = Acts(w')|_j$, and where $p_i = p_j \in eqs(w')$.

Define the constraint φ^w_Φ of a data word w w.r.t. a set Φ of data words as $\varphi^w_\Phi \equiv eqs(w) \wedge neqs_\Phi(w)$. Let $\bar{p}$ be the sequence of parameters used to represent the data values in w. A data word w' *satisfies* a constraint φ^w_Φ if $Acts(w) = Acts(w')$ and $\varphi^w_\Phi[\bar{d}/\bar{p}]$ evaluates to true.

Lemma 1 *Let Φ be a set of data words and let $w \in \Phi$. A data word w' satisfies φ^w_Φ iff $w \preceq_\Phi w'$.*

PROOF. *If:* Assume that $w \preceq_\Phi w'$. We show that w' then satisfies φ^w_Φ. Let $Vals(w) = d_1 \cdots d_n$ and let $Vals(w') = d'_1 \cdots d'_n$. Recall that $w \preceq_\Phi w'$ implies that $w \sqsubseteq w'$, and thus also that $w \leq w'$. If $w = w'$, then by definition, w' satisfies φ^w_Φ. If $w < w'$, then there are (at least) two j, k such that $0 < j < k \leq n$ and such that $d_j \neq d_k$ but $d'_j = d'_k$. Then w' satisfies φ^w_Φ only if $p_j \neq p_k$ is not in φ^w_Φ.

Assume, to get a contradiction, that $p_j \neq p_k \in \varphi^w_\Phi$. Then, by definition, there is some other data word $w'' \in \Phi$ with $Vals(w'')|_{k-1} = Vals(w)|_{k-1}$ and with $p_j = p_k \in eqs(w'')$, i.e., such that $w < w''$. Since any equality in w is also in w'', we have that $w \sqsubseteq w''$ and, since any equality in w'' is also in w', we have that $w'' \sqsubseteq w'$. But then we would get $w'' \preceq_\Phi w'$ which contradicts our initial assumption. Thus $p_j \neq p_k$ cannot be in φ^w_Φ, and w' satisfies φ^w_Φ.

Only if: Assume that w' satisfies φ^w_Φ. We show (1) that $w \sqsubseteq w'$ and (2) that w is the $<$-largest word in Φ with this property, concluding that $w \preceq_\Phi w'$.

Since any equality in w will also be in the guard g^w_Φ, two data values d'_j, d'_k in w' must be equal if d_j and d_k are equal in w. Thus $w \sqsubseteq w'$.

Assume, to get a contradiction, that there is a data word $w'' \in \Phi$ such that $w < w''$ and $w'' \sqsubseteq w'$. By definition, there is some $0 < l < m \leq n$ such that $p_l = p_m \in eqs(w'')$ but $p_l \neq p_m \in \varphi^w_\Phi$. Since w' satisfies φ^w_Φ, we must have $d'_l \neq d'_m$. Then we get that $w' < w''$, so we cannot have $w'' \sqsubseteq w'$, contradicting our initial assumption. Thus $w \preceq_\Phi w'$.

We now state the main result of our paper, which relates our Nerode congruence to DRAs.

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

PROOF. *If:* We construct a DRA from a given equivalence $\equiv_\lambda$, as the DRA $\mathcal{A} = (L, l_0, X, \Gamma, \lambda)$, where

- L is given by the finitely many equivalence classes of the equivalence relation $\equiv_\lambda$ on λ-essential words. We assume that each equivalence class has exactly one representative data word.

- l_0 is $[\epsilon]_{\equiv_\lambda}$
- X maps each location $[w]_{\equiv_\lambda}$ to the set of λ-memorable data values of w.
- Γ is constructed as follows[3]: For each location $l = [w]_{\equiv_\lambda}$ in L, with λ-memorable data values $X([w]_{\equiv_\lambda})$ and for each λ-essential one-symbol w-suffix of the form $\alpha(\bar{d})$ where $\bar{d} = d_{k+1} \cdots d_n$, there is a transition in Γ of form $\langle l, (\alpha, \bar{p}), g, \rho, l' \rangle$, where
  - $l' = [w; \alpha(\bar{d})]_{\equiv_\lambda}$,
  - $\alpha(\bar{p})$ is such that $\bar{p} = p_{k+1} \cdots p_n$;
  - ρ maps location variables in l' to location variables in l and formal parameters in $\alpha(\bar{p})$, such that for $d_i \in \mathrm{mem}_\lambda(w)$ where $w; \alpha(\bar{d}) \equiv_\lambda w'$, $\rho(x_{d_i}) = p_j$ if $\gamma(d_i) = d_j$ for $d_j \in \bar{d}$, and $\rho(x_{d_i}) = x_{\gamma(d_i)}$ otherwise.
  - g is obtained from the constraint φ^w_Φ, where Φ is the set of all λ-essential words of form $w; \alpha(\bar{d})$, in the following way. For each conjunct $p_i = p_j$(or analogously, $p_i \neq p_j$) in φ^w_Φ,
    - $p_i = p_j$ (or analogously, $p_i \neq p_j$) is a conjunct in g if p_i, p_j are formal parameters in $\alpha(\bar{p})$, and
    - $x_{d_i} = p_j$ (or analogously, $x_{d_i} \neq p_j$) is a conjunct in g if p_j is a formal parameter in $\alpha(\bar{p})$ and $x_{d_i} \in X(l)$, and
- $\lambda([w]_{\equiv_\lambda}) = \lambda(w')$ whenever $w \simeq w'$.

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. By construction, the transition relation is total and the automaton is determinate.

Note that it does not matter which data word we choose to represent an equivalence class, since, by definition, if $w \equiv_\lambda w'$, we can use γ to map the memorable parameters of w to those of w' without affecting the classification of each word.

[3]We assume that data values and corresponding formal parameters are consecutively numbered and renamed to avoid clashes.

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

Consider an arbitrary sequence s of transitions of $\mathcal{A}$, of the form

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

The guards $g_1 \cdots g_k$ can be conjoined to form a constraint $\varphi_s = g_1 \wedge \rho_1(\cdots \wedge \rho_{k-1}(g_k))$

Let w' be a data word such that the sequence s of transitions generates a run over w'. If w' is a $\sqsubseteq$-minimal such data word, then $p_i = p_j \in eqs(w')$ iff $p_i = p_j \in \varphi_s$, and analogously for negated equalities. Then, by definition, $\varphi_\Phi^{w'} = \varphi_s$, where Φ is the set of λ-essential words. For any other data word w'' such that s generates a run over w'', we have that w'' satisfies $\varphi_\Phi^{w'}$ iff $w' \preceq_\Phi w''$. This ensures that $\mathcal{A}$ correctly classifies data words that satisfy any of its runs.

Assume any right-invariant DRA that accepts λ. The proof idea then is to show that two λ-essential words 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 using the definition of right-invariance, as follows.

We define a location variable $x \in X(l)$ as *non-live* if the value of x has no influence on whether a run that passes l will be accepting or rejection (we omit the precise details). Define a location variable to be *live* if it is not non-live. We can then establish that if w is a λ-essential word that causes a sequence of transitions leading to location l, then any live location variable in $X(l)$ must be assigned to a data value in $\text{mem}_\lambda(w)$. By right-invariance, there is thus a bijection between the live location variables in $X(l)$ and the data values in $\text{mem}_\lambda(w)$. It follows that whenever w and w' are two λ-essential words that reach the same location, then their residual languages must be equivalent w.r.t. $\equiv_\lambda$, and thus w and w' are equivalent according to Definition 3. □

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.

6. Comparison between different automata models

In this section, we compare our register automata to other proposed formalisms, showing that our models can be exponentially more succinct. We

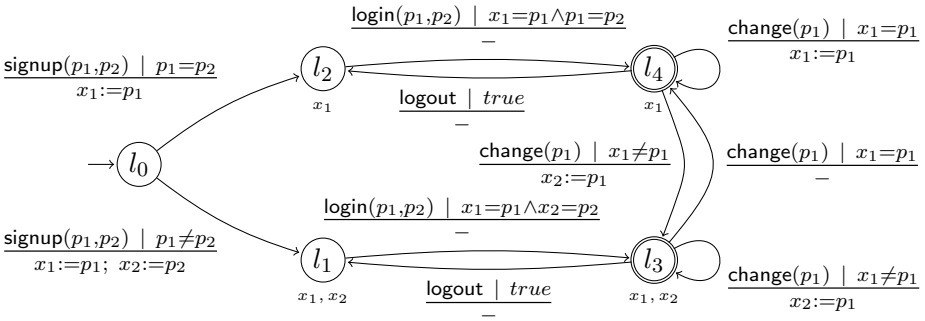


Figure 3: Deterministic unique right invariant RA

will discuss some of the restrictions imposed on these other kinds of register automata, and how they affect the size of the automata models. In particular, we will discuss register automata that require variables to store unique data values, to store variables in order of appearance, or both.

Unique-valued register automata. An RA is *unique-valued* if the valuation ν in any reachable state $\langle l, \nu \rangle$ is injective, i.e., two variables can never store the same data value. An example of a deterministic unique-valued RA (DURA) can be seen in Figure 3. This DURA accepts the same language (XMPP) as the DRA in Figure 1. Since variables are required to store unique values, the automaton transitions to one of two different locations after reading $\mathsf{signup}(d_1, d_2)$ depending on whether $d_1 = d_2$ or $d_1 \neq d_2$. After reading $\mathsf{change}(d_1)$, the automaton will be in either l_3 or l_4, depending on whether $d_1 = x_1$ (where x_1 stores the first data value from $\mathsf{signup}(d_1, d_2)$). The DURA thus needs five locations instead of three in order to accommodate the uniqueness constraint.

Ordered register automata. An RA is *ordered* if there is an ordering (here, we use $\ll$) of variables. In a deterministic ordered RA (DORA), if x_i and x_j are two location variables with $x_i \ll x_j$, then the transition at which x_i was last assigned a data value must coincide with or precede the transition at which x_j was last assigned a data value. A DORA accepting the XMPP language can look exactly like the automaton in Figure 1, but it can also look like the automaton in Figure 4. The difference lies in the order in which the data values in the first data symbol appear. Recall that in the signup data symbol, the first data value represents a username and the second, a

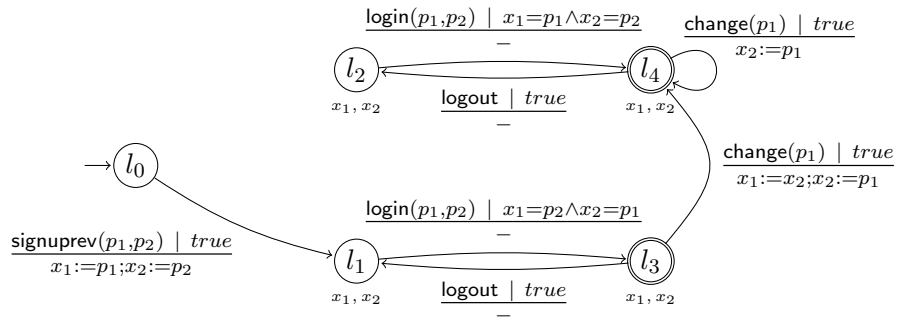


Figure 4: Deterministic ordered right invariant RA

password. The signuprev symbol is the signup symbol, but with the order of the data values reversed, i.e., the first data value now represents a password and the second, a username. If the order of the data values is thus reversed, then at the first occurrence of the change data symbol, the DORA will need to reorder the stored data values. This is necessary to allow changing the password later, since x_2 must be re-assigned a data value after x_1 is. This DORA thus needs five locations instead of three.

We can also define an OURA, which is both ordered and unique-valued. The automata of [3] correspond to deterministic OURAs (DOURAs). Figure 5 shows a deterministic OURA accepting the XMPP language. The DOURA must at any time store either one or two data values in variables (depending on whether they are equal or not). After the initial location, the automaton will be in locations l_2 or l_4 if only one variable is needed and transition to one of the other states whenever a second variable is needed. Compared to the DURA above, the DOURA thus needs an additional two locations to accommodate the uniqueness constraint. This results in a total of seven locations instead of five in the DORA.

As we have seen, automata that require variables to store unique data values, or to store data values in a certain order, may need more locations than the minimal DRA in order to accept the same data language. In the worst case, the blow-up can be exponential in terms of state space. In the models described here, we can identify two such exponential blow-ups: one between DRAs and DURAs, and another 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. The corresponding DURA then has to maintain in the set of locations which of the n

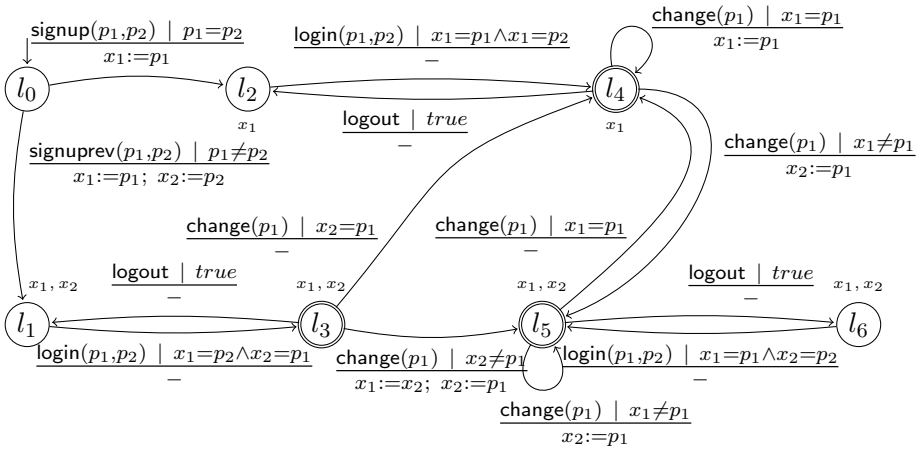


Figure 5: Deterministic ordered unique-valued right invariant RA

variables have the same value.

Proposition 1 There is a sequence of data languages $\mathcal{L}_1, \mathcal{L}_2, \ldots$, such that the number of locations in the minimal DRA for $\mathcal{L}_n$ is always 4, but the number of locations in the minimal DURA for $\mathcal{L}_n$ is exponential in n. □

Proof. Consider the language

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

where a and b are actions with arity n. The minimal DURA for $\mathcal{L}_n$ has an exponential number of locations after the first a-symbol to encode potential equalities between data values. Since each unique value can only be stored once in a DURA, one location is needed for every possible partition of $d_1, \ldots, d_n$ wrt. $=$. The minimal DRA, on the other hand, has only four locations: one initial, one after the first a-symbol, one accepting, and one rejecting sink location. □

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 then has to maintain in the set of locations the order in which the variables are written.

Proposition 2 There is a sequence of languages $\mathcal{L}_1, \mathcal{L}_2, \ldots$, such that the number of locations in the minimal DURA for $\mathcal{L}_n$ is $\mathcal{O}(n)$, but the number of locations in the minimal DOURA for $\mathcal{L}_n$ is exponential in n. □

Proof. The idea is to construct a language that is accepted by a DURA with n variables that can be accessed randomly, while the DOURA will have to encode the concrete sequence of accessed variables into the set of locations.

Let A contain the symbols a_i and b_i for $i = 1, \ldots, n$, each of which is of arity one. We will define a language that is accepted by a DURA providing random write access to n uniquely valued variables by means of a_i; variables can be tested using b_i. Let the language $\mathcal{L}_n$ be the set of sequences of the form $a_{11}(d_{11}) \cdots a_{n1}(d_{n1})\ a_{i2}(d_{i2}) \cdots a_{ik}(d_{ik}) b_m(d_m)$ where $1 \leq i, j \leq n$, such that $d_{11} \cdots d_{n1}$ are pairwise different, d_{il} is not equal to any $d_{jl'}$ whenever l' is the largest index such that $l' < l$, and such that d_m is equal to $d_{mm'}$ where m' is the largest index of d_m.

We can describe intuitively how this language is accepted by a register automaton: First, a sequence of n data symbols with unique data values is read and stored into n corresponding variables. Then, any number of data symbols of the form $a_{i_1}(d_{i_j})$ is read and stored in the ith variable. The data value d_{i_j} must always be different from any of the data values currently stored. Finally, the data symbol $b_m(d_m)$ is read and d_m is compared to the data value currently stored in the mth variable. If they are equal, the register automaton accepts.

A DURA for $\mathcal{L}_n$ will need n variables to store the n unique data values. The minimal DURA for $\mathcal{L}_n$ has $n + 3$ locations: one initial location followed by a sequence of n locations processing $a_{11}(d_{11}) \ldots a_{n1}(d_{n1})$, and where, after reading each data symbol $a_{ij}(d_{ij})$, a variable x_i is initialized and assigned the data value d_{ij}. In addition to these, the DURA also has one accepting location and a one non-accepting sink.

The minimal DOURA for $\mathcal{L}_n$, on the other hand, has more than $n!$ locations. Since the DOURA must store all data values in order of occurence, a control location is needed for every possible re-ordering of the variables. Otherwise, it will be impossible to correctly classify the final $b_m(d_m)$ symbol. □

7. Conclusions and Future Work

We have presented a novel form of register automata, which also has an intuitive and succinct minimal canonical form that can be derived from a

Nerode-like congruence. Key to this canonical form is the representation of a data language as a classification of a set of data words. We have shown that any set of data words (with some restrictions) can be correctly classified by a minimal subset of data words. We have also formulated a Nerode-like congruence that enables the construction of a succinct and canonical register automaton from such a minimal subset. Finally, we have compared our form of register automata to other proposed formalisms, showing that our register automata can be exponentially more succinct.

As a practical application, we have used the results in this paper to generalize Angluin-style active learning to data languages over infinite alphabets. The learning algorithm, which we describe in [13], can be used to characterize, e.g., protocols, services, and interfaces.

We have also further built on the theoretical concepts, in [9], generalizing our canonical model to more expressive signatures by allowing data values to be compared using binary relations. An interesting direction for future work would be to learn such models. To this end, we plan to extend the current learning algorithm in [13] to be able to handle arbitrary binary relations between data values.

References

[1] R. Alur, D.L. Dill, A theory of timed automata, Theor. Comput. Sci. 126 (1994) 183–235.

[2] D. Angluin, Learning regular sets from queries and counterexamples, Inf. Comput. 75 (1987) 87–106.

[3] M. Benedikt, C. Ley, G. Puppis, What you must remember when processing data words, in: A.H.F. Laender, L.V.S. Lakshmanan (Eds.), AMW, volume 619 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2010.

[4] H. Björklund, T. Schwentick, On notions of regularity for data languages, Theor. Comput. Sci. 411 (2010) 702–715.

[5] M. Bojanczyk, C. David, A. Muscholl, T. Schwentick, L. Segoufin, Two-variable logic on data words, ACM Trans. Comput. Log. 12 (2011) 27.

[6] M. Bojanczyk, B. Klin, S. Lasota, Automata with group actions, in: LICS, IEEE Computer Society, 2011, pp. 355–364.

[7] P. Bouyer, A logical characterization of data languages, Inf. Process. Lett. 84 (2002) 75–85.

[8] S. Cassel, F. Howar, B. Jonsson, M. Merten, B. Steffen, A succinct canonical register automaton model, in: T. Bultan, P.A. Hsiung (Eds.), ATVA, volume 6996 of *Lecture Notes in Computer Science*, Springer, 2011, pp. 366–380.

[9] S. Cassel, F. Howar, B. Jonsson, B. Steffen, A succinct canonical register automaton model for data domains with binary relations, in: ATVA, to appear.

[10] N. Francez, M. Kaminski, An algebraic characterization of deterministic regular languages over infinite alphabets, Theor. Comput. Sci. 306 (2003) 155–175.

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

[12] J.E. Hopcroft, J.D. Ullman, Introduction to Automata Theory, Languages and Computation, Addison-Wesley, 1979.

[13] F. Howar, B. Steffen, B. Jonsson, S. Cassel, Inferring canonical register automata, in: V. Kuncak, A. Rybalchenko (Eds.), VMCAI, volume 7148 of *Lecture Notes in Computer Science*, Springer, 2012, pp. 251–266.

[14] M. Kaminski, N. Francez, Finite-memory automata, Theor. Comput. Sci. 134 (1994) 329–363.

[15] P.C. Kanellakis, S.A. Smolka, Ccs expressions, finite state processes, and three problems of equivalence, Inf. Comput. 86 (1990) 43–68.

[16] R. Lazic, D. Nowak, A unifying approach to data-independence, in: C. Palamidessi (Ed.), CONCUR, volume 1877 of *Lecture Notes in Computer Science*, Springer, 2000, pp. 581–595.

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

[18] R. Paige, R.E. Tarjan, Three partition refinement algorithms, SIAM J. Comput. 16 (1987) 973–989.

[19] A. Petrenko, S. Boroday, R. Groz, Confirming configurations in EFSM testing, IEEE Trans. Software Eng. 30 (2004) 29–42.

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

[21] P. Saint-Andre, Extensible Messaging and Presence Protocol (XMPP): Instant Messaging and Presence, RFC 6121 (Proposed Standard), 2011.

[22] L. Segoufin, Automata and logics for words and trees over an infinite alphabet, in: Z. Ésik (Ed.), CSL, volume 4207 of *Lecture Notes in Computer Science*, Springer, 2006, pp. 41–57.

[23] T. Wilke, Specifying timed state sequences in powerful decidable logics and timed automata, in: H. Langmaack, W.P. de Roever, J. Vytopil (Eds.), FTRTFT, volume 863 of *Lecture Notes in Computer Science*, Springer, 1994, pp. 694–715.

Inferring Canonical Register Automata*

Falk Howar[1], Bernhard Steffen[2], Bengt Jonsson[3], Sofia Cassel[3], and Malte Isberner[2]

[1] Carnegie Mellon University, Silicon Valley
howar@cmu.edu
[2] Technical University Dortmund, Chair for Programming Systems, Dortmund, D-44227, Germany
{bernhard.steffen|malte.isberner}@cs.tu-dortmund.de
[3] Dept. of Information Technology, Uppsala University, Sweden
{bengt.jonsson|sofia.cassel}@it.uu.se

Abstract. In this paper, we present an extension of active automata learning to *Register Automata*, an automaton model which is capable of expressing the influence of data on control flow. Register Automata operate on an infinite data domain, whose values can be assigned to registers and compared for equality. 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 Automaton models.
The application of our algorithm for inferring semantic models of data structures shows the impact of this new method. We evaluated our algorithm on a complex data structure, a "stack of stacks", the largest of which we could learn in merely 20 seconds with less than 4,000 membership queries, resulting in a model with roughly 800 locations. In contrast, even when restricting the data domain to just four values, the corresponding plain Mealy machine would have more than 10^9 states and presumably require billions of membership queries.

1 Introduction

The model-based approach to development, verification, and testing of software systems (e.g., [15,12,26]) is a key path towards efficient development of reliable software systems. However, its application is hampered by the current lack of adequate specifications for most actual systems. The use of component libraries with very partial specifications, and the problem of maintaining specifications of evolving systems aggravate the situation. Automata learning techniques [20] have been proposed to overcome this, by allowing to construct and later update behavioral models automatically. This has been illustrated in a number of case studies like, e.g., the concrete setting of Computer Telephony Integrated (CTI) systems [20], and in protocol specification [33], analysis [41], and testing [43].

* This work is supported by the European FP 7 project CONNECT (IST 231167).

Recently, automata learning has even been investigated as a key enabler to the automated synthesis of mediators for network protocols: The CONNECT project [28] is the development of "Emergent Middleware", i.e., a set of enablers synthesizing mediators between systems automatically at runtime. We discuss the CONNECT project in detail here as it allows us to illustrate the state-of-the art in automata learning and where improvement is needed in order to meet the high expectations in contexts as automated testing or mediator synthesis. The envisioned scenario of the CONNECT is characterized by two important observations.

1. Encountered systems for which models are to be inferred are black-boxes. Only the static interface, i.e., the offered communication primitives are known.
2. The behavior of these (real networked systems) will usually not only depend on the order in which messages are invoked. The data parameters that are used in the communication (e.g., sequence numbers or session identifiers) also influence a system's behavior.

These two observations are not particular to the CONNECT project. Basically, they hold wherever one is interested in generating specifications of real systems dynamically. The first observation may be varied slightly in some cases (e.g., if source code of the analyzed system is available). The second one expresses a requirement and long standing research interest in the realm of dynamic analysis: the combination of control- and data-dependent characterizations of systems.

Current 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 [20,27,4,41], or automata learning techniques (e.g., [6,37]). Another class generates invariants over state variables [17] or exchanged data values by generalizing from concrete observations. For many applications in testing and verification, and also in commercial model-based testing tools (e.g., ConformiQ Qtronic [26]), it is, however, important to generate models that capture combined behavior of control and data. Parameters such as sequence numbers, identifiers, etc. have a significant impact on control flow in typical protocols. For instance, a valid sequence number or session identifier has a very different influence on subsequent behavior than an invalid one.

In particular *active* automata learning has been applied successfully in a number of interesting case studies (cf. Section 8 for a detailed discussion). The state-of-the-art approach to inferring models of real (i.e., infinite state) systems in active automata learning is using predefined abstractions, realized by so-called *mappers* (cf. [30]), which are placed between the component to be inferred and the learning algorithm. The aim of a mapper is to provide a regular projection of the component. This approach is described explicitly for the generation of specifications from protocol entities in [1]. However, defining mappers is an error-prone and laborious manual effort.

In [25] automated alphabet abstraction refinement is integrated with active learning to overcome this problem. A similar approach, using lazy alphabet construction without counterexample driven alphabet abstraction refinement, is

presented in [18] for the inference of assumptions in assume-guarantee reasoning. A major drawback of current abstraction-based approaches is that the inferred models do not reflect the causal influence of data values appropriately. The models remain representations of some concrete traces of a system for which a regular projection exists. For example, the models can express that a certain data values has to be bigger than some constant threshold but they cannot express that a data value has to be bigger (or somehow else causally related) to a prior data value in a trace.

For both classes of approaches, mapper-based and abstraction-based, the problem is that the influence of data values is encoded into the alphabet symbols of the underlying automaton formalism. Our approach, on the other hand, will make data values first class citizens in control models: In this paper, we present an extension of active automata learning to *register automata*, an automaton model which is capable of expressing the influence of data on control flow. Register automata operate on an infinite data domain, whose values can be assigned to registers and compared for equality by very natural mechanisms. This suffices to handle parameters like user names, passwords, identifiers of connections, sessions, etc., in a fashion similar to, and slightly more expressive than, the class of "data-independent" systems, which was the subject of some of the first works on model checking of infinite-state systems [46,29]. Thus register automaton learning is particularly suited for the validation of protocols, connectors or mediators, as we will discuss based on a small fragment of the XMPP protocol (cf. Figure 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. Conceptually, it is based on a generalized Nerode relation and a corresponding canonical form for register automata, which, like in the classical regular case, identifies the required control locations [13]. Algorithmically, the L^*-typical partition refinement process [6] is 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.

2 Organization and Overview

In this section we present a high-level summary of our contribution and align in with well-known ideas of in active automata learning, i.e., with the popular L^*-algorithm [6] in particular. Our main technical contribution is a novel active learning algorithm, which infers a canonical register automaton (RA) for an unknown data language L, of which it initially knows only the set of actions. Canonical register automata will be defined properly in the next section. Active learning in general proceeds by asking two kinds of queries.

– A *membership query* consists in asking if a (data) word w is in L.

– An *equivalence query* consists in asking whether a hypothesized RA $\mathcal{H}$ is correct, i.e., whether $L(\mathcal{H}) = L$. 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 L and $L(\mathcal{H})$.

Key to (classic) L^*-like learning is the well known Nerode congruence [34], which allows to identify words that lead to the same location in a canonical acceptor for some language 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 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 a so-called *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 $uv \in L$, i.e., whether v is in the 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 extended successively, using a sequence of membership queries, until the table satisfies satisfies certain 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 ensure that during the next round of hypothesis construction [37,42] more states can be found. During learning, hypothesis automata will grow monotonically in size, until they have the size of the canonical acceptor for 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 RA for some data language L. The theoretical backbone will be how to

– identify data values to be stored in variables by an automaton,
– identify representative words from which the logical formulas guarding transitions can be derived, and
– identify locations using an extended Nerode relation including relations between data values.

This will be presented in Section 3, which basically revisits our results from [14], emphasizing the parts that are relevant to learning. In Section 4 we present an active learning algorithm for register automata based on these ideas. In essence, the overall pattern of learning RA is a three-dimensional maximum fix-point computation, determining (a) the locations, (b) the required variable assignments, and (c) the guarded transitions in a partition-refinement fashion. Following the pattern of L^*, we will show in detail

– how so-called abstract suffixes can be used to approximate the Nerode congruence,
– how an observation table can be realized,
– how at certain points well-defined hypothesis automata can be constructed from the observation table, and
– how counterexamples can be exploited to guarantee strictly monotone progress.

Finally, 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 RA for L. Subsequently, we discuss an application example in Section 5.

We present our new learning algorithm for register automata, accepting so-called data languages. It is not hard to extend the ideas to register Mealy machines, i.e., systems with outputs and (already known) data values in outputs. We will briefly discuss the necessary modifications in Section 6. Then, we will evaluate both approaches (inferring register automata and register Mealy machines) on a set of data structures in Section 7. Related work is discussed in Section 8 and we conclude in Section 9.

3 Canonical Register Automata

This section introduces Register Automata, the class of data languages they accept and discusses how the canonical automaton can be generated for a data language. This construction relies on several concepts, all of which will later be used by the learning algorithm as well. The only difference is that, while in this section we will generally work with infinite sets of words or the the set of all words, the learning algorithm will always work with finite sets.

3.1 Register Automata

We assume an unbounded domain D of data values. In the examples presented in this paper we let D be the set $\mathbb{N}$ of natural numbers. We also assume an arbitrary strict total order $\prec$ on D. This order has the only purpose of allowing us to ease presentation. We use $<$ on $\mathbb{N}$ as strict total order in our examples.

Let Σ be a finite set of *actions*. Each $a \in \Sigma$ has a fixed arity (i.e, takes a fixed number of data values). We write Σ^D for the set of all possible combinations $a(d_1, \ldots, d_k)$ of actions and data values (respecting the arities of actions). Thus, $(\Sigma^D)^*$ denotes the set of *data words*. For a data word w, let $Acts(w)$ denote its sequence of actions, and $Vals(w)$ the sequence of data values occurring in w, while $ValSet(w)$ is the set of data values that occur in w. A *data language* L is a set of data words.

We use the set $P = \{p_i \ : \ i \in \mathbb{N}\}$ as *formal parameters*, and the set $\mathcal{X} = \{x_i \ : \ i \in \mathbb{N}\}$ to denote *variables*. A *parameterized action* is a pair $a(\bar{p})$, of an action a of arity k and a sequence of k formal parameters $\bar{p} = p_1, \ldots, p_k$. A

guard is a boolean combination of atomic predicates over formal parameters and variables, generated by the grammar

$$G ::= G \wedge G \mid G \vee G \mid p_i \otimes z \mid \mathit{true},$$

where $\otimes \in \{=, \neq\}$, where $z \in \mathcal{X} \cup P$, and where true denotes the atomic predicate that is always satisfied. Intuitively, a guard can compare parameters with other parameters, or variables, for (in)equality and combine these tests using boolean connectives.

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

- Σ *is a finite set of actions,*
- 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.*
- $\lambda : L \to \{+, -\}$ *is the* output function
- Γ *is a finite set of* transitions *of form* $\langle l,\ a(\bar{p}),\ \bar{g},\ \rho,\ l' \rangle$*, where*
  - l *and* l' *are source and target locations,*
  - $a(\bar{p})$ *is a parameterized action,*
  - $\bar{g}$ *is a guard, and*
  - $\rho :\ X(l') \to (X(l) \cup \bar{p})$ *is an assignment.*

Informally, an RA is well-formed if guards only use defined variables and parameters. Formally, we require that a guard of an $a(\bar{p})$-transition with source l has predicates over $X(l)$ and $\bar{p}$ only. We only consider well-formed automata.
Let us now define the semantics of an RA $\mathcal{A} = (\Sigma, L, l_0, X, \lambda, \Gamma)$, which will be used to define the data language $L(\mathcal{A}) \subseteq (\Sigma^D)^*$ accepted by $\mathcal{A}$. A *valuation* 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 of $X(l)$. The *initial state* is the pair $\langle l_0, \nu_0 \rangle$ of initial location and valuation ν_0 with empty domain of definition.

A *step* of $\mathcal{A}$, denoted by $\langle l, \nu \rangle \xrightarrow{a(\bar{d})} \langle l', \nu' \rangle$, transfers $\mathcal{A}$ from $\langle l, \nu \rangle$ to $\langle l', \nu' \rangle$ on input $a(\bar{d})$ with data values from D if there is a transition $\langle l,\ a(\bar{p}),\ \bar{g},\ \rho,\ l' \rangle \in \Gamma$ such that

1. $\nu \models \bar{g}[\bar{d}/\bar{p}]$, i.e., ν makes $\bar{g}$ true when replacing $p_i \in \bar{p}$ by $d_i \in \bar{d}$.
2. ν' is the updated valuation with

$$\nu'(x_i) = \begin{cases} \nu(x_j) & \text{if } \rho(x_i) = x_j \\ d_j & \text{if } \rho(x_i) = p_j. \end{cases}$$

A *run* of $\mathcal{A}$ over a data word $a_1(\bar{d}_1) \dots a_k(\bar{d}_k)$ is a sequence of steps

$$\langle l_0, \nu_0 \rangle \xrightarrow{a_1(\bar{d}_1)} \langle l_1, \nu_1 \rangle \quad \dots \quad \langle l_{k-1}, \nu_{k-1} \rangle \xrightarrow{a_k(\bar{d}_k)} \langle l_k, \nu_k \rangle.$$

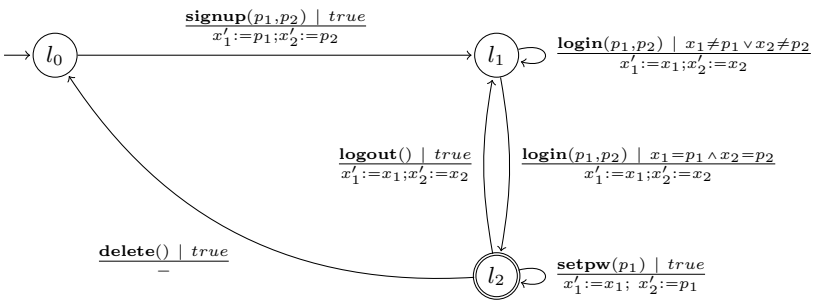


Fig. 1. RA for the authentication fragment of a protocol (from [13]). We write $x'_i := z$ in assignments for $\rho(x_i) = z$. The accepting location l_2 is marked with a double circle. Some reflexive transitions are omitted for better readability.

An RA $\mathcal{A}$ is *input-enabled* if every word in $(\Sigma^D)^*$ has a run of $\mathcal{A}$. It is deterministic if no data word in $(\Sigma^D)^*$ has more than one run of $\mathcal{A}$. We will restrict our attention to input-enabled, deterministic RAs for the remainder of this paper. Finally, a data word w is in the data language $L(\mathcal{A})$ of $\mathcal{A}$ if and only if the run of $\mathcal{A}$ over w ends in a state $\langle l, \nu \rangle$ with an accepting location, i.e., where $\lambda(l) = +$.

Example 1. Figure 1 shows an example of a Register Automaton. The RA describes a service that allows a user **signup** for an account providing her name as parameter p_1 and the desired password as p_2. In the figure, this corresponds to the transition from l_0 to l_1. Once registered, a user may **login** to the system providing name and password as parameters. The system allows an unbounded number of attempts to log in. Finally, it is possible to **logout**, **delete** the account, or change the password (**setpw**) when logged in.

3.2 Data languages

So far, we have shown how an RA processes data inputs and accepts data words. When inferring RA models, the learning algorithm can only perform tests on the system under learning (SUL), i.e., test the language of the SUL and not its structure. In this section we will revisit some of the ideas from [13] for a concise representation of data languages which later will provide the basis for constructing a canonical register automaton for a data language L.

Let us start with some helpful properties of data words and data languages. For a data word uv let u be a *prefix*. Then uv is called a *flat* extension of u if the data values in v are pair-wisely different and if $ValSet(u) \cap ValSet(v) = \varnothing$. In our example, $\mathbf{signup}(1,2)$ is a prefix of $\mathbf{signup}(1,2)\mathbf{login}(1,2)$, while the data word $\mathbf{signup}(1,2)\mathbf{login}(1,2)\mathbf{login}(3,4)$ is a flat extension of the word $\mathbf{signup}(1,2)\mathbf{login}(1,2)$.

We define a replacement operation on the data values of data words: for a permutation $\pi : D \to D$ let $\pi(w)$ be the data word that results from replacing

all data values in w according to π. Sometimes we will apply permutations to sets of data words, i.e., apply them point-wise to each data word in the set.

For $w, w' \in (\Sigma^D)^*$, let $w \simeq w'$ denote that there is a permutation π on D such that $\pi(w) = w'$. A *data language* L is a set of data words, such that each equivalence class w.r.t. $\simeq$ is either completely inside or completely outside L. It is not difficult to see that the language $L(\mathcal{A})$ of an RA $\mathcal{A}$ is always a data language in this sense. In our running example, e.g.,

$$\mathbf{signup}(1,2)\mathbf{login}(1,2) \;\simeq\; \mathbf{signup}(5,3)\mathbf{login}(5,3)$$

are both in the language of the RA of Figure 1.

Closedness of data languages under permutations on D allows us to reason about classes of words instead of individual data words. However, since classes of words are difficult to handle in presentation, we select one representative data word for every class. A data word w is *representative* if the data values that appear in any prefix of $Vals(w)$ is of form $\{1, 2, \ldots, k\}$ for some k. In the above example, $\mathbf{signup}(1,2)\mathbf{login}(1,2)$ is a representative data word, but $\mathbf{signup}(1,3)\mathbf{login}(2,3)$ is not. We use $\mathcal{R} \subset (\Sigma^D)^*$ to denote the set of representative data words.

We need to relate transitions (i.e., mainly guards) of register automata and data words. In a register automaton many different data words can have the same sequence of transitions as transitions test only important equalities and inequalities. We will characterize data words by logical formulas, which will allow to group data words in a similar fashion, using only the important equalities. Before providing formal definitions, let us provide some intuition.

We characterize words by formulas describing only the equalities in these words and not inequalities, making data words with no equal data values the default, or *true*-case and all other data words refinements or special cases. E.g., the data word $\mathbf{signup}(1,2)\mathbf{signup}(3,4)$ is $true$, while $\mathbf{signup}(1,2)\mathbf{signup}(1,3)$ is described by $d_1 = d_3$, and $\mathbf{signup}(1,2)\mathbf{signup}(1,2)$ is described by $d_1 = d_3 \wedge d_2 = d_4$, where we name $Vals(w)$ by $d_1, \ldots, d_k$. Selecting a default case is an arbitrary decision – we could as well start from the case where all data values are identical.

The important property of this construction is that for a fixed sequence of actions the formulas describing data words form a lattice under implication. We want to use this implication relation on formulas on the set of data words and single out "least constrained" data words to summarize more constrained data words wherever possible. This allows us to use only important tests when constructing automata from data words. However, we have to be careful to respect the order in which an automaton can perform the tests describing a word (i.e., from left to right).

More formally, we use two ordering relations on $\mathcal{R}$. For $w, w' \in \mathcal{R}$ with $Acts(w) = Acts(w')$ let

- $w \sqsubseteq w'$ if for $d_i, d_j \in Vals(w)$ and $d'_i, d'_j \in Vals(w')$ we have $d'_i = d'_j$ whenever $d_i = d_j$. Intuitively, w' has the same (and maybe more) equalities between

Algorithm 1 Computing E_λ^S

Require: A set S of data words, a function $\lambda : S \to \{+, -\}$
Ensure: The set E_λ^S of λ-essential words for S

```
1: E_λ^S := ∅
2: for w ∈ S in <-order do
3:     if |max_<{(E_λ^S)^⊏w}| ≠ 1 then
4:         E_λ^S := E_λ^S ∪ {w}
5:     else
6:         w' := max_<{(E_λ^S)^⊏w}
7:         if λ(w') ≠ λ(w) then
8:             E_λ^S := E_λ^S ∪ {w}
9:         end if
10:    end if
11: end for
12: close E_λ^S under prefixes
13: close E_λ^S under flat extensions
14: return E_λ
```

its data values than w has. Let $\sqsubset$ denote the corresponding strict order. In our example

$$\mathbf{signup}(1,2)\mathbf{login}(3,4) \;\sqsubset\; \mathbf{signup}(1,2)\mathbf{login}(1,2).$$

– $w < w'$ if there are prefixes u of w and u' of w' such that $u \sqsubset u'$. Intuitively, we use $<$ to model the branching behavior of an automaton on a set of words. In the example, e.g.,

$$\mathbf{signup}(1,2)\mathbf{login}(1,2) \;<\; \mathbf{signup}(1,1)\mathbf{login}(2,2).$$

We use these two orders to define the set of *L-essential* data words for some set S of data words. For a mapping $\lambda : S \to \{+, -\}$ with domain S, an λ-*cover* of S is a set $E_\lambda^S \subseteq S$ such that for every $w \in S$, the set $\{v \in E_\lambda^S \;:\; v \sqsubseteq w\}$ contains a unique maximal (w.r.t. $<$) word w', and $\lambda(w) = \lambda(w')$.

Algorithm 1 shows how we compute the canonical λ-cover E_λ^S for any mapping $\lambda : S \to \{+, -\}$. For a data word $w \in \mathcal{R}$, let $\mathcal{R}^{\sqsubset w}$ be the set $\{w' \in \mathcal{R} \;:\; w' \sqsubset w\}$. The canonical λ-cover is well-founded and contains at least all the minimal data words (w.r.t. $\sqsubseteq$) from S (lines 3-4). We then add words in ascending $<$-order if they don not have a unique $<$-maximal $\sqsubseteq$-smaller word in E_λ^S (lines 3-4) or if they disagree in λ with this word (lines 7-8). Finally, we close the set under its prefixes and flat extensions, which will later become important, when constructing automata from E_λ^S. The canonical construction yields the following proposition, which will be useful for reasoning about counterexamples in Section 4.3.

Proposition 1. *For two functions f, g with domain S, with canonical covers E_f^S and E_g^S, and $w \in S$ with $f(w) \neq g(w)$*

$$w \notin E_f^S \ \wedge \ w \notin E_g^S \quad \Rightarrow \quad \exists w' \in S^{\sqsubset w} \ . \ f(w') \neq g(w').$$

We can use the λ-cover of $\mathcal{R}$ to classify a data language L concisely: Every word w is represented faithfully (w.r.t. λ) by the $<$-maximal $\sqsubseteq$-smaller word in $E_\lambda^\mathcal{R}$. We thus refer to $E_\lambda^\mathcal{R}$ as the set of L-essential words.

3.3 Canonical Register Automata

To actually construct a canonical RA $\mathcal{A}_L$ for L from the set of L-essential data words, we need to derive three things from these words. First, we need to determine which data values of words are to be stored in variables. Second, we need to derive a set of locations. Third, we have to construct transitions from words. A proof showing that indeed $L(\mathcal{A}_L) = L$ can be found in [14].

Memorable data values. Intuitively, a data value d in a data word $u \in E_\lambda^\mathcal{R}$ must be remembered if there is an extension uv of u such that a comparison between d and some value(s) in v is needed to determine whether $uv \in L$. Each such "memorable" data value must be stored in registers in $\mathcal{A}_L$. Let us make a formal definition

Definition 2. Let $\pi_{d,d'} : D \to D$ be the permutation which swaps d and d' and leaves all other data values untouched. A data value $d \in ValSet(u)$ is *memorable* in the word $u \in E_\lambda^\mathcal{R}$ (w.r.t. L) if for some suffix $v \in (\Sigma^D)^*$ and some permutation of form $\pi_{d,d'}$, where $d' \notin ValSet(uv)$,

$$uv \in L \ \Leftrightarrow \ u\pi_{d,d'}(v) \notin L. \tag{1}$$

The set of memorable data values w.r.t. L in u is denoted by $mem_L(u)$. □

In our running example 1 and 2 are memorable in $\mathbf{signup}(1,2)$ since the word $\mathbf{signup}(1,2)\mathbf{login}(1,2)$ is in L while replacing 1 (or 2) by 3 in $\mathbf{login}(1,2)$ will result in words not in L, e.g., the word $\mathbf{signup}(1,2)\mathbf{login}(3,2)$.

Locations. The set of locations will be generated by first defining a Nerode equivalence on the set $E_\lambda^\mathcal{R}$ of L-essential words, and thereafter letting locations correspond to equivalence classes of this equivalence. For a word u let

$$u^{-1}E_\lambda^\mathcal{R} = \{v \in (\Sigma^D)^* \ : \ uv \in E_\lambda^\mathcal{R}\}. \tag{2}$$

The set $u^{-1}E_\lambda^\mathcal{R}$ is called the set of *L-essential u-suffixes*. We define a Nerode-equivalence $\equiv_L$ on $E_\lambda^\mathcal{R}$ as follows.

Definition 3. *Let u and u' in $E_\lambda^\mathcal{R}$. Then $u \equiv_L u'$ iff there is a permutation π on D such that*

$$\pi(u^{-1}E_\lambda^\mathcal{R}) \ = \ u'^{-1}E_\lambda^\mathcal{R} \quad \wedge \quad \lambda(uv) = \lambda(u'\pi(v)) \ \ \textit{for} \ \ v \in u^{-1}E_\lambda^\mathcal{R}. \tag{3}$$

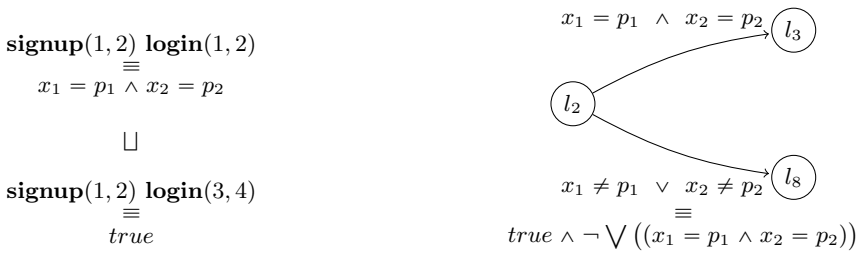


Fig. 2. Poset of essential words and corresponding positive guards (left) and resulting transition guards in an RA (right) for the **login**-transitions of the RA from Figure 1 after L-essential word **signup**$(1, 2)$. It is assumed that 1 is stored in x_1 and 2 is stored in x_2

We say that data language $\equiv_L$ is *regular* if $\equiv_L$ has finite index. We let $[u]_L$ denote the equivalence class of the word u. For every class of $\equiv_L$, we fix one particular member of the equivalence class, and call it the *access string* of this equivalence class. Thus, if u is an access string, then $[u]_L$ is a location, which we sometimes denote l_u. The variables $X(l_u)$ of location l_u are the set of memorable data values of u and $\nu_u : X(l_u) \to mem_L(u)$ describe, how memorable data values of u are stored in $X(l_u)$.

The location l_ϵ with ϵ as access string is the initial location l_0. A location l_w will be accepting, i.e., $\lambda(l_w) = +$, *iff* $w \in L$.

Transitions. Let l_u be a location with access string u. We now describe how to construct the set of a-transitions for $a \in \Sigma$ from the location l_u. For each L-essential word of form $ua(\bar{d})$ there is a transition to the location $[ua(\bar{d})]_L$. The guard g of this transition is the conjunction of equalities, containing

- $p_i = p_j$ if $d_i = d_j$ for $d_i, d_j \in \bar{d}$, and
- $x_i = p_j$ if $\rho_u(x_i) = d_j$ for h $x_i \in X_u$ and $d_j \in \bar{d}$

Redundant predicates can be removed using a canonic representation. Please observe that the partial order $\sqsubseteq$ of A is preserved on the set of corresponding positive guards $G_{u,a} = \{g_{u,a}(\bar{d}) \ : \ ua(\bar{d}) \in A\}$. On $G_{u,a}$ the partial order $\sqsubseteq$ coincides with implication and we use it accordingly.

Every conjunction of binary equalities in $G_{u,a}$ may imply some other guards in this set (namely the ones in $G_{u,a}^{\sqsupseteq \bar{g}}$). In order to get deterministic transitions, we exclude these implied guards in the final guard of the transition. Let thus $\hat{g} = \bar{g} \wedge \neg\big(\bigvee G_{u,a}^{\sqsupseteq \bar{g}}\big)$ be the final guard for a positive guard $\bar{g}$.

Figure 2 shows the correspondence between L-essential words and guards for two cases from our example (in the left). It is assumed that after prefix **signup**$(1, 2)$ the memorable data values 1 and 2 are stored in x_1 and x_2. Then, e.g., for **login**$(1, 2)$ the corresponding positive guard $x_1 = p_1 \wedge x_2 = p_2$ describes exactly the equalities between data values of the prefix and of **login**. In the right of the figure it is shown how these positive guards are extended to their final

versions in an automaton. The two guards in this example will become part of the **login**-transitions of the RA shown in Figure 1.

Using the mapping $f : E^{\mathcal{R}}_{\lambda} \to E^{\mathcal{R}}_{\lambda}/\equiv_L$ from the previous step we define a transition $\langle l_{[u]_L},\ a(\bar{p}),\ \hat{g},\ \rho,\ l_{f(ua(\bar{d}))}\rangle$ for every $\bar{g} \in G_{u,a}$. The only thing we did not fix yet is the assignment: let w be the fixed representative data word for $l_{f(ua(\bar{d}))}$ and let $ua(\bar{d})$ with $g_{u,a}(\bar{d}) = \bar{g}$ such that $w^{-1}E^{\mathcal{R}}_{\lambda} = \pi(ua(\bar{d})^{-1}E^{\mathcal{R}}_{\lambda})$ and $\lambda(w\pi(v)) = \lambda(ua(\bar{d})v)$ for $v \in ua(\bar{d})^{-1}E^{\mathcal{R}}_{\lambda}$. Then, let $\nu_{ua(\bar{d})} = \pi \circ \nu_w$ and

$$\rho(x'_i) \;=\; \begin{cases} p_j & \text{if } \nu_{ua(\bar{d})}(x'_i) = d_j \\ x_j & \text{if } \nu_{ua(\bar{d})}(x'_i) = \nu_u(x_j). \end{cases}$$

Intuitively, ρ is used to create the updated valuation $\nu_{ua(\bar{d})}$ from the old valuation ν_u and the set of parameters (and data values) of $a(\bar{p})$ and $a(\bar{d})$. The desired resulting updated valuation $\nu_{ua(\bar{d})}$ is specified relative to the representative data word w of $[ua(\bar{d})]_L$ so that the data values in $ua(\bar{d})$ are stored in an order that is compatible with the transitions from l_w.

Automata construction. The RA $\mathcal{A}_L = (\Sigma, L, l_0, X, \lambda, \Gamma)$ is well-defined: The set of actions Σ is fixed by L. The set of locations is determined by the classes of $\equiv_L$ The empty word ε is in one of these classes. Hence there is one dedicated initial location l_0. The set of variables is finite as we selected finite words from every class of $\equiv_L$ for which we computed the finitely many memorable data values as a basis for the set of variables. Not all of these variables are defined at every location but the particular construction of transitions guarantees that at no time undefined variables are read (i.e., occur in the co-domain of an assignment or in a guard). The relation between guards and essential words makes the automaton input-enabled and deterministic. Finally, λ is fixed in accordance to L for the fixed L-essential words.

4 Active Learning for Register Automata

In this section we present our main technical result, an L^*-like active learning algorithm for register automata.

4.1 Observation tables

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.

In Section 3.3, we have used sets of L-essential suffixes to define $\equiv_L$. Unfortunately these sets where infinite. We will show that the ideas can be transferred to finite sets of suffixes easily. Let $V \subset \Sigma^*$ a finite set of *abstract suffixes*. An abstract suffix $\overline{v}$ is just a sequence of actions without data values.

For a set of abstract suffixes V, let $V(u)$ be a subset of the L-essential u-suffixes. Concretely, let uV be the set $(\{u\} \times \{v \ : \ V^D\}) \cap \mathcal{R}$ and E_λ^{uV} its canonical λ-cover. Then, $V(u) \ = \ \{v \in V^D \ : \ uv \in E_\lambda^{uV}\}$.

We can compute the $V(u)$ and the corresponding fragment of λ using membership queries: Let the *u-closure of V* be the mapping $C_u^{V(u)} : V(u) \to \{+, -\}$ with

$$C_u^{V(u)}(v) \ = \ \lambda(uv).$$

Since in the definition of memorable data values (Eq. 1) only data values are exchanged in a suffix, and since $C_u^{V(u)}$ covers λ faithfully on $\{u\} \times V^D$, we can use $C_u^{V(u)}(v)$ to compute the set $mem_V(u)$ of data values from u that are proven to be memorable by V.

We adapt the Nerode congruence from Eq. 3 and write $u \equiv_V u'$ if for two L-essential words u and u' there is a permutation π on D for which

$$\pi(V(u)) \ = \ V(u') \quad \wedge \quad C_u^{V(u)}(v) = C_{u'}^{V(u)}(\pi(v)) \text{ for } \ v \in V(u).$$

We immediately get the following two properties, which allow us to use closures as the basis for our observation tables and will become part of the invariant promised at the beginning of this section, showing that the learning algorithm terminates with an optimal result.

Proposition 2. *For all sets V of abstract suffixes $mem_V(u) \subseteq mem_L(u)$.*

Proposition 3. *If $u \equiv_L u'$ then $C_u^{V(u)} \equiv_V C_{u'}^{V(u')}$ for all sets of abstract suffixes V.*

We have shown that $\equiv_V$ will never refine $\equiv_L$ and that we will never falsely identify memorable data values using closures.

In order to prove that the learning algorithm can progress, we have to show that there is a finite set of abstract suffixes V for which $u \not\equiv_L u' \ \Rightarrow \ u \not\equiv_V u'$ and $mem_L(u) \subseteq mem_V(u)$. This, however, is trivial. Since $\equiv_L$ has finite index n, the set V only has to generate all suffixes up to length n in the worst case.

Definition 4 (Observation table). An observation table is a tuple $\langle U, V, T \rangle$, of a prefix-closed set of 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 $U \subset E_\lambda^{\mathcal{R}}$ consists of a prefixed-closed subset $\hat{U}$ of *short prefixes*, and contains for every prefix $u \in \hat{U}$ at least the one-action L-essential extension $ua(\bar{d})$ where data values in $a(\bar{d})$ 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 $(\{u\} \times V^D) \cap \mathcal{R}$ as described above. Our algorithm will initialize $\hat{U} = V = \{\epsilon\}$, and maintain the invariant that $T(u) \not\approx T(u')$ for $u, u' \in \hat{U}$.

An intermediate observation table for our example is shown in the left of Figure 7. The left column contains prefixes. Prefixes from $\hat{U}$ are shown in the upper part of the table. The two 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.

	ϵ	$\mathbf{login}(p_1, p_2)$
ϵ (l_0)	-	$\mathbf{login}(1,2)$ -
$\mathbf{signup}(1,2)$ (l_1)	-	$\mathbf{login}(1,2)$ + $\mathbf{login}(3,4)$ -
$\mathbf{delete}()$	-	$\mathbf{login}(1,2)$ -
$\mathbf{signup}(1,2)\mathbf{login}(3,4)$	-	$\mathbf{login}(1,2)$ + $\mathbf{login}(5,6)$ -

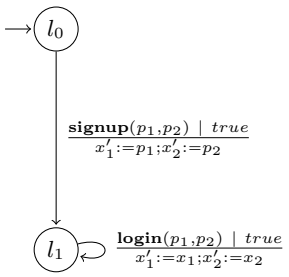


Fig. 3. Intermediate Observation Table (only showing a subset of all prefixes) for Example from Figure 1 and corresponding hypothesis.

4.2 Constructing hypothesis automata

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

Definition 5 (Closedness). *An observation table* $\langle U, V, T\rangle$ *is* closed *if for every prefix* $u \in U$ *there is a prefix* $u' \in \hat{U}$ *such that* $T(u) \equiv_V T(u')$.

Please note that in general there can be multiple effective permutations satisfying the condition from Eq. 3. This can be due to true symmetry of parameters (a special case that does no harm an which we do not cover here in detail), but also to the approximative nature of intermediate results in learning.

However, since the existence of effective permutations is transitive, there can never be two permutations proving the same word from $(U \backslash \hat{U})$ equivalent to different words from $\hat{U}$. The prefixes in $\hat{U}$ will become the locations of a hypothesis automaton. Closedness then ensures that all transitions of the hypothesis, defined by prefixes in U, have well-defined destinations.

Definition 6 (Register-consistency). *An observation table* $\langle U, V, T\rangle$ *is* register-consistent *if for every prefix* $ua(\bar{p}) \in U$

$$mem_V(\ ua(\bar{p})\ )\ \cap\ ValSet(u)\ \subseteq\ mem_V(u).$$

When constructing a hypothesis automaton from an observation 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 Section 3.3. The key idea is that 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 L-essential words in U directly, and λ will be defined using values from the closures. More formally, we construct $\mathcal{H} = (\Sigma, L, l_0, X, \lambda, \Gamma)$ as follows.

- Prefixes u in $\hat{U}$ become locations l_u in $\mathcal{H}$.
- Location l_ϵ is the initial location.
- The set of variables is defined implicitly when constructing transitions.
- λ can be defined from the observation table since V contains the empty words ϵ.
- Transitions $\langle lu,\ a(\bar{p}),\ \bar{g},\ \rho,\ l' \rangle$ in the hypothesis are defined from L-essential prefixes $(\{u\} \times a^D) \cap U$ as described in Section 3.3.

The constructed automaton is well-defined. The relation $\equiv_V$ has a finite index. The empty word ϵ is in the set of short prefixes. Since we enforce that for every prefix $u \in \hat{U}$ at least the continuation $ua(\bar{d})$ with no additional equalities between data values in u and $a(\bar{d})$ is contained in U the transitions will be well-defined. This yields the following proposition.

Proposition 4. *From a closed and register-consistent observation table $\langle U, V, T \rangle$ a well-defined hypothesis automaton $\mathcal{H}$ can be constructed, for which $\lambda_{\mathcal{H}}(u) = \lambda_L(u)$ for $u \in U$.*

Figure 3 shows an observation table and the corresponding hypothesis (both shown only partially). Prefixes in the upper part of the table become locations. The abstract suffix ϵ determines that all locations are rejecting. From the second abstract suffix, the memorable data values of prefix **signup**$(1, 2)$ can be identified. In the hypothesis these are thus stored in registers along the transition from l_0 to l_1.

4.3 Analyzing counterexamples

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 c from the symmetric difference of L and $L(\mathcal{H})$. We will process c from left to right in order to localize where precisely hypothesis and target system behave differently.

Starting with c, we will iteratively generate derived counterexamples, towards the word from $\hat{U}$ that leads to the same location in $\mathcal{H}$ as c. We refer to this word as the *access sequence* of c and denote it by $as_{\mathcal{H}}(c)$. Key idea is that, since $c \in L \ \Leftrightarrow\ as_{\mathcal{H}}(c) \notin L$, words generated in the process will at some point stop being counterexamples (cf. [37,42]).

Technically, we will decompose the current counterexample c in every single step into three parts

- a prefix u from $\hat{U}$,
- a middle part $a(\bar{d})$ consisting only of one symbol, and
- a suffix v.

We begin with the triple where u is the empty word ϵ, and $a(\bar{d})v = c$. We now proceed in three steps, which will be iterated until we find a concrete discrepancy between $\mathcal{H}$ and the (unknown) canonical acceptor $\mathcal{A}_L$.

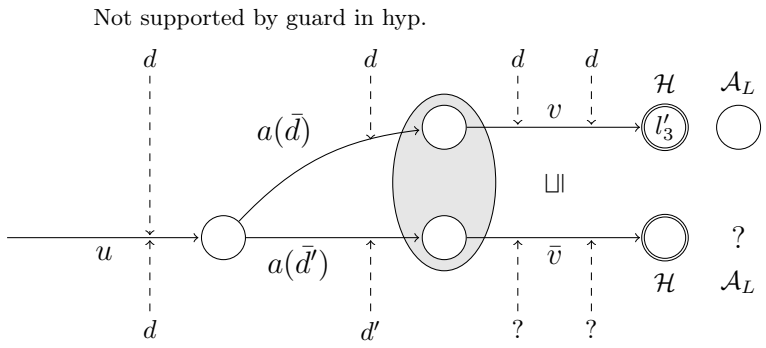


Fig. 4. Counterexamples, new transition.

Step A: New transition Let $ua(\bar{d})v$ such that u is L-essential and a short prefix of the observation table. For $u = \epsilon$ this is the case. For all other prefixes we analyze in this step this will be ensured, too. Intuitively, in this first step we check if $ua(\bar{d})$ is a L-essential prefix that is not yet a prefix in the observation table. In this case, we can add it to the set of prefixes and get a new transition in subsequent hypothesis automata. If $ua(\bar{d})$ is not L-essential we would like to find another counterexample $ua(\bar{d}')v'$ where $ua(\bar{d}')$ is already a prefix in the observation table. In case $ua(\bar{d})$ is already a prefix in the table itself, we can of course skip this step and continue with the next one.

Technically, let $ua(\bar{d}') \sqsubset ua(\bar{d})$ be the unique element of $max_<\{U^{\sqsubseteq ua(\bar{d})}\}$. Ideally, we want to exchange $ua(\bar{d})$ by $ua(\bar{d}')$ in the counterexample. However, there may be data values in the suffix v which we have to refine. Figure 4 shows this situation schematically.

We assume that $ua(\bar{d})$ is not L-essential, which has two implications. Let $\overline{v} = Acts(v)$.

1. by definition of L-essential there are no words $a(\bar{d})v$ in the domain of $C_u^{\overline{v}(u)}$, and
2. by Proposition 1 there is a counterexample in $ua(\bar{d}')v' \in \{u\} \times (W^D)^{\sqsubset a(\bar{d})v}$ and it is a representative data word from $\mathcal{R}$ using a λ_L-cover and a $\lambda_{L(\mathcal{H})}$-cover of $\{u\} \times a^D$.

For a $m = |Vals(a(\bar{d})v)|$ there are less than m^m candidate counterexamples.

In case none of these is a counterexample, we know that $ua(\bar{d})$ has to be L-essential. This follows from Proposition 1 and the fact that $ua(\bar{d})$ is not $L(\mathcal{H})$-essential. In the former case, we add $ua(\bar{d})$ to U and stop analyzing the counterexample. In case we find a new counterexample $ua(\bar{d}')v'$ with $ua(\bar{d}') \sqsubset ua(\bar{d})$, we repeat this step until $ua(\bar{d}') \in U$.

Step B: New memorable data values The previous step ensured that we can continue with a counterexample $ua(\bar{d})v$ where the prefix $ua(\bar{d})$ is L-essential and

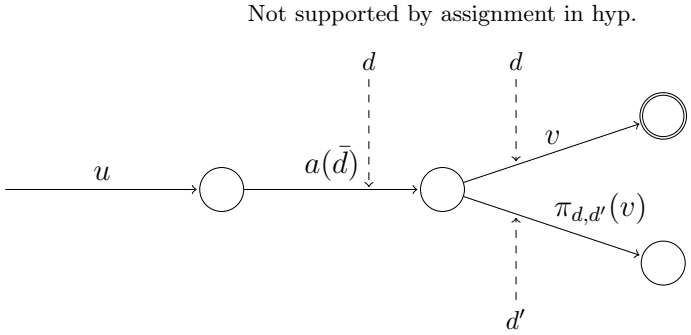


Fig. 5. Counterexamples, new memorable data value

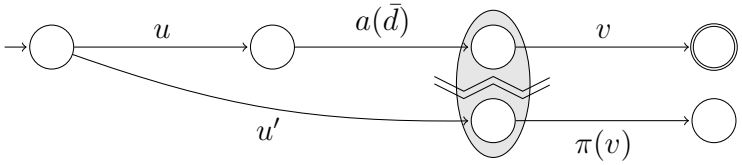


Fig. 6. Counterexamples, new permutation or new location.

in the set of prefixes of the observation table. Some data values in v may equal data values in the prefix. In this step we use the test for memorable data values (cf. Eq. 1) to determine if these equalities are accidental or if they uncover new memorable data values in $ua(\bar{d})$.

As shown in Figure 5, it may be that v in $ua(\bar{d})v$ "uses" data values of $ua(\bar{d})$ which are not stored in variables in $\mathcal{H}$ after processing the prefix since they are not (yet) in $mem_V(ua(\bar{d}))$.

For d in $ValSet(ua(\bar{d})) \cap ValSet(v)$ and not in $mem_V(ua(\bar{d}))$ we get v' by replacing d in v by $d' \notin ValSet(ua(\bar{d})v)$. If (w.l.o.g) $ua(\bar{d})v \in L$ but $ua(\bar{d})v' \notin L$, which can be determined using a membership query, the suffix v proves d to be memorable in ua. In this case, we add $Acts(v)$ to the set of suffixes and can stop processing the counterexample and go back to completing the observation table.

Otherwise we have a new counterexample $ua(\bar{d})v'$ and can repeat the above procedure until $ValSet(ua(\bar{d})) \cap ValSet(v)$ is a subset of $mem_V(ua(\bar{d}))$. For $a(\bar{d})$ of arity k this is the case after at most k iterations and as many membership queries.

Step C: New location Finally, let u' be the short prefix of $ua(\bar{d})$ in the observation table. Since $ValSet(ua(\bar{d})) \cap ValSet(v)$ is a subset $mem_V(ua(\bar{d}))$ we can replace $ua(\bar{d})$ by u' using the permutation π which makes the rows of ua and u' equal. We have used this permutation during hypothesis construction, too. The general

situation is shown in Figure 6. The gray circle indicates one location of the hypothesis.

If (w.l.o.g) $ua(\bar{d})v \in L$ while $u'\pi(v) \notin L$ the suffix v contradicts π. Adding $Acts(v)$ to the set of suffixes will remove π from the set of potential permutations and eventually lead to a new location $ua(\bar{d})$. In this case we can stop here and continue completing the observation table.

Otherwise $u'v$ still is a counterexample for which we know that u' is L-essential and where we can decompose v into $a(\bar{d})v'$ and start over with the first step. In both cases this step requires exactly one membership query.
Since c is a counterexample, at some point one of the three steps will deliver a new prefix or suffix. For a counterexample with m data values, i.e., with $m = |Acts(c)|$ we can estimate the number of membership queries needed to process a counterexample by $O(m^m)$. We thus have:

Proposition 5. *From a counterexample c from the symmetric difference of L and $L(\mathcal{H})$ a prefix u and a suffix v can be derived such that either*

1. *u is in $\hat{U}$ and the abstract suffix $Acts(v)$ witnesses a new memorable data value in u,*
2. *u is in $U \backslash \hat{U}$ and the suffix abstract suffix $Acts(v)$ disproves the permutation used in the table for $T(u) \equiv_V T(u')$ for some $u' \in \hat{U}$*
3. *$u = u'a(\bar{d}) \notin U$, where $u' \in \hat{U}$, the prefix u is a new L-essential word.*

Thus, a counterexample will lead to progress in one of the three dimensions when extending the observation table accordingly.

4.4 The L^*_{RA} algorithm

Put together, this results in Algorithm 2. Lines 1-3 initialize the observation table. The set $\hat{U}$ contains the prefix ϵ, reaching the initial location. The remaining prefixes are one-letter words with no equalities between data values. The set of abstract suffixes is initialized to contain only the empty word.

Hypothesis construction is covered in lines 5-19: First, in line 6 closures are computed as described in Section 4.1. Then, the observation table is checked for closedness (lines 7-11) and for register consistency (lines 12-17). This is repeated until a hypothesis can be constructed from the observation table (line 19) as discussed in Section 4.2.

The second phase, hypothesis validation, begins in line 20 with performing an equivalence query. If no counterexample is returned the algorithm terminates successfully with the last hypothesis (line 22). Otherwise, the counterexample is analyzed as described in Section 4.3. In case the obtained prefix is in the set of prefixes, the obtained suffix will be used as the basis for a new abstract suffix (line 26). In case the prefix is unknown, it will be added to the set of prefixes (line 28).

As discussed in the previous section, this leads to progress in one of three dimensions: new locations (or at least less permutations), new register assignments, or new guarded transitions. Progress achieved in any of the three dimensions is

Algorithm 2 L^*_{RA}

Require: A set of parameterized input symbols I
Ensure: An RMM model $\mathcal{H}$ with $tracesof\mathcal{H} = tracesofSUL$

$\hat{U} := \{\epsilon\}$ ▷ *Initialize observation table*
$U := \hat{U} \cup \{a(\bar{d}) \in min_{\sqsubseteq}\{\mathcal{R}\}\}$ ▷ *Use one "base-case" per input*
$V := \{\epsilon\}$ ▷ *Use inputs as suffix patterns*
loop
  repeat
    $T :=$ compute_residuals(U, V) ▷ *Fill table using MQs*
    if $\langle U, V, T\rangle$ not closed **then**
      Let u in $U \backslash \hat{U}$ s.t. $\forall u' \in \hat{U} \;.\; T(u) \not\equiv_V T(u')$ ▷ *New access seq.*
      $\hat{U} := \hat{U} \cup \{u\}$ ▷ *Extend prefixes*
      $U := U \cup \{ua(\bar{d}) \;:\; a(\bar{d}) \in min_{\sqsubseteq}\{u^{-1}E^{\mathcal{R}}_{\lambda}\}\}$ ▷ *by "base cases"*
    end if
    if $\langle U, V, T\rangle$ not register-consistent **then**
      Let $ua(\bar{d}) \in U$ s.t. for $d \in ValSet(u) \setminus ValSet(a(\bar{d}))$:
        - d is memorable in $T[ua(\bar{d})]$ proven by $\overline{v} \in V$
        - d is *not* memorable in $T[u]$ ▷ *To make d memorable in u:*
      $V := V \cup \{a \cdot \overline{v}\}$ ▷ *Extend suffixes accordingly*
    end if
  until $\langle U, V, T\rangle$ **is** closed **and** register-consistent.
  $\mathcal{H} :=$ construct_hypothesis(U, V, T)
  $ce := eq(\mathcal{H})$ ▷ *Perform equivalence query*
  if $ce = 'OK'$ **then**
    return $\mathcal{H}$ ▷ *Done!*
  end if
  $(u, v) :=$ decompose(ce) ▷ *cf. Prop 5*
  if $u \in U$ **then**
    $V := V \cup \{Acts(v)\}$ ▷ *New remapping, location, or assigment*
  else
    $U := U \cup \{u\}$ ▷ *New guarded transition*
  end if
end loop

strictly monotonic (Proposition 5). On the other hand, we will never have more locations, register assignments, or transitions as in the canonical automaton for L (Prop. 3, Prop. 2, and Prop. 5).

As usual, we will estimate the number of necessary membership and equivalence queries in terms of the size of the canonical RA for the considered 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.[4]

Each processing of a counterexample, which may require $O(m^m)$ membership queries, will lead to a refined observation table from which a new hypothesis automaton can be constructed. This automaton 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. 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 RA for 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 m abstract parameters, which can be instantiated by less than np parameters in the potential of a word in less than $(np)^m$ combinations. The number of membership queries needed to construct all closures lies therefore in $O(tnr \cdot (np)^m)$.

Theorem 1. Regular data languages can be learned with $O(t+nr)$ equivalence queries and $O(tnr \cdot (np)^m \ + \ (t + nr) \cdot m^m))$ 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 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, the first m can be reduced to r quite easily using "restricted" abstract suffixes, i.e., ones with some fixed constraints about equalities in the suffix. The number of required registers r will typically grow much slower than the model size. The second m estimates a worst case where all data values in the suffix of a counterexample are identical. This is a rather theoretic case. In practice, parameters often have different types with the effect that the largest group of identical data values in a counterexample is considerably smaller than size m. These observations are supported by our experiments.

5 An Example Run of the Algorithm

In this section we give an example of a complete run of our algorithm, using the protocol example from Figure 1. and present some performance data for our implementation of the algorithm.

[4] 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).

	ϵ	$\mathbf{login}(p_1, p_2)$	$\mathbf{logout}()\ \mathbf{login}(p_1, p_2)$
ϵ (l_0)	-	$\mathbf{login}(1,2)$ -	$\mathbf{logout}()\mathbf{login}(1,2)$ -
$\mathbf{signup}(1,2)$ (l_1)	-	$\mathbf{login}(1,2)$ + $\mathbf{login}(3,4)$ -	$\mathbf{logout}()\mathbf{login}(1,2)$ - $\mathbf{logout}()\mathbf{login}(3,4)$ +
$\mathbf{signup}(1,2)\mathbf{login}(1,2)$ (l_2)	+	$\mathbf{login}(3,4)$ +	$\mathbf{logout}()\mathbf{login}(1,2)$ + $\mathbf{logout}()\mathbf{login}(3,4)$ -
$\mathbf{signup}(1,2)\mathbf{login}(3,4)$	-	$\mathbf{login}(1,2)$ + $\mathbf{login}(5,6)$ -	$\mathbf{logout}()\mathbf{login}(1,2)$ + $\mathbf{logout}()\mathbf{login}(5,6)$ -
$\mathbf{signup}(1,2)\mathbf{login}(1,2)\mathbf{setpw}(3)$	+	$\mathbf{login}(4,5)$ +	$\mathbf{logout}()\mathbf{login}(1,3)$ + $\mathbf{logout}()\mathbf{login}(4,5)$ -
$\mathbf{signup}(1,2)\mathbf{login}(1,2)\mathbf{logout}()$	-	$\mathbf{login}(1,2)$ + $\mathbf{login}(3,4)$ -	$\mathbf{logout}()\mathbf{login}(1,2)$ + $\mathbf{logout}()\mathbf{login}(3,4)$ -
$\mathbf{signup}(1,2)\mathbf{login}(1,2)\mathbf{delete}()$	-	$\mathbf{login}(3,4)$ -	$\mathbf{logout}()\mathbf{login}(3,4)$ -

Fig. 7. Observation Table (only showing a subset of all prefixes)

	ϵ
ϵ (l_0)	-
$\mathbf{signup}(1,2)$	-
$\mathbf{delete}()$	-

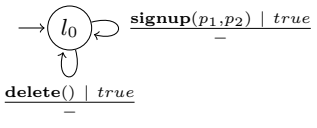


Fig. 8. Observation Table after first round of learning (only showing a subset of all prefixes) for the example of Figure 1 and corresponding hypothesis.

The resulting first observation table for the example is shown (partly) in Figure 8. The left column contains prefixes. Prefixes from $\hat{U}$ are shown in the upper part of the table.

The second column is labeled with the initial abstract suffix, the empty word. All table cells represent the trivial closure with empty domain. The table was initialized as described in Section 4.4. The algorithm starts by constructing closures for all prefixes in U and the empty suffix. Since all prefixes are not in L, the table is immediately closed and consistent. In the constructed hypothesis, shown in the right of Figure 8, all prefixes lead to one non-accepting state.

An equivalence query returns the counterexample $\mathbf{signup}(1,1)\mathbf{login}(1,1)$ which is in L but rejected by the hypothesis. Performing step A of handling counterexamples results in a word $\mathbf{signup}(1,2)\mathbf{login}(1,2)$, which still is a counterexample. When refining the word to be supported by the (empty) assignment along the **signup**-transition in the hypothesis (step B of handling counterex-

amples), the words $\mathbf{signup}(1,2)\mathbf{login}(1,3)$ and $\mathbf{signup}(1,2)\mathbf{login}(3,2)$ are no longer counterexamples. In order to subsequently correct the yet empty assignment, we add the abstract suffix **login** to the table. This analysis is also shown in Figure 10.

Fig. 9. Analysis of first counterexample

Step	u	$a(\bar{d})$	v	λ_L / $\lambda_{L(\mathcal{H})}$
Transition		$\mathbf{signup}(1,1)$	$\mathbf{login}(1,1)$	+ / −
Memorable		$\mathbf{signup}(1,2)$	$\mathbf{login}(1,2)$	+ / −
		$\mathbf{signup}(1,2)$	$\mathbf{login}(i,j)$	− / −

Fig. 10. Analysis of first counterexample leading to new memorable data values. Data values i and j indicate all representative words with any data value in these positions.

When completing the table, the closure for the prefix $\mathbf{signup}(1,2)$ will be incompatible with the other closures, which can be seen in the intermediate observation table in Table 3. In order to get a *closed* observation table, $\mathbf{signup}(1,2)$ is added to $\hat{U}$, and $(U \backslash \hat{U})$ will be extended accordingly. From the closed table we construct the hypothesis that is shown in the right of Figure 3.

Step	u	$a(\bar{d})$	v	λ_L / $\lambda_{L(\mathcal{H})}$
Transition		$\mathbf{signup}(1,1)$	$\mathbf{login}(1,1)$	+ / −
Memorable		$\mathbf{signup}(1,2)$	$\mathbf{login}(1,2)$	+ / −
Permutation		$\mathbf{signup}(1,2)$	$\mathbf{login}(1,2)$	+ / −
Transition	$\mathbf{signup}(1,2)$	$\mathbf{login}(1,2)$		+ / −
	$\mathbf{signup}(1,2)$	$\mathbf{login}(i,j)$		− / −

Fig. 11. Analysis of second counterexample leading to a new L-essential prefix. Data values i and j indicate all representative words with any data value in these positions.

We will get the same counterexample as in the first round. Analyzing it, we perform the refinement steps described in Section 4.3 and shown explicitly in Figure 5. We first perform the refinement steps for the empty prefix. First we transform $\mathbf{signup}(1,1)\mathbf{login}(1,1)$ to $\mathbf{signup}(1,2)\mathbf{login}(1,2)$ (step A). Steps B and C will not modify this counterexample since the equalities are supported already by the hypothesis and since $\mathbf{signup}(1,2)$ is its own access sequence. The

second round starts with **signup**$(1,2)$ as u, **login**$(1,2)$ as $a(\bar{d})$, and an empty suffix v. When refining **login**$(1,2)$ to be supported by the corresponding guard of the **login**-transition from l_1 (step A), we discover that **signup**$(1,2)$**login**$(1,3)$ and **signup**$(1,2)$**login**$(3,2)$ and **signup**$(1,2)$**login**$(3,4)$ are no counterexamples. Hence, **signup**$(1,2)$**login**$(1,2)$ must be L-essential. We add it to $(U \backslash \hat{U})$ in order to represent the guarded **login**-transition in the table.

To *close* the table, we have to move the new prefix to $\hat{U}$ as its closure is incompatible with the other closures. We extend $(U \backslash \hat{U})$ 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 **signup**$(1,2)$**login**$(1,2)$**logout**$()$, however, has two memorable data values, namely 1 and 2. We thus add **logout login** to the set of suffixes. From the closed and consistent observation table, shown in Table 7, we construct the final model: the canonical RA of Figure 1. The learning algorithm terminates after 403 membership queries and three equivalence queries. These numbers were recorded from the actual implementation, which is described in the next section.

6 Register Mealy Machines

In this section we will present a Register Automaton model that allows for modeling data in outputs and discuss how such an automaton can be reconstructed from its semantics.

Definition 7 (Register Mealy Machine). *A* Register Mealy Machine *(RMM) is a tuple* $\mathcal{A} = (\Sigma, \Omega, L, l_0, X, \Gamma)$*, where*

- Σ *is a finite set of* parameterized inputs*,*
- Ω *is a finite set of* parameterized outputs*,*
- 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* $\langle l, a(\bar{p}), \bar{g}, o(\bar{x}), \rho, l' \rangle$*, where* l *is the* source location*,* l' *is the* target location*,* $a(\bar{p})$ *is a symbolic input,* $\bar{g}$ *is a guard,* $o(\bar{r})$ *with* $o \in \Omega$ *and* $\bar{r} \in (X(l) \cup \bar{p})^k$ *is a symbolic output, and* $\rho : X(l') \to (X(l) \cup \bar{p})$ *is an assignment.*

A Register Mealy Machine runs over words from W^D exactly like a Register Automaton. Only, instead of ending in an accepting or rejecting location, it produces outputs from Ω^D on the way. There, in a step $\langle l, \nu \rangle \xrightarrow{(a,\bar{d})/(o,\bar{d}')} \langle l', \nu' \rangle$ are determined by the symbolic output $o(\bar{r})$. Elements of $\bar{r}$ can refer to either variables or parameters of a, hence $d'_i = \nu(x_j)$ if $r_i = x_j$ and $d'_i = d_j$ if $r_i = p_j$, for $d'_i \in \bar{d}'$.

The *semantics* of an RMM $\mathcal{A}$ can be expressed as a function $T_{\mathcal{A}} \colon W^D \to \Omega^D$, with $T_{\mathcal{A}}(w)$ being the *last* output symbol that was emitted in the run of $\mathcal{A}$ over

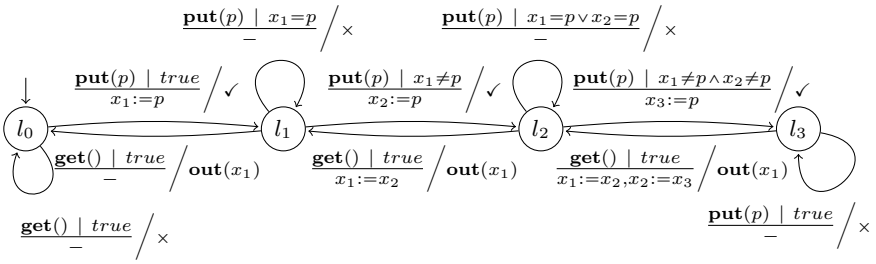


Fig. 12. RMM for a FIFO-set with a capacity of 3.

$w \in W^D$. Since a Register Mealy Machine $\mathcal{A}$ can test data values in parameters only against values in registers (and not against constants), the function $T_\mathcal{A}$ is closed under permutations on the data domain in the following sense: For all permutations π on D it holds that $T_\mathcal{A}(\pi(w)) = \pi(T_\mathcal{A}(w))$.

Instead of λ we now use $T_\mathcal{A}$ and permute the values in the range of $T_\mathcal{A}$ as well when comparing words. Other than that the ideas from Section 3 and Section 4 can be applied directly. We will skip formal definitions and just give an example of a Register Mealy Machine.

Example 2. At this point, we introduce our running example, which aligns with the field of application we envision for our technique: inferring semantic interfaces of data structures. Consider a collection data structure that allows storing a bounded number of data values, which combines aspects of both a queue and a set: when retrieving values from the collection, FIFO semantics apply. However, like in a set, it is not possible to store the same value twice; doing so will have no effect. For insertion and retrieval, the interface offers the input actions $\mathbf{put}(p)$ and $\mathbf{get}()$. The response upon **put** is ✓ or ×, signaling whether or not the collection was modified. A **get** operation is either answered by $\mathbf{out}(x)$, with x being the value that is returned, or × if the collection is empty.

An RMM of this data structure is depicted in Fig. 12 for a capacity of three. A transition $\langle l, a(\bar{p}), \bar{g}, o(\bar{r}), \rho, l' \rangle$ is represented by an arrow between l and l', with the label $\frac{a(\bar{p}) \mid \bar{g}}{\rho} \Big/ o(\bar{r})$.

7 Evaluation

We have implemented the algorithm outlined in this paper on top of LearnLib [36], our framework for active automata learning. We conducted several experiments to demonstrate the efficiency of our algorithm. Note that for all of the experiments we conducted, we used a cache, preventing membership queries for the same words to be posed twice.

In a first series of experiments, we employed our algorithm for learning models of small container data structures: a stack, a queue, and a (FIFO-)set with fixed

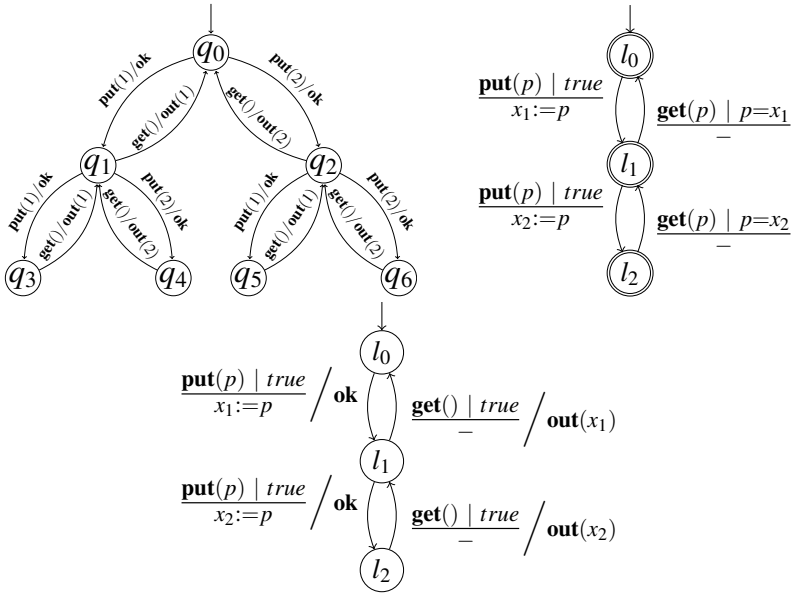


Fig. 13. Three variants of modeling a stack with a capacity of two: As a Mealy machine with a sample data domain $D = \{1, 2\}$ (left), as a prefix-closed Register Automaton (right), or as a Register Mealy Machine (middle). Unsuccessful operations (e.g., reflexive transitions) and sink locations are omitted in all three models.

capacities. All those data structures expose two input actions: **put** of arity one, and **get** without any parameters. The semantics of **put** and **get** is the same as the one in our running example (cf. Fig. 12 for the example RMM of a FIFO-set with a capacity of three). The queue and the stack allow storing the same object multiple times, while the set can only store distinct elements.

For assessing the efficiency of our algorithm, we considered two different approaches that can be employed in order to infer models of such data structures: We used a classical active learning algorithm, treating the data structure as an ordinary Mealy machine. In this case, it was necessary to restrict the size of the (visible) data domain in order to gain a finite representation. For a stack with a capacity of two and $D = \{1, 2\}$, this is exemplarily displayed in the left of Fig. 13. In the experiments we used $n + 1$ as size of the data domain for data structures of capacity n. This allows to observe the behavior of the data structures in the case where all registers store different values. The additional "new" data value is used as data parameter. We have used *symmetry reduction*, i.e., normalizing the order of data values occurring in an input word as described in [32], to reduce the number of queries when inferring plain Mealy machine models.

Table 1. Experimental results for inferring register automata models from data structures using various algorithms

Name	Mealy ($\|D\| = n + 1$)			Mealy w/ sym.red.		RA [24][5]			RMM		
	$\|Q\|$	MQs	EQs	MQs	EQs	$\|L\|$	MQs	EQs	$\|L\|$	MQs	EQs
Stack (1)	3	30	0	16	0	3	35	2	2	10	0
Stack (2)	13	252	1	52	1	4	135	4	3	18	0
Stack (3)	85	2,833	3	232	3	5	554	6	4	38	1
Stack (4)	781	39,996	4	890	4	6	2,998	8	5	53	2
Queue (4)	781	39,996	4	890	4	6	2,711	5	5	76	2
FIFO-Set (4)	206	9,484	2	128	2	6	1,566	15	5	129	12

We also compared our algorithm to an alternative way of representing output in systems with data: by modeling them as Register Automata, i.e., acceptors, and considering the (prefix-closed) data language of all valid combinations of input symbols with the respective data values in the output. This is detailed in the right of Fig. 13 for the case of a stack: here, the input symbol **get** also has a parameter, and transitions are only valid if the provided data value matches the one in the output. This resembles a common way of encoding Mealy machines as (prefix-closed) DFA. For inferring these models, we used the algorithm presented in [24]. The difference between an RA model and an RMM model is apparent in the figure: While in the RMM model (middle) transitions have outputs with data values, these outputs have to be encoded as guarded transitions in the RA model.

The results of this evaluation series are displayed in Tab. 1. Our novel algorithm impressively outperforms the alternative approaches in all but one cases. When looking at the series of stacks with growing capacities, it is particularly striking that, while the number of membership queries for learning RAs grows quickly, there is only moderate growth for the inference of RMMs. As was analyzed in [24], handling counterexamples in order to infer guards is a task with an exponential worst-case complexity in the number of registers, as numerous combinations of (in-)equalities between parameter values have to be considered. When modeling the component as an RMM, however, memorable data values are provided by output symbols without any additional effort. Apart from this improvement in terms of efficiency, our algorithm also produces a much more intuitive model. In the case of the FIFO set of size 4, on the other hand, inferring plain Mealy machines using symmetry reduction is as efficient as inferring RMMs. This is due to the fact that for the FIFO set guards have to be inferred, which is expensive.

[5] The algorithm infers a complete model also containing a sink, hence the greater number of locations compared to our new algorithm.

Table 2. Impact of the size of D on model and algorithmic complexity when inferring classical Mealy machine models of a stack with a capacity of 4

$\|D\|$	$\|Q\|$	w/o sym.red.		w/ sym.red.	
		MQs	EQs	MQs	EQs
1	5	32	2	32	2
2	31	486	4	277	4
3	121	3,072	4	657	4
4	341	12,710	4	854	4

Considering the plain Mealy machines, one notices the rather large state space. This is due to the fact that, since Mealy machines are data-unaware, each possible combination of data values results in a different state (as can also be seen in Fig. 13). Further, to faithfully relate data values in both input and output in a Mealy machine, it would be necessary to have at least as many different data values as can be distinguished by the component. This leads to an exponential growth of the state space (and thus complexity in terms of membership queries), as can be seen in Tab. 2, where both the size of the state space and the query complexity are displayed for growing values of $|D|$ and a fixed capacity of 4. As with increasing capacities more data values are needed to observe the behavior exhaustively, one easily sees that this becomes intractable very quickly. Symmetry reduction helps to reduce the number of membership queries, but does not solve the issues regarding the large state space.

We conducted a second series of experiments in order to analyze the behavior of our algorithm on more complex data structures. For this, we chose a two-dimensional data structure, a *stack of stacks*. The interface exposes operations **push2d**, **pop2d**, **push**, **pop**, the former two operating on the (outer) "stack of stacks", the latter two on the (inner) stack (of plain values) currently at the top: **push2d**() puts an additional stack on top of the outer stack (as long as this does not violate capacity restrictions), and **pop2d**() removes this stack. On the other hand, **push**(p) pushes a value onto the current inner stack, while **pop**() outputs and removes the top value of the inner stack. The capacity of the inner stacks is denoted by m, while n denotes the capacity of the outer stack. The experimental results can be found in Tab. 3.

The inferred RMM model for the case $m = n = 2$ is shown in Figure 14: From the initial location a **push2d**() in required to make the first inner stack accessible. The transitions between locations l_1, l_5, and l_9 are operations on the first inner stack. From each of these locations a **push2d**() will lead to a subgraph, describing actions on the second inner stack – relative to the state (contents) of the first inner stack.

For this series of experiments we did not compare our algorithm to alternative approaches as this would certainly be a vain endeavor: Considering the stack for $m = 4, n = 4$ (thus capable of holding in total 16 elements), the state space of a Mealy machine with $|D| = 4$ would have significantly more than

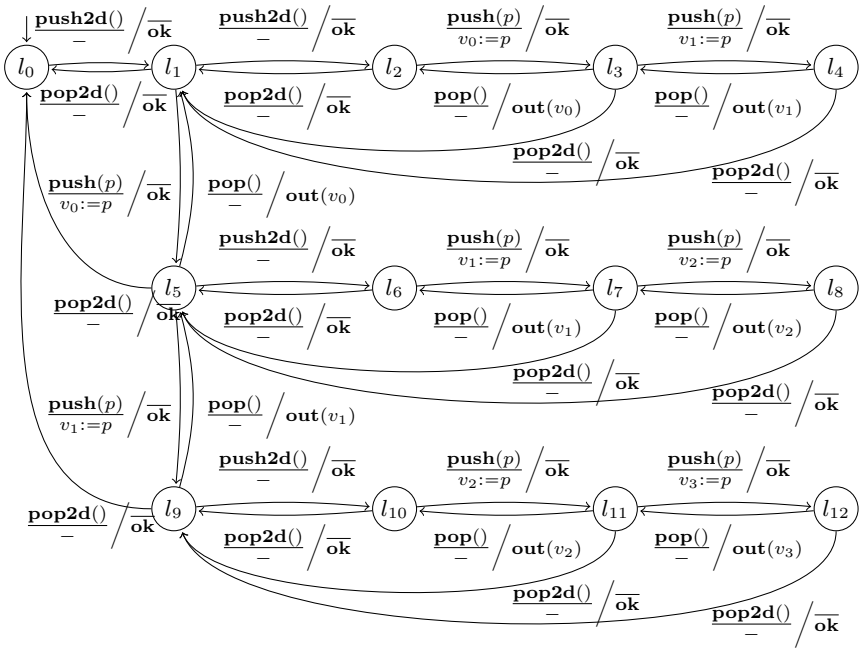


Fig. 14. RMM for a 2-dimensional stack of overall capacity 4. Operations **push2d** and **pop2d** operate the outer stack while **push** and **pop** operate the inner stacks. Unsuccessful operations (i.e., reflexive transitions) are omitted.

$4^{16} = 2^{32}$ states, which is several orders of magnitude higher than the number of *membership queries* alone required by our algorithm. In particular, we tested this for $n = 3, m = 3, |D| = 3$, where the unfolded Mealy machine has 65,641 states, compared to 3,910 membership queries for inferring the respective RMM. Further, when increasing any of these values, it was not possible to unfold the model in reasonable time any more.

We did not measure time in our experiments, as we deem the complexity in terms of membership queries the more relevant result. However, even these complex models could be inferred rather quickly with our tool, not exceeding 20 seconds even for $n = 4, m = 4$. This is by far lower than the time required to unfold the RMM in order to obtain a plain Mealy machine, even for $|D| = 2$.

8 Related Work

Synthesis of component interfaces has been a research interest for the past decade. Presented approaches fall into three classes.

Table 3. Experimental results for inferring a two-dimensional stack with outer capacity n and inner capacity m.

	$n = 2$			$n = 3$			$n = 4$		
m	$\|L\|$	MQs	EQs	$\|L\|$	MQs	EQs	$\|L\|$	MQs	EQs
1	7	160	1	15	470	3	31	1,142	5
2	13	373	2	40	1,596	5	121	5,126	5
3	21	744	3	85	3,910	6	341	16,454	6
4	31	1,283	5	156	8,551	9	781	44,589	9

First, *client-side static analysis* uses a static analysis of source code using the component of which a model is to be inferred. The approaches described in [40,44] mine Java code to infer sequences of method calls. These sequences are used to produce usage patterns statistically. Deviations from these patterns are then considered as candidates for erroneous usages of the component and have to be verified manually.

Second, *component-side static analysis* uses a static analysis on the component itself. In [3] an approach is presented that generates behavioral interface specifications for Java classes by means of predicate abstraction and active automata 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. Another approach uses counterexample guided abstraction refinement (CEGAR) [22] instead of active learning in order to derive a regular model from the Boolean program obtained by predicate abstraction.

Static analysis methods rely on access to source-code, either of the component or of code using the component. This cannot be assumed in all scenarios.

The third class of methods, namely *dynamic analysis*, infers interface models or properties from actual program executions. The authors of [5], e.g., present an approach for inferring probabilistic finite state automata (PFSA) describing a components' interface using a variant of the k-tail algorithm [10] for learning finite state automata from sets of examples. In [45] a special form of component interfaces, consisting of one finite state automaton per field of a class, is presented along with a static analysis and a dynamic analysis for inferring such component interfaces.

The SPY approach [19] aims at inferring behavioral specifications, represented as graph transformation rules, from Java classes. Inference is organized in two phases: First, the run-time behavior of components is observed on a small concrete data domain. Then, (symbolic) transformation rules are constructed inductively from the concrete observations.

All three of these dynamic approaches use passive learning and are thus limited to (possibly small) sets of observed executions. In case some functionality of a component is not executed, it will not be captured in the inferred model. Using active methods, on the other hand, allows for active exploration of the interface

of a component: The authors of [16], e.g., use a combination of component-side static analysis, identifying side-effect free methods (so-called *inspectors*), which are then used to identify states of the component when interacting with the component systematically.

Finally, active automata learning has been used succesfully to infer control models in anumber of case studies. It has been used to infer models of the abovementioned CTI systems [21,20], web-applications [35], communication protocol entities [1], the new biometric European passport [2], bot nets [11], a network of integrated controllers in the door of a car [39], and enterprise applications [8,7]. The particular challenges of practical application are discussed in [23] along with illustrating examples from case studies.

However, all previous approaches that infer models describing data explicitly from black-box components (i.e., [19], [38], [9], and [31]) work in two steps. The first step consists in inferring models (or in obtaining sets of examples in the case of passive learning) at the level of a concrete finite data domain. Our results show that this becomes unfeasible already for relatively small models of data structures. The active learning algorithm presented in this paper, on the other hand, infers register automata directly, without generating such an intermediate representation.

Except for [19], the approach to interface synthesis presented to in this paper differs from all previous attempts wrt. the expressiveness of the inferred models. While in all other approaches inferred interfaces are modeled as "plain" automata, the automata used in this paper allow for modeling of (restricted) program behavior comprising conditions and assignments.

9 Conclusions

In this paper, 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: 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.

We have applied this algorithm successfully and generated semantic (i.e., data-aware) interfaces for black-box components. The complexity of our "stack of stacks" examples is far beyond the reach of the state of the art in interface synthesis: our largest example, whose RMM has only 781 states, *independently* of the size of the data domain and is learned fully automatically in 20 seconds using only 9 equivalence queries, would lead to more than 10^9 states for an abstract data domain of just four values!

References

1. Fides Aarts, Bengt Jonsson, and Johan Uijen. Generating Models of Infinite-State Communication Protocols using Regular Inference with Abstraction. *submitted to ICTSS 2010*, 2010.
2. Fides Aarts, Julien Schmaltz, and Frits W. Vaandrager. Inference and Abstraction of the Biometric Passport. In *ISoLA (1)*, pages 673–686, 2010.
3. Rajeev Alur, Pavol Cerný, P. Madhusudan, and Wonhong Nam. Synthesis of interface specifications for Java classes. In *POPL*, pages 98–109, 2005.
4. G. Ammons, R. Bodik, and J. Larus. Mining specifications. In *Proc.* 29^{th} *ACM Symp. on Principles of Programming Languages*, pages 4–16, 2002.
5. Glenn Ammons, Rastislav Bodík, and James R. Larus. Mining specifications. In *POPL*, pages 4–16, 2002.
6. D. Angluin. Learning Regular Sets from Queries and Counterexamples. *Information and Computation*, 75(2):87–106, 1987.
7. Oliver Bauer. Beherrschung emergenten Verhaltens auf Basis regulärer Extrapolation am Beispiel einer prozessgesteuerten Anwendung. Master's thesis, TU Dortmund, Department of Computer Science, Chair of Programming systems, 2011.
8. Oliver Bauer, Johannes Neubauer, Bernhard Steffen, and Falk Howar. Reusing System States by Active Learning Algorithms. In Alessandro Moschitti and Riccardo Scandariato, editors, *Eternal Systems*, volume 255 of *CCIS*, pages 61–78. Springer Verlag, 2012.
9. Therese Berg, Bengt Jonsson, and Harald Raffelt. Regular Inference for State Machines Using Domains with Equality Tests. In José Luiz Fiadeiro and Paola Inverardi, editors, *Proc. FASE '08,* 11^{th} *Int. Conf. on Fundamental Approaches to Software Engineering*, volume 4961 of *LNCS*, pages 317–331. Springer Verlag, 2008.
10. A. W. Biermann and J. A. Feldman. On the Synthesis of Finite-State Machines from Samples of Their Behavior. *IEEE Trans. Comput.*, 21:592–597, June 1972.
11. G. Bossert, G. Hiet, and T. Henin. Modelling to Simulate Botnet Command and Control Protocols for the Evaluation of Network Intrusion Detection Systems. In *Network and Information Systems Security (SAR-SSI), 2011 Conference on*, pages 1 –8, may 2011.
12. Manfred Broy, Bengt Jonsson, Joost-Pieter Katoen, Martin Leucker, and Alexander Pretschner. *Model-based Testing of Reactive Systems*, volume 3472 of *LNCS*. Springer Verlag, 2005.
13. S. Cassel, F. Howar, B. Jonsson, M. Merten, and B. Steffen. A Succinct Canonical Register Automaton Model. In *ATVA*, volume 6996 of *LNCS*, pages 366–380. Springer Verlag, 2011.
14. S. Cassel, F. Howar, B. Jonsson, M. Merten, and B. Steffen. A Succinct Canonical Register Automaton Model. *Submitted to JLAP*, 2012.
15. E.M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.
16. Valentin Dallmeier, Christian Lindig, Andrzej Wasylkowski, and Andreas Zeller. Mining object behavior with ADABU. In *Proceedings of the 2006 international workshop on Dynamic systems analysis*, WODA '06, pages 17–24, New York, NY, USA, 2006. ACM.
17. 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. *Science of Computer Programming*, 69(1-3):35–45, 2007.

18. Mihaela Gheorghiu Bobaru, Corina S. Păsăreanu, and Dimitra Giannakopoulou. Automated Assume-Guarantee Reasoning by Abstraction Refinement. In *Proceedings of the* 20^{th} *Int. Conf. on Computer Aided Verification, CAV'08*, volume 5123 of *Lecture Notes in Computer Science*, pages 135–148. Springer Verlag, 2008.
19. C. Ghezzi, A. Mocci, and M. Monga. Synthesizing Intentional Behavior Models by Graph Transformation. In *ICSE 2009, Vancouver, Canada*, 2009.
20. Andreas Hagerer, Hardi Hungar, Oliver Niese, and Bernhard Steffen. Model generation by moderated regular extrapolation. *LNCS*, pages 80–95, 2002.
21. Andreas Hagerer, Tiziana Margaria, Oliver Niese, Bernhard Steffen, Georg Brune, and Hans-Dieter Ide. Efficient regression testing of CTI-systems: Testing a complex call-center solution. *Annual review of communication, Int.Engineering Consortium (IEC)*, 55:1033–1040, 2001.
22. Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. Permissive interfaces. In *ESEC/SIGSOFT FSE*, pages 31–40, 2005.
23. Falk Howar, Maik Merten, Bernhard Steffen, and Tiziana Margaria. *Formal Methods for Industrial Critical Systems*, chapter Practical Aspects of Active Automata Learning. Wiley-VCH, 2012.
24. Falk Howar, Bernhard Steffen, Bengt Jonsson, and Sofia Cassel. Inferring Canonical Register Automata. In *VMCAI 2012*, to appear.
25. Falk Howar, Bernhard Steffen, and Maik Merten. Automata Learning with Automated Alphabet Abstraction Refinement. In *Twelfth International Conference on Verification, Model Checking, and Abstract Interpretation*, 2011.
26. 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.
27. H. Hungar, O. Niese, and B. Steffen. Domain-specific optimization in automata learning. In *Proc.* 15^{th} *Int. Conf. on Computer Aided Verification*, 2003.
28. Valérie Issarny, Bernhard Steffen, Bengt Jonsson, Gordon S. Blair, Paul Grace, Marta Z. Kwiatkowska, Radu Calinescu, Paola Inverardi, Massimo Tivoli, Antonia Bertolino, and Antonino Sabetta. CONNECT Challenges: Towards Emergent Connectors for Eternal Networked Systems. In *ICECCS*, pages 154–161, 2009.
29. B. Jonsson and J. Parrow. Deciding bisimulation equivalences for a class of non-finite-state programs. *Information and Computation*, 107(2):272–302, Dec. 1993.
30. Bengt Jonsson. Learning of Automata Models Extended with Data. In *SFM*, volume 6659 of *LNCS*, pages 327–349. Springer, 2011.
31. Davide Lorenzoli, Leonardo Mariani, and Mauro Pezzè. Automatic generation of software behavioral models. In *Procceedings of the* 30^{th} *Int. Conf. on Software Engineering, ICSE'08*, pages 501–510. ACM, 2008.
32. Tiziana Margaria, Harald Raffelt, and Bernhard Steffen. Knowledge-based relevance filtering for efficient system-level test-based model generation. *Innovations in Systems and Software Engineering*, 1(2):147–156, July 2005.
33. Maik Merten, Bernhard Steffen, Falk Howar, and Tiziana Margaria. Next Generation LearnLib. In *þ17 International Conference on Tools and algorithms for the construction and analysis of systems, TACAS 2011*, 2011.
34. A. Nerode. Linear Automaton Transformations. *Proceedings of the American Mathematical Society*, 9(4):541–544, 1958.
35. Harald Raffelt, Maik Merten, Bernhard Steffen, and Tiziana Margaria. Dynamic testing via automata learning. *Int. J. Softw. Tools Technol. Transf.*, 11(4):307–324, 2009.

36. Harald Raffelt, Bernhard Steffen, Therese Berg, and Tiziana Margaria. LearnLib: a framework for extrapolating behavioral models. *Int. J. Softw. Tools Technol. Transf.*, 11(5):393–407, 2009.
37. Ronald L. Rivest and Robert E. Schapire. Inference of finite automata using homing sequences. *Inf. Comput.*, 103(2):299–347, 1993.
38. H. Sakamoto. Learning Simple Deterministic Finite-Memory Automata. In *ALT '97: Proc. 8th International Conference on Algorithmic Learning Theory, Sendai, Japan.*, volume 1316 of *LNCS*, pages 416–431. Springer Verlag, Oct. 1997.
39. Muzammil Shahbaz, K. C. Shashidhar, and Robert Eschbach. Iterative refinement of specification for component based embedded systems. In 20^{th} *Int. Symposium on Software Testing and Analysis, ISSTA 2011*, pages 276–286, 2011.
40. Sharon Shoham, Eran Yahav, Stephen Fink, and Marco Pistoia. Static specification mining using automata-based abstractions. In *ISSTA 2007*, pages 174–184, New York, NY, USA, 2007. ACM.
41. 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.
42. B. Steffen, F. Howar, and M. Merten. Introduction to Active Automata Learning from a Practical Perspective. In *SFM*, volume 6659 of *LNCS*, pages 256–296. Springer, 2011.
43. 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.
44. A. Wasylkowski, A. Zeller, and C. Lindig. Detecting Object Usage Anomalies. In *ESEC-FSE '07, pp. 35-44*. ACM, 2007.
45. John Whaley, Michael C. Martin, and Monica S. Lam. Automatic extraction of object-oriented component interfaces. In *Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, ISSTA '02, pages 218–228, New York, NY, USA, 2002. ACM.
46. Pierre Wolper. Expressing interesting properties of programs in propositional temporal logic (extended abstract). In *Proc.* 13^{th} *ACM Symp. on Principles of Programming Languages*, pages 184–193, Jan. 1986.