Establishing basis for learning algorithms

Fides Aarts, Johan Blom, Therese Bohlin, Yu-Fang Chen, Falk Howar, Bengt Jonsson, Maik Merten, Ralf Nagel, Antonino Sabetta, Siavash Soleimanifard, et al.

CONNECT

Emergent Connectors for

Eternal Software Intensive Networked Systems

ICT FET IP Project

Deliverable D4.1

Establishing basis for learning algorithms

SEVENTH FRAMEWORK PROGRAMME

http://www.connect-forever.eu

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

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

Deliverable Number	:	D4.1
Title of Deliverable	:	Establishing basis for learning algorithms
Nature of Deliverable	:	R
Dissemination Level	:	Public
Internal Version Number	:	1.0
Contractual Delivery Date	:	M12
Actual Delivery Date	:	15 Feb. 2010
Contributing WPs	:	WP4
Editor(s)	:	Bengt Jonsson (UU), Bernhard Steffen (TUDo)
Author(s)	:	Fides Aarts (UU), Johan Blom (UU), Therese Bohlin (UU), Yu-Fang Chen (UU), Falk Howar (TUDo), Bengt Jonsson (UU), Maik Merten (TUDo), Ralf Nagel (TUDo), Antonino Sabetta (CNR), Siavash Soleimanifard (UU), Bernhard Steffen (TUDo), Johan Uijen (UU), Thomas Wilk (TUDo), Stephan Windmüller (TUDo)
Reviewer(s)	:	Antinisca di Marco (UDA), Valerie Issarny (INRIA)

Abstract

The CONNECT Integrated Project aims at enabling continuous composition of networked systems, by developing techniques for synthesizing connectors. A prerequisite for synthesis is to learn about the interaction behavior of networked peers. The role of WP4 is to develop techniques for learning models of networked peers and middleware through exploratory interaction.
In this deliverable, we survey the CONNECT process in order to derive requirements for the participating learning techniques. We also report on a number of case studies from which such requirements are extracted. These include requirements on the ability to interact with the networked peer, whose behavior is being learned, as well as requirements on the learned model, in order that it can be used for subsequent manipulation in the CONNECT process, such as connector synthesis. A major challenge is to extract and maintain detailed information about the interface of the networked peer, as well as to relate this information to the produced model, in which abstractions have been employed in order to make synthesis tractable. We describe our approach for representing and maintaining this information, and how we have adopted existing learning techniques to make use of it. Ontologies are proposed as a suitable vehicle for representing the information. We also report on work performed to develop and adapt existing learning tools, in order that they be suitably useful in the CONNECT process.
This deliverable summarizes the progress and achievements during Year 1 in WP4.

Table of Contents

1 Introduction

The CONNECT Integrated Project aims at enabling continuous composition of networked systems to respond to the evolution of functionalities provided to and required from the networked environment. At present the efficacy of integrating and composing networked systems depends on the level of interoperability of the systems' underlying technologies. However, interoperable middleware cannot cover the ever growing heterogeneity dimensions of the networked environment. CONNECT aims at dropping the interoperability barrier by adopting a revolutionary approach to the seamless networking of digital systems, that is, synthesizing on the fly the connectors via which networked systems communicate. Connectors are implemented through a comprehensive dynamic process based on (i) extracting knowledge from, (ii) learning about and (iii) reasoning about, the interaction behavior of networked systems, together with (iv) synthesizing new interaction behaviors out of the ones exhibited by the systems to be made interoperable, and further (v) generating and deploying corresponding connector implementations. This aim raises challenges for modeling and reasoning about system and connector behaviors, and for synthesizing specifications of connector behavior. One cannot expect all networked systems to provide formal specifications of their interaction behavior. It is then necessary to have learning algorithms and techniques to dynamically infer specifications or models of the connector-related behavior of networked peers and middleware.

A high level view of CONNECT operation is described, in Section 5.1 of D1.1[1], as a system of various enablers that exchange information about the networked system to be constructed, as shown in Figure 1.1. In particular, the **Learning enabler** (represented by the box labeled "WP4: Learning") takes interface

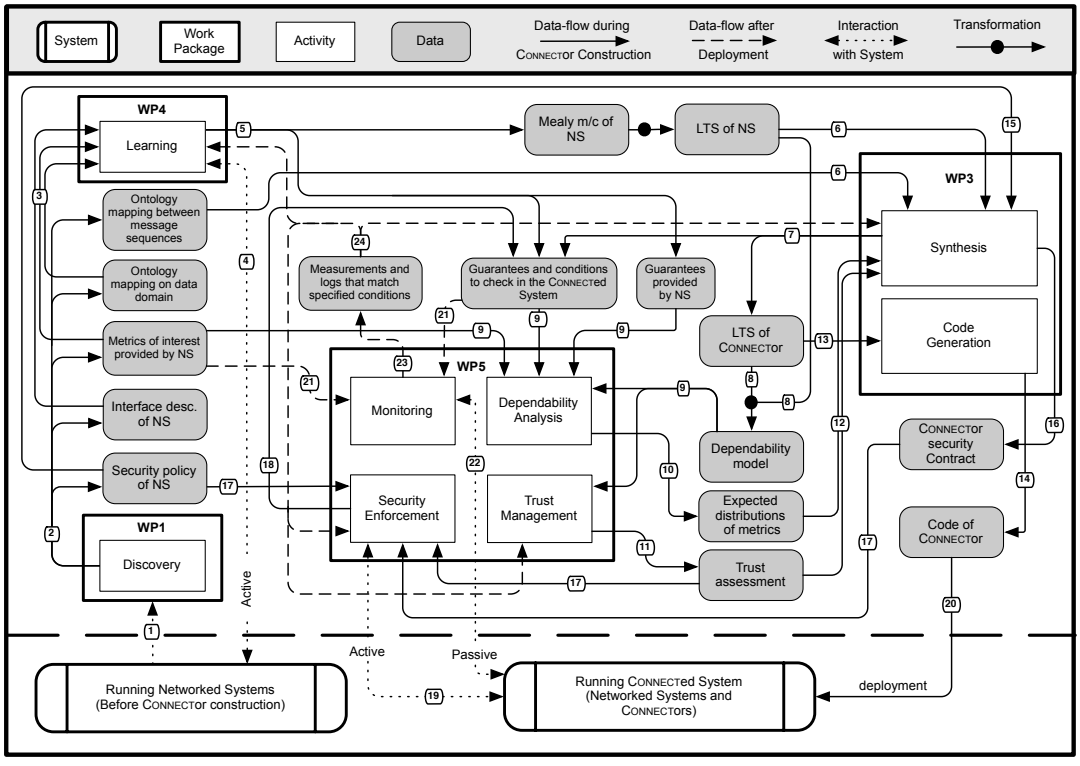


Figure 1.1: Data-Flow in the CONNECT System

descriptions of components in the networked system, along with information on used data domains, to

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

create a formal model of the behavior of networked systems using exploratory interaction, i.e., analyzing the messages exchanged with the environment. The resulting formal model can be in the form of a Mealy machine or a Labeled Transition System (LTS).

Furthermore the learning enabler will take as input information about metrics that are measureable when interacting with the networked systems and guarantees the systems make concerning those metrics. From this and through measurements, made during the course of learning, the learning enabler will produce refined or enriched data concerning the guarantees as a secondary output.

It is envisioned for the learning enabler to cooperate closely with the monitoring system, developed in WP5 (see D5.1), in order to get information about the (in-)correctness of inferred models. The learning enabler will therefore provide the monitoring system with corresponding conditions, which are to be checked on the running system. In case such conditions are violated, the learning enabler will be notified and the (incremental) process of inference will be reissued. The use of monitoring to enable the update of inferred models is one of the research goals that WP4 will pursue in the second year of the project.

1.1 The role of Work Package 4

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

> '... to develop techniques for learning and eliciting representative models of the connector-related behavior of networked peers and middleware through exploratory interaction, i.e., analyzing the messages exchanged with the environment. Learning may range from listening to instigating messages. In order to perform this task, relevant interface signatures must be available. A bootstrapping mechanism should be developed, based on some reflection mechanism. The work package will investigate minimal requirements on the information about interfaces provided by such a reflection mechanism in order to support the required bootstrapping mechanism. The work package will further support evolution by developing techniques for monitoring communication behavior to detect deviations from learned behavior, in which case the learned models should be revised and adaptors resynthesized accordingly.'

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

Task 4.1: Learning application-layer and middleware-layer interaction behaviors in which techniques are developed for learning relevant interaction behavior of communicating peers and middleware, 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

[2] CONNECT Grant Agreement, Annex I

learning procedure was (already) successfully completed. The typical behavior of a learning algorithm is to start by asking a sequence of membership queries, and gradually build a hypothesized automaton using the obtained answers. When a "stable" hypothesis has been constructed, an equivalence query finds out whether it is equivalent to the target language. If the query is successful, learning has succeeded; otherwise it is resumed by more membership queries until converging at a new hypothesis, etc. A more detailed account of active automata learning is presented in Section 2.1.

There are several techniques (e.g., [4, 21, 30, 40, 5]) essentially based on the same principles; they differ in how observations may be chosen and in the details of automata construction. The techniques guarantee that a correct and minimal automaton will be constructed if "enough" information is obtained. This class of techniques has recently started to get attention in the testing and verification community, e.g., for regression testing, and for building models to support testing of legacy systems [25, 27, 28]. Another use is in inferring rules of usage of APIs from observations of call sequences [3]. A third use is in combination with conformance testing and model checking [36, 24, 18], in order to check temporal logic properties of modules without consulting their source code. For example, we ran a series of learning experiments on a parametric network router which led to a learned model with more than 22.000 states [39].

1.2 Challenges for WP4

In order to meet the challenges posed by CONNECT, we must further develop learning techniques to make them cope with realistic systems as envisaged by CONNECT, and also adapt them to fit into the overall CONNECT operation. The former of these issues requires that we develop more efficient learning algorithms, which can deal with more complex features of component interfaces, including certain forms of Quality of Service (QoS) features, and embody them in efficient implemented learning tools. For the latter issue, we see the following challenges for WP4.

Producing models that can be used by other CONNECT enablers: we approach this by focusing our work on inferring Mealy machine models. Mealy machines are a natural model for communicating systems. We have previously developed a straight-forward adaptation of automata learning to Mealy machines [34], also implemented in LearnLib [39]. Mealy machines can be straightforwardly transformed into LTSs. The challenge is to produce Mealy machine models of the behavior of realistic networked systems, which are also useable in the context of CONNECT.

Utilizing interface descriptions and data domain information: previous work in the learning community has assumed the set of interaction primitives to be an unstructured finite set. How to use interface descriptions and information about message data has not been considered, nor in what form this information must be in order to enable successful learning.

Bridging between abstract formal models and concrete system interfaces: in WP4, we must take into consideration that learning can only be performed in terms of concrete interaction with a networked peer, whereas the desired output of the learning enabler is abstract models, e.g., in the form of LTSs. A systematic approach for bridging between these very different levels of abstraction must be developed.

Prerequisites for the learning process: our developed techniques are based on performing repeated experiments on the networked peers. We should investigate under which conditions such experiments can be performed.

We conclude that it is necessary to develop a comprehensive framework in which different interface descriptions, data information, and different levels of abstractions can be taken into account by a learning enabler. We must also investigate prerequisite requirements on networked peers.

1.3 Outline

In this deliverable, we describe our approach to solving these problems and how, so far, we have addressed the *Challenges for WP4*. We also present results from validating our solutions on a number of case studies. The included contributions are as follows.

Prerequisites for the learning process: in Chapter 2, we provide our conclusions on required conditions for learning. These are derived from conducting a number of case studies, including the popcorn scenario. The popcorn scenario will be described in detail in Section 2.3. The other case studies are described in detail in the appendices.

Handling interface descriptions and data domain information: In Chapter 3, we give an overview of our framework for incorporating interface descriptions, data information, and different levels of abstraction into the operation of a learning enabler. A key idea is to use *ontologies* for representing the information associated with these issues. We describe the information that such ontologies will provide, including:

- information about interface primitives, their parameters and data types;
- information about data types and their ranges;
- information about dependencies between interface primitives.

Bridging between abstract formal models and concrete system interfaces: in Section 3.3, we show how information about interfaces, together with other information (e.g., about data domains) is used to build an abstract alphabet of action symbols for, e.g., an LTS. We start by observing that the move between different levels of abstraction is a cross-cutting concern between CONNECT enablers, implying that information about the bridging between different abstraction levels must be represented in such a way that it can be reused in other work packages. We adapt techniques for representing and revising abstractions in areas like formal verification and program analysis, to the task of representing the connection between different levels of abstraction in the Learning enabler. We describe how the abstraction is initialized by the information available *a priori* to the Learning enabler, and how it may be revised when subsequent exploration reveals more about the structure of the networked peer.

Tool support: in Chapter 4, we report on extensions we have made to existing tools, in order to enable the implementation of the methods developed in Chapter 2 and Chapter 3.

Feasibility study: As a small feasibility study, investigating under what conditions different CONNECT enablers will cooperate, the work packages in CONNECT collectively carried out a dry run of the so-called popcorn scenario. This scenario is reported in Section 2.3 from a WP4 perspective, shedding light on the *Challenges for WP4*. The conclusions that can be drawn from the feasibility study and from several other small case studies are discussed in Chapter 2 (with respect to requirements on systems to be inferred) and in Chapter 3 (with respect to necessary expert knowledge and techniques for data abstraction).

Future plans: Chapter 5 covers future plans.

Case studies: this deliverable also reports results from a number of case studies that have been performed. They include:

- the 'Pay on The Move' (PoTM) case study, reported in Appendix A.1;
- a reformulation of the NASA Voyager mission, reported in Appendix A.2;
- the synthesis dry run (popcorn), investigation cooperation between CONNECT enabler, reported in Section 2.3;
- learning models of SIP and TCP protocol modules, reported in Appendix A.3 ;
- learning a model of the Mobile Arts Advanced Mobile Location Center (A-MLC) protocol, reported in Appendix A.4.

2 Understanding Requirements for Learning

According to the DoW, one part of the work in Task 4.1 is finding minimal requirements on interface information that enables learning of interaction behavior. In the context of CONNECT, interface information is primarily information about the syntax and semantics of parameters of the interfaces' primitives. Information about the primitives themselves can also be included: the notion of primitives is equivalent to the notion of message types, and message types are parameters of messages.

The requirements on interface information can be classified into two groups. On the one hand, there is information that is crucial for theoretical reasons, e.g., an abstraction leading to finite state models or information about the semantics of certain data values, such as sequence numbers. Exploiting this kind of information and techniques is the key to a theoretical framework that enables the behavioral inference and representation of networked systems. On the other hand, there is information that is crucial when it comes to the practical application of the theoretical methods. This includes information about how to reset a networked system into an initial state or how to prevent the tests executed during learning from having permanent effects in a productive environment.

In this chapter, we report on progress towards understanding the practical requirements for learning techniques in the context of CONNECT. Theoretical requirements, related to the handling of data values and (abstract) behavior modeling are discussed in Chapter 3. Before discussing the requirements we will give a short and informal introduction to the flavor of automata learning we use. For a more formal description we refer to [39].

2.1 A brief account of active automata learning

Machine learning deals in general with the problem of automatically generating system descriptions. Aside from synthesis of static software and hardware properties, in particular invariants [10, 19, 35], the field of automata learning, also called regular extrapolation [25] or regular inference [17], is of particular interest for software and hardware engineering [15, 33, 36]. We have used automata learning techniques in a number of contexts, e.g. to automatically construct models of web applications as demonstrated in [37]. Automata learning attempts to construct a deterministic finite automaton that matches the behavior of a given target automaton on the basis of observations of the target automaton and perhaps some further information on its internal structure. The interested reader may refer to [25, 28, 32] for our view on the use of learning. Here, we only summarize the basic aspects of our realization, which is based on Angluin's learning algorithm L^* from [4].

Definition 1 *A Deterministic Finite Automaton (DFA) is a tuple* $M = (S, s_0, \Sigma, \delta, F)$ *where:*

- S *is a finite nonempty set of states,*
- $s_0 \in S$ *is the initial state,*
- Σ *is a finite alphabet,*
- $\delta : S \times \Sigma \to S$ *is the transition function, and*
- $F \subseteq S$ *is the set of accepting states.*

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

L^*, also referred to as an active learning algorithm, learns DFAs by actively posing membership queries and equivalence queries to the target automaton in order to extract behavioral information, and

by refining successively an own hypothesis automaton based on the answers. A membership query tests whether a string (a potential run) is contained in the target automaton's language (its set of runs), and an equivalence query compares the hypothesis automaton with the target automaton for language equivalence, in order to determine whether the learning procedure was (already) successfully completed. In this case, the experimentation can stop. In its basic form, L^* starts with a hypothesis automaton that treats all words over the considered alphabet (of elementary observations) alike (i.e., it has one single state), and refines this automaton on the basis of query results iterating two steps. Here, the dual method by which L^* characterizes (and distinguishes) states on its way to construct the minimal deterministic automaton following the pattern of the well-known Nerode congruence is central [28]:

- from below, by words reaching the states. This characterization is too fine, as different words may well be Nerode congruent, i.e., having the same suffix language. This characterization thus leads to a relation between states that is contained in the relation corresponding to the Nerode congruence.

- from above, by the future behavior of the states with respect to a dynamically increasing finite set of words, which the learning algorithm produces as evidence of the difference with respect to the Nerode congruence. Future behavior is thus essentially characterized by bit vectors, where a "1" means that the corresponding word of the set is guaranteed to lead to an accepting state and a "0" captures the complement. This characterization is typically too coarse, as the considered sets of words are typically rather small, and do not fully capture the Nerode congruence. This characterization thus leads to a relation between states that contains the relation corresponding to the Nerode congruence.

The second characterization directly defines the hypothesis automata: each occurring bit vector corresponds to one state in the hypothesis automaton, which is successively refined during the learning process. The initial hypothesis automaton is characterized by the outcome of the membership query for the empty observation. It thus accepts any word in case the empty word is in the language and no word otherwise. The learning procedure (1) iteratively establishes local consistency, after which it (2) checks for global consistency.

Local consistency This first step (also referred to as automatic model completion) again iterates two phases. One checks whether the constructed automaton is closed under the one-step transitions, i.e., each transition from each state of the hypothesis automaton ends in a well defined state of this very automaton. The other phase checks for consistency according to the bit vectors characterizing the future behavior as explained above, i.e., whether all reaching words with an identical characterization from above possess the same one-step transitions. If this is not the case, a distinguishing transition is taken as an additional distinguishing future in order to resolve the inconsistency, i.e., the two reaching words with different transition potential are no longer considered to represent the same state.

Global equivalence After local consistency has been established, an equivalence query checks whether the language of the hypothesis automaton coincides with the language of the target automaton. If this is true, the learning procedure successfully terminates. Otherwise, the equivalence query returns a counterexample, i.e., a word which distinguishes the hypothesis from the target automaton. This counterexample gives rise to a new cycle of modifying the hypothesis automaton and starting the next iteration.

In a practical attempt of learning legacy systems in a black-box setting, the equivalence tests can only be approximated, but membership queries can be answered by testing the target systems. We investigated several methods for approximating equivalence queries. One such way was via conformance testing [12, 20]. In fact, it turns out that learning and conformance testing have a lot in common [6].

In contrast to DFA's, reactive systems do not distinguish between accepting states and non accepting states, but produce some output in response to the inputs. Mealy machines are well-known models of "deterministic" reactive systems. We therefore adapted Angluin's algorithm to work on Mealy machines in order to better capture the needs of reactive systems [32].

Improved algorithms for equivalence queries If a representation of the target automaton is available (i.e., if we are in a white-box context), then an equivalence query can be performed by comparing the language of the hypothesis automaton with that of the target automaton. Such algorithms are applicable, e.g., in compositional verification, where learning can be employed to find small assumptions on the context of a component [11], or when combining model checking and learning [36]. During Y1, we have improved on the efficiency of the best available algorithms for performing such comparisons, even for the case where the representation of the target automaton is nondeterministic [2]. Since white-box scenarios are not the main focus of CONNECT, we provide only a brief summary of this improvement in the following paragraph; details are available in [2].

In a white-box scenario, an equivalence check is essentially the problem of checking for language inclusion between two finite automata, which is PSPACE-complete when one of the automata (the one acting as specification) may be nondeterministic. Broadly, there are two types of methods for proving language inclusion. One type is based on computing a simulation relation on the states of the two automata: this can be done in polynomial time, but the method is incomplete. Another type is based on performing a subset construction: this is complete, but typically causes an exponential blow up. Recently, Wulf et al. [41] proposed a new approach based on so-called *antichains*, which improved the efficiency of methods based on the subset construction. In our Y1 work [2], we describe a new approach that nicely combines the simulation-based and the antichain-based approaches: a computed simulation relation is used for pruning out unnecessary search paths in the antichain-based method. Extensive experimentation on a large set of finite automata obtained from verification examples show that this approach significantly outperforms the previous antichain-based approach.

2.2 Conducted Case Studies

The CONNECT requirements on learning reach quite far. Ultimately, learning is supposed to provide missing knowledge about the behaviour of systems independent of the platform they run on and of the way they can technically be addressed; and all this, if possible, in real time. Learning under these circumstances and at this scale is a challenge, which we from the start, tried to better understand by investigating a varying collection of case studies, e.g.:

- Pay on The Move (WSDL)
  The "Pay on The Move" case study, reported in Appendix A.1, is centered on a web-service for mobile payment. It is taken from the EU STREP project SHADOWS (IST FP6). While in SHADOWS healing and correctness were the main interests, the case study was re-evaluated for CONNECT with respect to issues when learning web services.
- SIP and TCP
  We have shown that the techniques we have developed for building and using abstractions, reported in Section 3.3, scale to the learning of realistic protocol models, by case studies where we learn models of entities in the SIP and TCP standard protocols, reported in Appendixr̃efappend:a. The implementations to be learned were existing ones, provided by the protocol simulator ns-2 [29].
- A-MLC
  Another validation, reported in Appendixr̃efappend:b, has been performed on the Mobile Arts Advanced Mobile Location Center (A-MLC) protocol, which is a commercially available middleware protocol allowing mobile network operators to provide presence information from the GSM/UMTS network. We have used an executable specification of A-MLC, developed by developers and test engineers.
- NASA Voyager (ASSL)
  ASSL is an executable specification language. A reformulation of the NASA Voyager mission in ASSL is the basis for this case study, reported in Appendix A.2. Here behavioral models were learned using the generated executable Java code corresponding to the mission's specification.
- The synthesis dry run (Popcorn)
  For the dry run, a scenario involving different stadiums and mobile devices was assumed. Each

stadium had a slightly different food ordering system. Merchants and customers would, by means of their mobile devices, establish a market place for goods typically traded in stadiums (food and drinks). The aim was to simulate parts of the connector-synthesis process envisioned for a connected world.

Whereas the first case study concerns the learning of Web services, one of the many options of how to address the peers' functionality in CONNECT, the 2nd and 3rd case study, in particular, deal with the inevitable problem of adequate abstraction. The problem studied in the fourth case study was special in that it triggered its target system at the JVM-level, while the fifth case study was specifically designed for CONNECT. It will therefore be presented in more detail in the next subsection. Detailed presentations of the other case studies are included in this document as appendices.

Despite all these differences, which required special care and dedicated technology e.g. concerning the necessary resetting between membership queries, or the mapping between the different abstraction levels, we could identify a simple learning pattern in all these case studies: a learning algorithm is instantiated with an (abstract) set of input symbols for the system to be inferred. The membership queries (sequences of input symbols) the algorithm produces in the course of learning, are then passed to a special test driver. The test driver operates the system to be inferred by 1) translating the membership queries into sequences of actions on the real system and 2) performing these actions one-by-one on the system. For each performed action a third component, a logging facility, records the reactions of the actual system. After the execution of a sequence of inputs has finished, the resulting sequence of reactions is translated into a sequence of (abstract) symbols understood by the learning algorithm and passed back (to the learning algorithm).

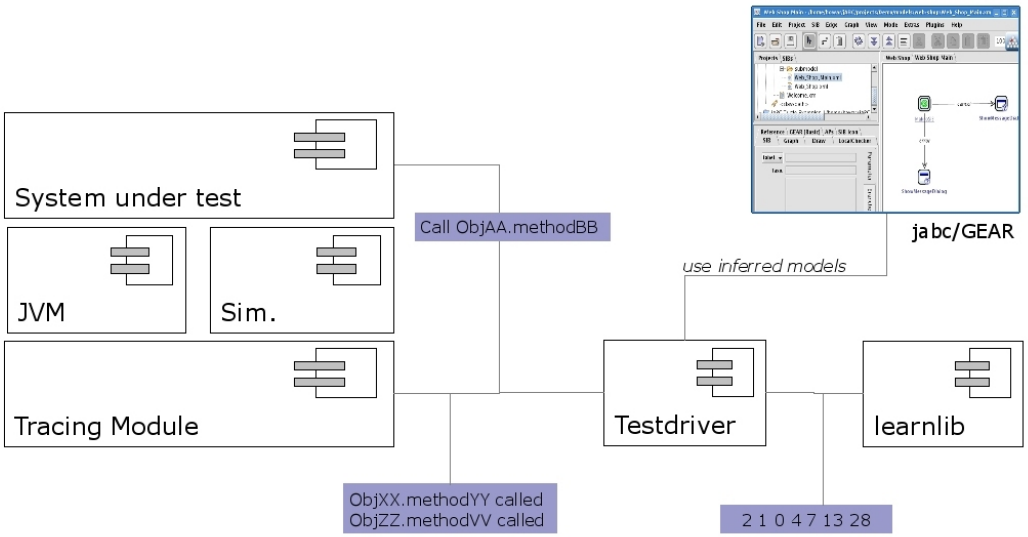


Figure 2.1: schematic view of the experimental setup

This leads to an experimental setup resembling closely the one for the case of a Java runtime environment, shown in Fig. 2.1. A learning algorithm (e.g. from LearnLib) is connected to a special runtime environment (in this case, a Java virtual machine). The virtual machine boots the system under test and some test driver. The test driver then executes actions on the system under test on behalf of LearnLib, while the virtual machine (or some other adequate facility) monitors the system reaction. The reactions are translated by the test driver and the translated versions are reported to LearnLib.

2.3 On the synthesis dry run: The Popcorn Case Study

The main aim of the dry run was to get an idea of divergences between the expected output and input of the single enablers shown in Figure 1.1. For a detailed description of the scenario we refer to D1.1 (Section 5). For the learning enabler, input was provided by the discovery enabler (WP 1) in the form of interface descriptions of the components to be inferred. On the basis of the provided information, behavioral models of the networked components were produced and additional requirements and assumptions were identified. The aim of WP4 was to produce models meeting the requirements of the synthesis enabler (or otherwise identifying obstacles).

The following explanation of the services' behaviors is derived from the scenario description given in D1.1 (Section 5). The "popcorn" services evolve around a scenario set in sport stadiums across Europe. In these stadiums merchants for popcorn and other goods can publish their adverts to the popcorn system. Consumers can search for products and place orders through the system. Payment is not part of the system. The case study is evaluated in different work packages with respect to different properties. Within WP4, the inference of behavioral models and associated concerns were considered.

During the dry run the realizations in two different stadiums have been analyzed: one stadium's implementation of the popcorn system was assumed to be based on a Tuple Space, whereas in the other stadium, a UPnP-based peer-to-peer system was assumed to exist. The models inferred by the learning enabler have been confirmed to match the ones assumed as input by WP 3 (synthesis).

Here we will only discuss the results for a part of the Tuple Space scenario (the merchant's web-service), as inference does not differ in principle for the other networked systems. The complete outcome of the dry run can be found as a special appendix in D1.1[1].

The whole dry run had several aims. Aside from making explicit divergences in the formalisms used by the different enablers (and developing methods to bridge the gaps) for the learning enabler, these aims were:

- *Interface descriptions*
  Using only the input provided by the discovery enabler, we attempted to infer the behavior of the services by means of active learning techniques. To apply those, some modifications to the original theoretical approach had to be made (Section 2.3.1).

- *Domain specific knowledge*
  Aside from interface descriptions, in a real setup extra domain specific knowledge would be needed. We analyzed what kind of knowledge this would be, and what impact different qualities of knowledge would have on the result (Section 2.3.1).

- *Output format*
  Considering the expected input for the synthesis enabler a method had to be found to transform the direct learning output (finite state acceptors) into a format meeting these requirements (Section 2.3.3).

- *Additional output*
  The synthesis enabler expects as input very abstract models. The learning enabler takes as input concrete service descriptions. To bridge this gap, abstractions of concrete models are produced by the learning enabler. For the synthesis enabler, to be able to generate real connectors after the matching (on the abstract level) is accomplished, knowledge about the necessary model refinements (back to the concrete level) will be needed. The knowledge (and additional output) required for this purpose should be examined (Section 2.3.2).

- *Additional assumptions*
  Active learning makes some (implicit) assumptions about the available ways to interact with the system to be inferred. One aim was to make these assumptions and possible related pitfalls explicit (Section 2.3.4).

[1] Available at http://www-roc.inria.fr/connect/connect-dry-run/.

```
  ...
  <xsd:element name="Publish">
  ...
     <xsd:element name="productID" type="xsd:string"/>
     <xsd:element name="location">
        <xsd:complexType>
...
     <xsd:element name="numberAvailable" type="xsd:int"/>
     <xsd:element name="price" type="xsd:int"/>
     ...
     <xsd:element name="merchantID" type="xsd:string"/>
     ...

 <xsd:element name="Request">
     <xsd:element name="productID" type="xsd:string"/>
     ...
     <xsd:element name="numberOrdered" type="xsd:int"/>
     <xsd:element name="requestID" type="xsd:string"/>
     <xsd:element name="merchantID" type="xsd:string"/>
     <xsd:element name="consumerID" type="xsd:string"/>
     ...

  <wsdl:message name="Out(Publish)">
     <wsdl:part name="body" element="Publish"/>
  </wsdl:message>

  <wsdl:message name="Notification(Request)">
     <wsdl:part name="body" element="Request"/>
  </wsdl:message>
```

Figure 2.2: Excerpt from the specification of the Tuple Space Vendor

2.3.1 From WSDL to finite Mealy machine models

To learn a networked system, an alphabet of the system under analysis is needed by the learning enabler. This alphabet is the set of input actions from which input sequences are formed. The complete set of actions that can be performed on the system is the cross product of the different valuations of each primitive's parameters. In order to circumvent the exponential influence of the set of input symbols on the number of experiments, a finite abstraction on the data domains of the primitives parameters was constructed (cf. Section 3.3). The information required, of the kind discussed in Chapter 3, was assumed to be available to the learning enabler.

From the discovery enabler, interface descriptions (WSDLs) for the single components were provided. The required abstractions on the data domains were hand tailored. All interface descriptions followed the same structure: definition of complex data types from primitive ones, definition of messages with complex data parts, definition of primitives as signatures of messages and finally definition of interfaces as sets of primitives. As an example, excerpts of the specification of the Tuple Space vendor is shown in Fig. 2.2. The figure contains definitions for complex data types, messages and primitives.

Learning uses symbolic alphabets whose letters are (state-aware) translated into concrete symbols - and back (see Section 3.3). The `Out(Publish)` primitive, for example, has a complex parameter, `Publish`, containing information about offered products, prices and available amounts along with the merchant's unique identifier. We introduced an abstract `Out(Publish)` symbol, concretized into a message always using the same unique identifier, the same product, price and amount. During the execution of the experiments the two symbolic valuations of the messages are replaced by known concrete valuations with the corresponding properties. Some data is shared between different messages of the same experiment, but needs to be reset between experiments (such as identifiers). These values are kept by the abstraction as global variables and used on demand during concretization.

As we assumed to be dealing here with a real black-box scenario, the equivalence queries would have to be approximated. This could be done, e.g., using conformance testing methods [12]. As a matter of fact, however, none of the analyzed systems would require an equivalence test in order to identify all

states. For each system all states would differ directly in the associated output behavior. This behavior is due to the assumptions we made about the setup and the hypothetical implementations.

We assumed that misplaced messages (resends, wrong ordering, etc.) would have been ignored or dropped by the systems.

Regarding the number of networked systems, we assumed only one instance of each entity of a kind during the experiments (one merchant, one client, one Tuple Space). Furthermore, for the sake of simplicity, it was assumed that certain workflows would act like transactions. An ordering once begun would not be interrupted by a search for new products or the arrival of a second order. Dropping any of these assumptions would have led to models with more transitions and more states. Other problems covered by this simplification are related to the transformation of learning output into adequate synthesis input.

The actual inference of the behavioral models was carried out in a setup resembling the one shown in Fig. 2.1. The runtime environment was then replaced by a human “oracle”, compiling by hand the output a system would generate.

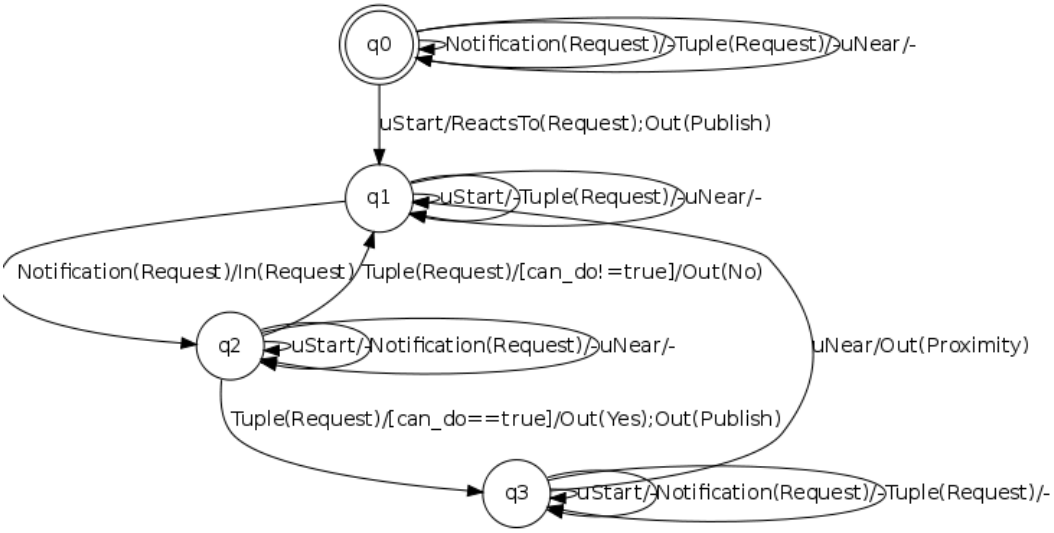


Figure 2.3: Mealy machine model of the Tuple Space Merchant

In all five cases (Tuple Space, Ts Merchant, Ts Consumer, Upnp Merchant, Upnp Consumer) the obtained models are quite small. This is mainly due to the assumptions of only one entity of a kind and of a perfect (as coarse as possible without missing behavior) abstraction. Figure 2.3 shows a graphical representation of the model a learning algorithm would create exploring the Tuple Space Merchant. The transitions are labeled with the appropriate input and output symbols. The shown output symbols are the return values (messages) of the primitives’ invocations.

According to the obtained model, the component behaves as follows: when started by the user, the service publishes its products and registers to receive requests by customers made for such products. On the notification of such requests the according request is evaluated with respect to the distance to the potential customer and the ordered amounts. If the evaluation is positive (the order can be processed) the service will answer accordingly. As soon as the merchant’s device then comes close to the consumer’s device a proximity notification is issued.

2.3.2 From Mealy machine models to LTS

The transformation from Mealy machine models into Labeled Transition System models (LTS), cf. deliverable D3.1 for a definition, is in principle not problematic. Each state of the Mealy machine will become a location of the LTS. The transitions are divided into input and output. Each part becomes a transition in the

LTS, connected to a dedicated location. An LTS produced this way is nearly bipartite. Only for transitions of the Mealy machine that contain several output symbols, chains of output transitions will be generated. Figure 2.4 shows the LTS derived from the Mealy machine from Fig. 2.3.

The main difference between the LTSs derived from the inferred Mealy machine model and the ones expected by the synthesis enabler is the existence of transitions that model user actions (colored red in Fig. 2.4). These transitions are needed during learning in order to obtain deterministic models. These transitions, however, are problematic for the synthesis enabler, as they do not have matching counterparts in the system to connect to. The synthesis enabler is concerned only with the network-related behavior of the system, but can (as opposed to learning algorithms) deal with nondeterminism in the components.

During the dry run, it was assumed that the user transitions could be removed from the models in a straightforward way, making the subsequent output transitions begin in the location of the user transition. It is however not evident if this transformation is unambiguously reversible and if it needs to be. It is furthermore not clear if the transformation is as simple in the case of parameterized more complex user inputs.

Another potential problem arises from the requirement of the synthesis enabler to have matching names in two to-be-connected systems or at least an ontology matching between them. These abstract names will typically be created as part of an abstraction during the process of learning and will have to be provided as additional output of the learning enabler. In order to enable synthesis, the learning enabler will have to be able to assign properties to those names (by means of ontological relations) thereby realizing what the naming reflects. To be able to synthesize running code from the model of a connector, knowledge about necessary model concretizations will have to be provided alongside the abstract models.

2.3.3 Conformance with synthesis input

One aim of the dry run was to make gaps (in the expressiveness) between the modeling formalisms used by different enablers explicit. The input from the discovery enabler could be used in a straightforward way. It covered only information about primitives and data types on a syntactic level. For the learning to be successful, more (semantic) information would have been needed. For a detailed discussion about the information needed about data types and data values, we refer to Chapter 3.

The output we produced almost met the models that have been assumed as input by the synthesis enabler. However, even the Mealy machine models we present here are quite abstract. One of the future challenges of WP 4 will be inferring models on a lower level, e.g., including parameters and developing means of generating appropriate abstractions.

Most of the differences between the output produced by the learning enabler and the input assumed by the synthesis enabler originate in different assumptions about the imagined actual systems. These differences would not exist if actual implementations had been the basis of the whole process.

The second source for differences has already been discussed. It is the fact that synthesis models contain only transitions for network-related primitives of the networked systems, while the models produced immediately by the learning enabler will contain transitions for other primitives (e.g., user inputs) as well.

2.3.4 Summary

Though this is only a very small scenario in which most of the results achieved by learning have been achieved manually, there are some lessons to be learned:

1. active learning relies on several properties of the inferred black box system as pointed out throughout the text. These properties will have to be guaranteed for the networked systems.

2. the abstraction we introduced was chosen carefully and validated to reveal complete, input-deterministic behavioral models of the systems. In general, this has to be considered as an approximation. It may well be that some parameter valuation leads to behavior an abstraction does not cover. In general

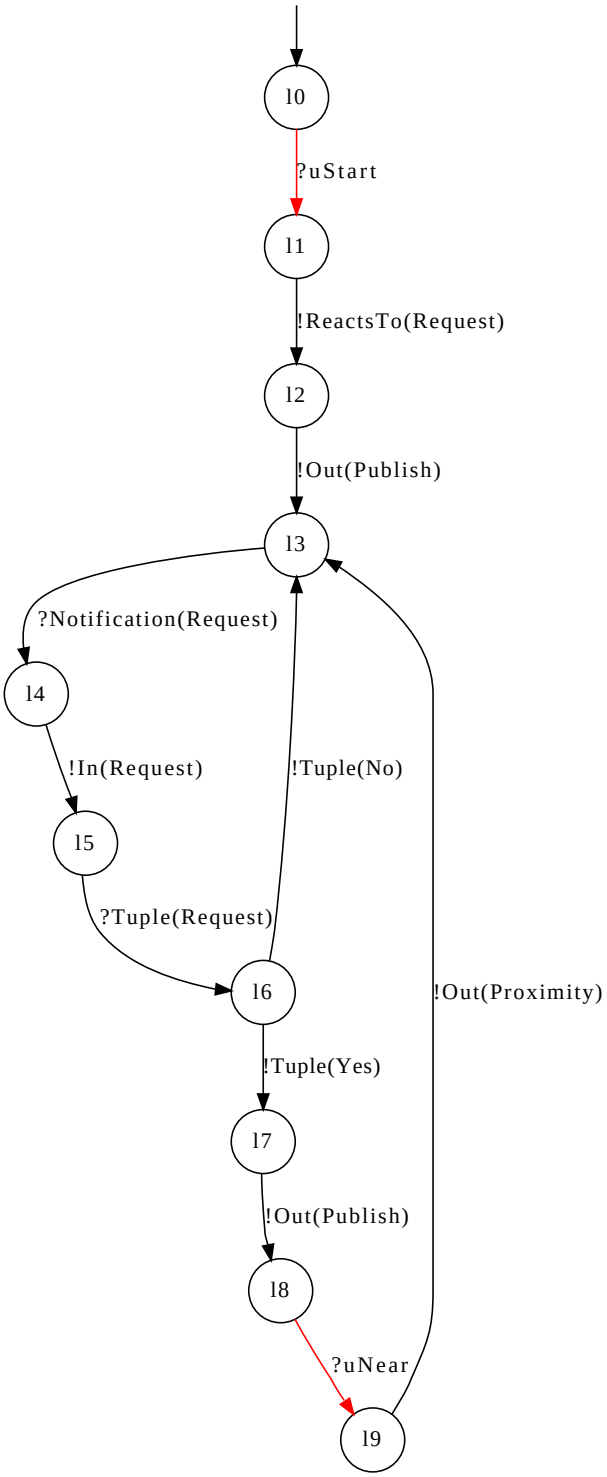


Figure 2.4: LTS model of the Tuple Space Merchant

the quality of abstractions will rely on domain knowledge and on an idea of what the intention of the inferred system is.

3. the model transformation used to provide synthesis ready output may not be trivial in the general case. It will be necessary to extend the dry run to include the generation of running code and to complement the dry run with a case study on implemented systems.

What has not been considered here is the effect that different levels of abstractions will have on the size of the models and on the number of queries needed. The number of parameters has an influence on the number of queries needed that is exponential. When choosing the right level of abstraction, the trade off between making things too expensive to be dealt with and missing relevant behavior has to be considered. One big step forward will be moving the responsibility for supplying a good abstraction refinement from the user to the learning algorithm. The learning algorithm will then incrementally increase the number of abstract valuations by exploring and partitioning the concrete parameter-space in whatever fashion.

A future interest will also be to integrate time related behavior learning and automata learning. For the sake of the dry run it was assumed that the inferred systems react instantaneously to messages and do not have any time dependent behavior (as timeouts) so that the automata learning algorithm will gain control, e.g., over timeout experiments to be performed.

2.4 Discussion

From the common experimental setup and from the lessons we learned in the case studies, the following general practical requirements on interface information, interfaces and networked systems in general can be stated.

Reset: Learning reactive systems in principle requires means of re-setting the inferred system to its initial state: in order to guarantee meaningful results, all membership queries (or experiments) have to be performed using the same initial conditions. This is a fairly strong requirement and may not be assumed to hold in the case of real-life systems. In none of the case studies an explicit reset mechanism was present. We have found several ways to handle this issue:

- In the WSDL study, abstraction was used to circumvent resets. This was achieved by using new, free unique transaction identifiers for every experiment. The “freshness” of the used values could be tested before every experiment by utilizing some of the primitives of the service that were assumed not to alter the state of the system and to provoke different reactions for free and used values.
- A similar solution was used in the A-MLC case study, where the learned service was able to create a new process to handle each newly initiated session.

  In the scenario of CONNECT this seems to be a promising approach. Networked systems (on the protocol level) are naturally able to handle multiple connections and provide a behaviorally equivalent service for each new connection.
- In the NASA Voyager case study a globally valid “homing sequence”, a sequence of actions leading to the initial state was given. This homing sequence however was derived manually before the actual learning process could start. The sequence consisted of a number of primitives that would empty the underlying message channels.

  This approach may be generalized and exploited for parts of the middleware that rely heavily on the message passing paradigm. For this system an arbitrary-length sequence of actions that consume messages, un-subscribe and destroy channels will leave the system in a defined state. Executing the sequence twice will still lead to the same “empty” state. For most other systems that comprise a more complicated control structure it is not obvious that a globally valid homing sequence even exists.

- In the SIP and TCP study, the implementation of the protocol module was restarted for each new membership query/test case. Realistically, this can be done if one has access to an implementation for off-line experimentation, which is not always the case in CONNECT scenarios.

  Part of the contribution of WP 4 is to identify requirements on the interfaces (or systems) to be inferred. We suggest a certain "design for CONNECT" or more specifically in our case a "design for learnability". One key feature of systems that are inferrable is a means of resetting the systems.

A general conclusion that can be drawn from the case studies is that means for resetting inferred systems can be provided in manifold ways. The hard assumption of an explicit reset can be relaxed to the assumption of behaviorally indistinguishable instances. Interface information (provided through discovery) will have to include information on how to reset a specific system under test into its initial state. As part of a "design for learnability", we propose that systems designed to be connected are designed to come with a reset mechanism of some kind.

Testability: Learning is essentially driven by counterexamples. As we assume to be dealing here with real black-boxes, equivalence queries will have to be approximated through (conformance) testing. The theoretical complexity of conformance tests may be exponential in the size of the system when measured in tests to be executed. The networked systems that are to be inferred as well as the infrastructure must be laid out to support this extra load.

As a matter of fact, all of the systems in the case studies would require only a small number of equivalence tests in order to identify all states. For most systems all states would differ directly in the associated output behavior. In order to keep costs and load for equivalence approximation low, systems that are especially designed to fit the needs of a connected world should be "talkative" in the sense that each internal control state produces a unique output signature.

Reversibility: To learn systems in a real environment, it is necessary to assume that the experiments conducted by the learning algorithms do not harm the system. This requirement is twofold: learning produces a huge amount of interaction with the inferred system (as discussed above). And each experiment that is run on a networked component may have consequences: imagine e.g. in the stadium example a real order being placed each time the corresponding experiment is executed. For learning to be applied at runtime, it is necessary to ensure that the experiments conducted by learning have no real (permanent or non-reversible) effects.

There are two possible ways to ensure this. One would be to guarantee the reversibility of all actions and to provide means to do so. The information about negative elements could be provided by discovery through ontologies of actions. Besides avoiding unwanted consequences in the inferred systems, reversibility of actions might be used to emulate the reset of systems. However there may be actions that are naturally non-reversible (such as firing an alarm). The other way would be including a special test mode in each component and providing the learning algorithms with a means of identification for enabling this test mode. Actions performed in this test mode would then be by definition without "real" consequences. While the former solution will not work in all scenarios, the latter one re-introduces a strong assumption similar to the one that systems can be reset into initial states.

Input-Determinism: All the systems in the case studies can be thought of as protocol entities: they expose one (behavioral) interface to a (network) peer and at least one other interface to some other party. In the popcorn case study, for example, all services are assumed to run on mobile devices. The mobile devices of consumers and merchants determine the distance between the devices by some (unspecified) means, and provide the distance information to the system realizing the trading system (which decides, e.g., when to issue a proximity notification on the basis of this information). The behavioral models of the inferred services depend on (1) the messages that can be received by peers and (2) decisions resp. inputs a user (or some other system) can provide.

More generally put, most systems will (from a learning perspective) not only communicate with one peer, but at least with two peers. From the CONNECT perspective in many cases all but one peer will not be relevant and, more importantly, outside the scope of discovery.

The merchant's system in the popcorn case study may, as a second example, reject orders if the distance between the merchant and the potential customer is too great. The regular update of the current location can be modeled as an additional input. To infer the behavioral model of the system completely and without contradictions, the learning algorithm will have to use setting the location explicitly as an input to the system. Otherwise the same experiment executed at random (and different) distances to the target may result in different observations.

The same input resulting in different outputs is a symptom of inherent nondeterminism. For inherently nondeterministic systems it is impossible to measure if at some point all nondeterministic choices have been observed. Even more severe is that in general it is not possible to make the system under test make the "right" choices at certain points (in order to study the behavior from a special state). This renders it very hard to infer models of these systems.

The systems in the scope of CONNECT (protocol entities) typically have a finite and input-deterministic control behavior. One problem when learning networked systems will be getting hold of all inputs a system exhibits - not only the ones it exhibits towards the network (those are provided through discovery).

Finiteness and Concurrency: The systems in the scope of CONNECT typically will behave identically and independently (without side-effects) for two different connections: protocol entities will exhibit the same behavior for every new session or to every new peer. This assumption enables the learning algorithms to infer a model for only one connection (or peer etc.). Models including multiple entities can then be constructed after the learning through composition.

In the dry run, e.g., we assumed only one instance of each kind of peer during each experiment (one merchant, one client, one Tuple Space). Furthermore for the sake of simplicity it was assumed that certain workflows would act like transactions. A ordering once begun would not be interrupted by a search for new products or the arrival of a second order. Dropping any of these assumptions would have led to models with more transitions and more states.

In the case where different connections can influence each other learning will become more expensive (in terms of the number of experiments to be performed). As part of a "design for learnability" we suggest information on independence of connections to be provided with a service.

3 Managing Information in the learning enabler

In this chapter, we give an overview of our approach for managing and utilizing information needed by the learning enabler, in order to describe how we address the following challenges outlined in Chapter 1.

Utilizing interface descriptions and data domain information: previous work in the learning community has assumed the set of interaction primitives to be an unstructured finite set. It has neither seriously considered how to use interface descriptions and information about the messages data, nor has it considered what form this information must have to enable successful learning.

Bridging between abstract formal models and concrete system interfaces: in WP4, we must take into consideration that learning can only be performed in terms of concrete interaction with a networked peer, whereas the desired output of the learning enabler are abstract models, e.g., in the form of LTSs. A systematic approach for bridging between these very different levels of abstraction must be developed.

To address these challenges, several things must be accomplished.

- Information about the interface must be provided and represented. This information should at least include the interaction primitives exchanged with the peer, their parameters with associated types and data domains; otherwise the learning process can not be realized. We report on our achievements in Section 3.2
- To bridge between concrete component interfaces and abstract formal models utilized by learning and synthesis techniques, abstractions should be developed to relate different levels of abstraction. We report on our achievements in Section 3.3
- Additional information about the interfaces and data domains can be used to make the learning more efficient. These issues are reporet on throughout Sections 3.2 and 3.3.

3.1 Ontologies

We have found *ontologies* to be a suitable vehicle for representing information associated with these issues. Ontologies have recently become a central means to support user-level semantic descriptions on various levels, simply by means of intuitive classification schemes together with some relations describing mutual dependencies. In essence these descriptions can be seen as very high level type descriptions from the user-level perspective enriched by some knowledge base, additionally relating facts, concepts and entities.

Ontological knowledge can be exploited at different levels during the learning process: initially, at the interface level, in order to fix the learning alphabet, and during the evaluation of the equivalence queries in order to steer the search.

Practical learning and ontological descriptions are mutually supportive:

- Ontological descriptions related to the involved component interfaces form a good basis for building abstractions at an adequate level for the learning procedure. They may help to guide the learning enabler in exploring the *a priori* unknown parts of component behavior, in particular when searching for counter-examples as part of the so-called equivalence queries.
- Practical learning, on the other hand, may refine the existing ontological knowledge by experimentally revealing previously unknown structure in the component interface, like precedence or causality relations between interface primitives or activities, or like specific input/output patterns.

It is the nature of learning to neither be correct nor complete, but in a certain sense optimal with respect to the considered observations. It is therefore important to distinguish assured knowledge, as may be given in terms of interface descriptions or service level agreements, and hypothesis, as provided

by learning. This distinction should be indicated within the ontologies, as it is important when facing differences between the model and the running system. In these cases, hypothetical knowledge must simply be refined, whereas assured knowledge indicates some kind of (system) error.

3.2 Representing Information needed by the learning enabler

In this section, we describe the different types of information used by the learning enabler, and how they can be represented.

Interface Descriptions must be provided and represented, otherwise it is impossible to perform the learning process. An interface description should at least include the interaction primitives exchanged over the interface, their parameters with associated types and data domains. This information can be obtained in several ways:

- In the synthesis dry run and the in the PoTM case study this information was provided partly through documents provided alongside the services (as WSDL document). From these descriptions the functionality necessary to interact with the system could be generated automatically. Information about data values was taken manually from examples that came with the scenario descriptions.

- In the NASA Voyager case study information about data types was obtained by means of reflection from the tested entities (Java objects). Valuations used in the learning process were determined in a semi-automated trial-and-error fashion.

- In the case studies, where models of the SIP, TCP, and A-MLC protocols were learned, the interface descriptions were obtained from additional documentation: standard documents (IETF RFCs) in the case of SIP and TCP, and internal documentation in the case of A-MLC.

Information about data types (number, text, truth value etc.) and their ranges This information is needed for the learning algorithm in practice to be able to construct a finite abstraction on the input alphabet of the system to be learned. Type information is typically too coarse to serve as an adequate abstraction for the learning procedure. In such cases, being able to provide a corresponding finite (abstract) data domain is essential. Knowing for example only that some value is a textual type will be insufficient. However knowing that the domain is [jan,feb,mar,...] directly supports a systematic learning process. In the next section, we elaborate further on how this information is used when building abstractions.

Parameters' role or function This refers to information such as "parameter x is of type SOME", where "SOME" is just a (type)label. In this situation, relational knowledge about type compatibility assists in optimizing the search for adequate (action) primitives following a given output of the system: a corresponding ontology might relate input and output parameters to each other allowing one to steer the selection of the next membership query, and therefore to focus the search inherent in practical realizations of equivalence queries.

Information about dependencies between primitives (actions) Ontologies may also express relationships between whole (action) primitives such as (in)dependence, causality, and reversibility. Using information about reversibility (negative elements) can be exploited for building homing sequences, leading the system back to the initial state (the requirement to reset a system in its initial state is discussed in Chapter 2). Information about independence of actions and causality of actions may be used to reduce the amount of membership queries needed during learning [27].

3.3 Building Abstractions for the learning enabler

As stated in Chapter 1, we must take into consideration that learning can only be performed in terms of concrete interaction with a networked peer, whereas the desired output of the learning enabler is abstract models, e.g., in the form of Mealy machines or LTSs.

A systematic method for bridging between these levels is to develop means to represent the relation between different levels of abstraction. Such a relation is needed, not only for generating abstract models by the learning enabler, but also later when generating running code from an abstract specification of a synthesized connector. Fig. 3.1 shows the assumed stack of model transformation during the learning and synthesis process. Starting from the lower left (resp. right) with discovery of services the synthesis

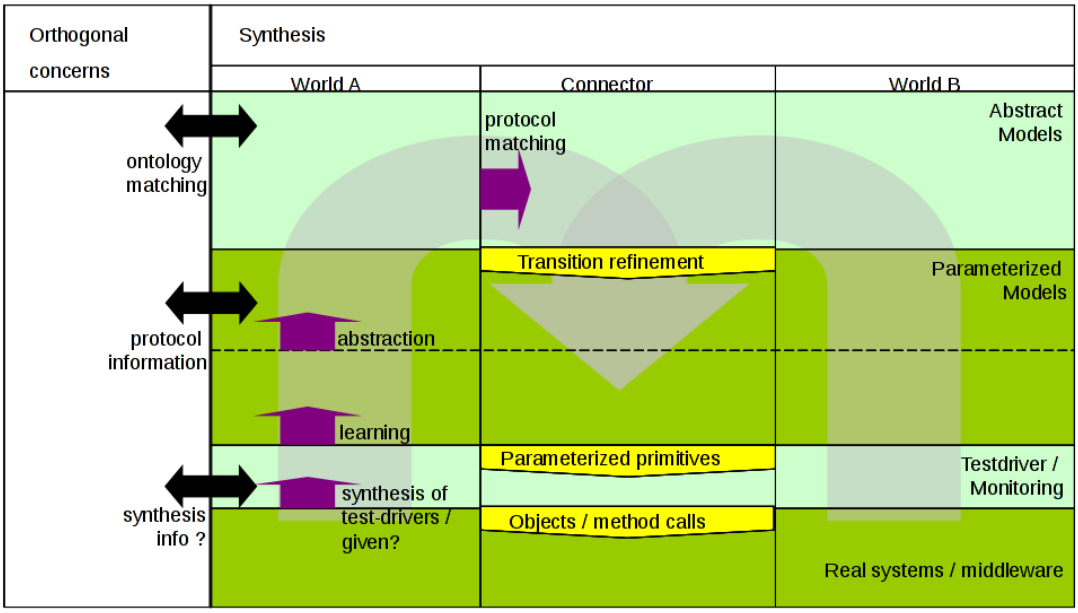


Figure 3.1: Protocol Transformation Stack

of connectors will rely on synthesizing test drivers from the information provided by discovery. These test drivers will provide the discovered interface to the upper levels and, on its downward side, operate the actual networked systems. They may be synthesized by means provided alongside standard technologies (e.g. WSDL binding). On top of these test drivers the learning enabler will produce several levels of abstractions resulting in abstract LTS models. The transition labels in the abstract models will correspond to primitives in the test drivers with concrete parameter valuations. These valuations may even be history dependent as learning may introduce state variables at the intermediate levels. At the (abstract) top of the stack connectors are synthesized via protocol matching. The derived abstract connector model must then be passed down the complete transformation stack in order to generate corresponding running code. This is indicated in the middle column of Fig. 3.1.

How are Abstractions Generated and Revised?

As stated, the relationship between different levels of abstraction in the protocol transformation stack of Fig. 3.1 are represented by adopting abstraction techniques. Abstraction techniques have been used in program analysis [16], and in formal verification [31, 14]. In contrast to abstraction for analysis and verification, however, we are now in a black-box setting, meaning that we do not have access to source code or formal models on which an abstraction can be straight-forwardly defined. One difference is that it is not equally easy to ascertain that the abstraction results in a (not too big) finite-state model. Instead,

building an abstraction will be part of the learning process. Typically, the learning process will start with an initial hypothesized abstraction, which may be revised as information about the component is revealed during the learning process.

A small Illustration In order to make the discussion more concrete, let us illustrate by a small example, taken from [1], also enclosed as an appendix. In this example, we infer the behavior of a protocol entity (called *SUT*) which services requests to set up a connection. To us, *SUT* is a black box, for which we have obtained an interface description which states that it communicates with its environment by receiving input messages of form *REQ*(id, sn) and *CONF*(id, sn), where the parameters id and sn are natural numbers, and transmitting messages of form *REPL*(id, sn), *ACK*(id, sn), or *REJ*.

In order to employ learning, we need to construct an abstraction between the interface of the *SUT* and a small finite alphabet. Ideally, the symbols in this small finite alphabet should represent equivalence classes of interface primitives of the *SUT*, which convey the same "meaning" on the interface. As an example, we could employ an abstract symbol of form *REQ*$(\mathrm{CUR}, \mathrm{CUR})$ to denote input symbols of form *REQ*(id, sn) where id is the identifier of the currently active connection, and sn is the lowest untransmitted sequence number. An *abstraction mapping* is used to represent the relation between these two sets of interface symbols. In Figure 3.2, we show how the abstraction mapping may translate between the two levels of abstraction.

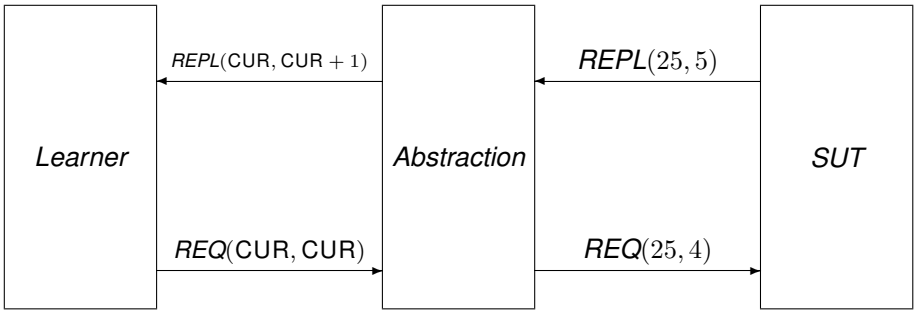


Figure 3.2: Introduction of *Abstraction* between *Learner* and *SUT*

Requirements on Abstractions It is important the abstraction employed reflects a natural classification of the concrete interface primitives. An important test is that the abstraction produces a deterministic interface. Intuitively, this requires that two concrete interface input symbols are mapped to the same abstract symbol only if they are "handled in the same way" by the component, i.e., alter the internal (control-)state of the component in the same way. (a formalization of this requirement appears in Section 5 of Appendix A.3). For example, in the example shown in Figure 3.2, the requirement would imply that whenever a message of form *REQ*(id, sn), where id is the identifier of the currently active connection and sn is the lowest untransmitted sequence number, is transmitted to the *SUT*, then the *SUT* will respond with a message of form *REPL*(id', sn'), where $id = id$ and $sn' = sn + 1$. If there are situations where it does not, techniques could be devised to refine the input abstraction appropriately, as discussed further below.

Building Abstractions Producing a suitable initial abstraction is obviously important for the success of learning. So far, we have approached this problem on a case-by-case basis using hand-tailored abstraction mappings.

- For common types of communication protocols, such as SIP and TCP, there are natural abstractions of parameters that represent, e.g., sequence numbers, identities of connections, sessions, etc. For identities of connections, abstractions typically only need to consider whether a connection is the one under consideration by the learning session. For sequence numbers, abstractions only need

to worry about the possibility to increment some counter (as, e.g., for SIP), whereas there are also cases with more complex flow control (as in TCP), where finding a good abstraction is more difficult.

- For web-services, initial abstractions can be built based on WSDL and additional heuristics that employ the defined types in the WSDL.

In subsequent work, we should aim to understand how to build good initial abstractions for different classes of applications.

Another concern is that in the work done so far, abstractions have been supplied by hand. It is desirable that the abstractions are automatically generated by the learning tool. We are approaching this problem in the same way as the problem of resolving nondeterminism (reported below). This means that whenever specific parameter values are relevant for the behavior of a module, we classify these values by adequate parameter/action refinement, which is delegated to algorithms tailored for the treatment of data. Our empirical results indicate the value of our general framework which allows the composition of complex learning processes from dedicated algorithms each with their specific application profile [26].

Revising Abstractions Typically abstractions as described above are maintained in mapping tables situated outside of the core learning algorithm, exposing only abstract symbols to the learner. Due to this opaque nature of the abstract mapping all modifications to the abstraction are driven outside of the inference process, often leading to manual modifications.

The employed active learning techniques assume a deterministic behavior, meaning in particular that this abstraction has to impose a deterministic behavior on the concrete system. This is a very strong requirement when dealing with black box systems and it would be highly beneficial to satisfy this requirement with automated means of abstraction-refinement.

We are working on a method to automatically refine a given abstraction until a level is reached where this abstraction imposes a deterministic behavior on the concrete system. Like automata learning itself, this method is in general neither sound nor complete, but it also enjoys similar convergence properties as long as the concrete system itself behaves deterministically. Key to this method is the switch of the learning scenario from the left configuration to the right configuration shown Fig. 3.3.

The left half of the figure shows the typical learning setup, that uses a fixed, hand-tailored abstraction mapper (cf. Fig 3.2). This setup suffers from the discussed flaws. The setup shown in the right half of the figure circumvents these problems by making the abstraction the dominating component. Thus the learning algorithm no longer relies on the correctness of the initially given abstraction and the abstraction may be refined during the course of learning. Technically this is achieved by a change of perspective: rather than working at the abstract level, the Learner is sitting now at the concrete level in order to observe the concrete system behavior for a set of representatives of the equivalence classes imposed by the abstraction. Thus abstraction is no longer a “filter” between the concrete system and the learning algorithm, but rather a teacher, helping the Learner to choose adequate representative tests. This Learner is able to automatically resolve *controllable* non-determinism, i.e., non-determinism which is due to the imposed abstraction, as long as all the entities of the concrete system are finite. Thus the control over abstraction becomes part of the learning process, with the effect that detected non-determinism does not lead to failure, but to a dynamic refinement of the abstraction.

Producing Compact models A problem is that existing regular inference techniques, even when employing abstraction, produce “flat” state machines, in which neither states nor transitions have the structure that would be used in a manually produced model. We have therefore experimented with techniques for restructuring the representation of an unstructured finite-state machine, in order to get less complex and better structured models. In [9], also included as Appendix A.4, we have employed an approach where the flat state machine is structured by means of a set of decision diagram from each control location. These are constructed by the ID3 Decision Tree generation algorithm. We have evaluated this technique by applying them to the Mobile Arts Advanced Mobile Location Center (A-MLC) protocol, which is a commercially available middleware protocol that allows mobile network operators to provide presence information from the GSM/UMTS network.

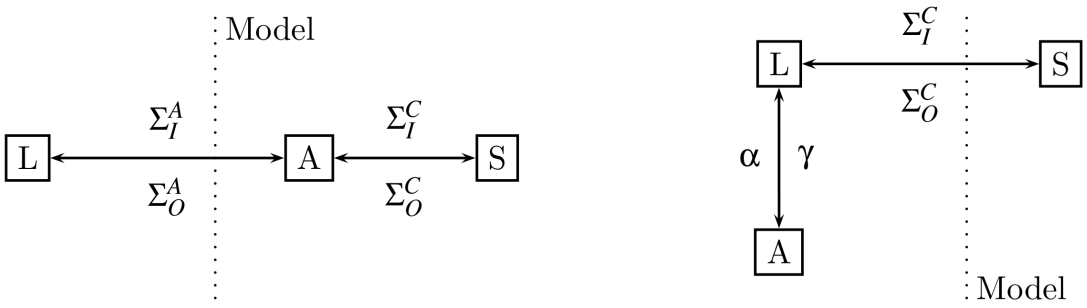


Figure 3.3: traditional use of abstraction (left) and abstraction as part of the learning process (right)

Summary During Year 1, we have spent significant effort on developing the use of abstraction in the context of automata learning: the conceptual foundations for the approach are under development, and several exploratory case studies have been performed. This provides a basis for further work, as outlined in Chapter 5.

4 Development of tools

LearnLib [39] is a library of tools for automata learning, which is explicitly designed for the systematic experimental analysis of the profile of available learning algorithms and corresponding optimizations.

Its modular structure allows flexible configuration of tailored learning scenarios, which exploit specific properties of their envisioned applications. Exploiting application-specific structural features enables optimizations that may lead to performance gains of several orders of magnitude, a necessary precondition to make automata learning applicable to realistic scenarios as those considered in the CONNECT project.

Work has been done on further extending the scope of modularization and configuration, converting LearnLib into a set of loosely coupled, reusable and easily interchangeable modules. Apart from allowing for easy setup of learning experiments this paves the way for advanced handling of abstraction within the learning framework.

In the context of CONNECT it is important to be able to adapt learning to a wide range of scenarios, each potentially challenging learning in different ways. Fixed learning strategies may prove to be inefficient when coping with parts of the scenario-landscape, thus the flexibility provided by the extended component-based approach should provide immediate benefits when dealing with CONNECT scenarios. The learning-solution should also be deployable on a wide range of systems and provide means to evaluate the efficiency of learning strategies, e.g., by providing statistics on runtime behavior.

4.1 Modernizing LearnLib

LearnLib originally was written in C++ for POSIX-compatible operating systems (such as Linux and Solaris), with dependencies on libraries only available on a limited number of systems. This has proven problematic as this restricts the environment in which LearnLib could successfully be deployed. Also of concern were issues with memory-management, as effective memory allocation (e.g., avoiding memory fragmentation) is very important to learn models of big systems.

Thus we ported LearnLib to the Java platform, ensuring that LearnLib is deployable on any system for which a Java runtime environment is available, including Windows and MacOS. Java provides a rich system library, which amongst other things provides highly optimized standard data structures, and supports multi-threading and web-services as core features, both of which are relevant to the CONNECT project for reasons of performance and connectivity.

Another emphasis was put on the ease of use and intuitive handling of the LearnLib in order to provide it for experimentation for the whole CONNECT consortium. This is important for CONNECT because it allows the other groups to better understand the concepts of automata learning.

4.2 Model-based learning

The extended component-based approach of the reengineered LearnLib enables easy integration into model-driven application building tools, like the Java Application Building Center (jABC). Within the jABC learning setups can be created and adapted to accommodate new scenarios (Fig. 4.1 shows a graphically composed learning process loaded into the jABC). This elevates LearnLib onto a new level, creating the Next Generation LearnLib. Integrated into Next Generation LearnLib are means for gathering and visualizing statistics on learning processes (Fig. 4.2 shows a diagram on the distribution of query lengths), which allows for apple-to-apple comparisons between learning setups and will help to evaluate learning strategies within CONNECT scenarios.

A complete learning system is usually composed of several components, some of which are optional while some are not:

- At the core of any learning system a learning algorithm has to be selected, fit to the scope of application chosen.

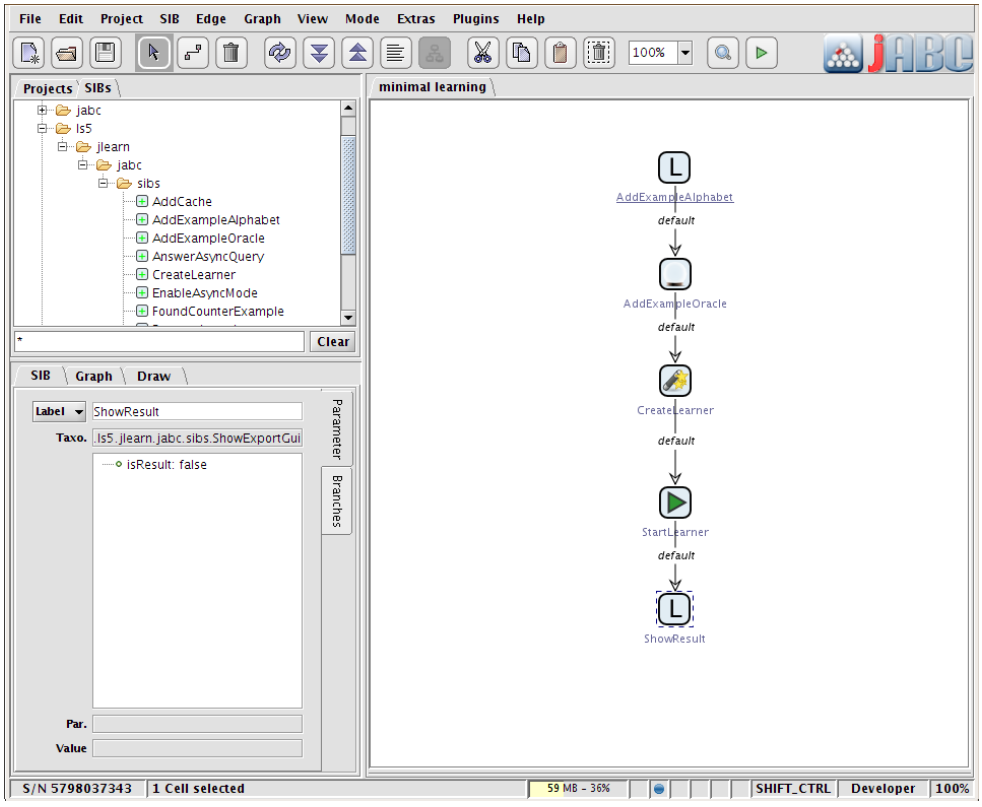


Figure 4.1: jABC with a minimal learning setup loaded

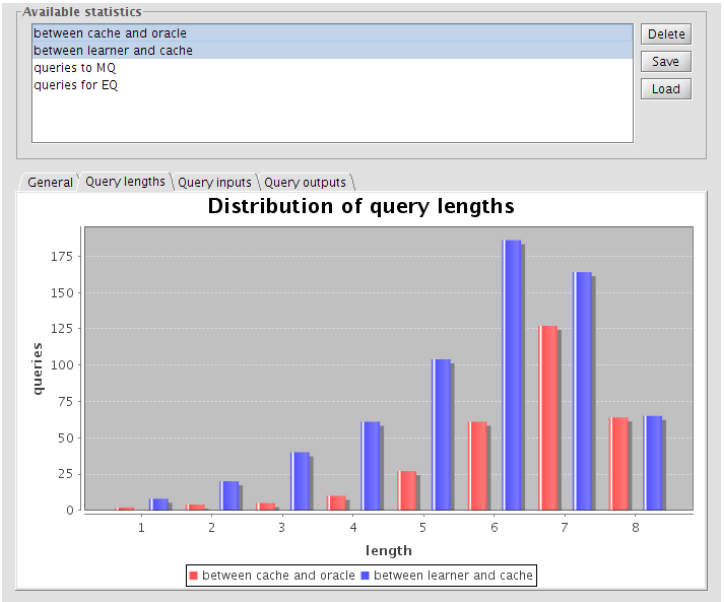


Figure 4.2: Visualization GUI for statistics on learning processes

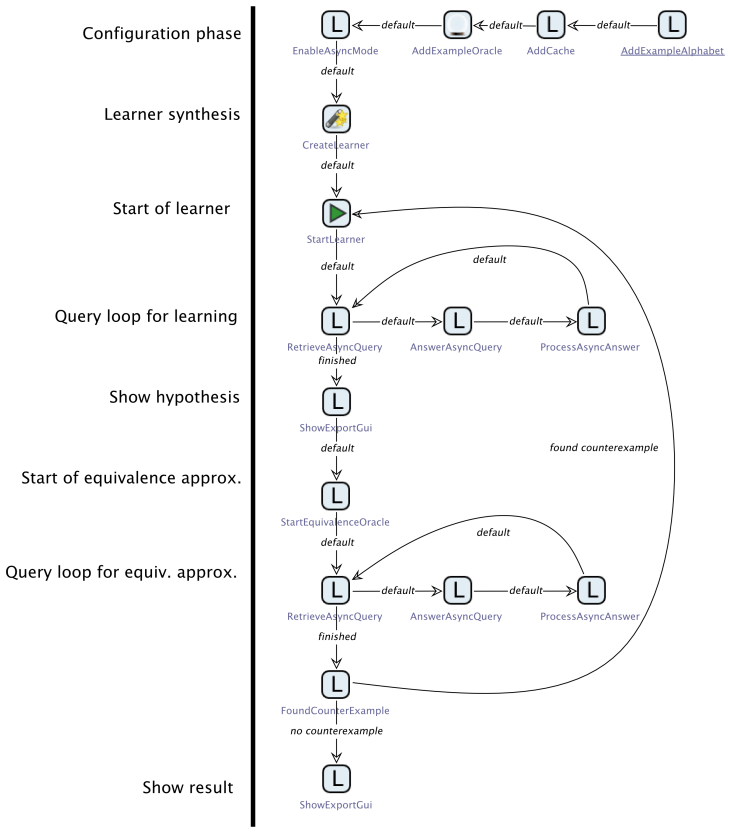


Figure 4.3: Structure of a generic learning process

- It is necessary to have means to connect to the system whose behavior is to be learned. In this context we call those connectors “oracles”, as they deliver answers (system responses) to questions (queries) generated by the learning algorithm.

- The number of queries an active learning algorithm generates can quickly reach millions. Depending on how fast system response can be generated it may be necessary to reduce the number of queries, which motivates the introduction of filters that answer queries using application-expert knowledge.

Many of those components are reusable in nature and should easily be employable in various application contexts. The Next Generation LearnLib exposes such components to the jABC, enabling engineers to compose application-fit learning experiments from easy-to-use building blocks.

When looking at the learning process (as shown in Fig. 4.3) one can easily identify recurring structures that are near-identical in many learning experiments: the learning setup undergoes a configuration phase, after which a learner is instantiated. After starting the learner instance queries are processed in a loop, involving system interaction. Once a hypothesis is constructed an approximate equivalence oracle is invoked, generating more queries that are processed in a loop. If the equivalence approximation found a counterexample disproving the hypothesis learning will resume via a loop to the top, otherwise learning is complete.

This general structure is the same for basically all learning experiments, thus modeling and editing those in jABC enables evolutionary development based on prior designs, maximizing reusability.

4.3 Conducted studies

The existing stack of learning technologies has already proven its usefulness for exploration of black-box-systems in case studies predating the CONNECT project.

In [38] dynamic testing is presented, a method that exploits automata learning to systematically test (black box) systems almost without prerequisites. Based on interface descriptions and optional sample test cases, this method successively explores the system under test (SUT), in order to extrapolate a behavioral model. This is in turn used to steer the further exploration process. Due to the applied learning technique, this method is optimal in the sense that the extrapolated models are most concise (i.e. state minimal) in consistently representing all the information gathered during the exploration.

This method can be elegantly combined with numerous optimizations of the learning procedure, with various choices of model structures, and with the option of dynamically/interactively enlarging the alphabet underlying the learning process. The latter is important in the Web context, where totally new situations may arise when following links, but is also relevant for other context, e.g., for protocols that negotiate client/server capabilities during a “connection handshake”.

For dynamic testing a study was conducted using the web application Mantis, a bug tracking system widely used in practice as was another case study demonstrating the scalability of the approach. It was shown that behavioral models arise that reveal the system structure from a user perspective. Besides steering the automatic exploration process, those models are ideal for user guidance and for improving system understanding. This is relevant in the context of CONNECT scenarios as “user perspective” in connectors often can be translated to “client perspective”, which reveals behavioral information on the involved counterpart of a client/server setup.

5 Future Plans

During Year 1, we have spent significant effort on understanding needs for learning, in order to develop its power and in order to fill the needs of the CONNECT context. Main directions for subsequent work are as below.

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

Priori to CONNECT, we proposed first ideas for dealing with such richer models [7, 8]. The proposed approaches however rely on plain automata learning. Rich models are constructed externally and not as part of the actual learning process. During the first year, we began to develop methods that are necessary to support rich models and can be tightly integrated with the learning process, as described in Chapter 3. From the current point of view the phenomena that will have to be integrated and will be (continued to be) studied are: abstract and local alphabets, variables and assignments, and parameterized symmetries.

Within a suitable general framework, we should develop appropriate specializations to handle learning for specific types of systems. Examples of such classes include communication protocols, which require a proper treatment of parameters, and web services, where associated metadata descriptions should be exploited by suitable techniques. These specializations should result in modules learning tools that are under development.

Handling QoS. The CONNECT framework also takes QoS parameters into account. As shown in Fig. 1.1 (in the introduction), the learning enabler is envisioned to be one source of data concerning the fulfillment of QoS properties. We will develop techniques for monitoring and learning QoS properties of interaction behavior.

Latency behavior can, in a first approximation, be modeled as an additional parameter that is supplied in inputs and outputs. Initial steps to consider this approach have already been proposed in [23, 22]. A satisfactory incorporation of QoS requires, however, a robust incorporation of **nondeterminism** into the learning framework. Nondeterminism is therefore one of the concepts that should be considered when extending automata learning during the second and third year of the project.

Monitoring to support re-learning. Observation of components behavior is inherently part of the overall learning process, therefore off-line monitoring is already used as a basic functionality that supports learning. While off-line monitoring could suffice in the initial learning phase, updating learned models once the CONNECTed system is in execution requires on-line monitoring techniques, which can be applied with minimal overhead to observe a running system that is already deployed in the field.

In CONNECT, on-line monitoring is implemented in WP5 as a cross-cutting functionality that will be realised according to a modular architecture, with a shared core (implementing generic monitoring features) and a set of specialised components that will provide specific support to activities including connector synthesis (WP3), dependability analysis (WP5), and, in WP4, re-learning (i.e., the update of existing learned models of NS).

In particular, in WP4, monitoring will contribute to bridge the gap between current learning approaches and the dynamic open world of evolving systems that is peculiar of the CONNECT project. Therefore, the key goal of WP4 research on monitoring in the next period of the project is to devise and implement mechanisms to observe running CONNECTed systems, in order to provide feedback to the learning enablers in such a way that existing learned behavioral models can be continuously updated, based on actual observations taken in the field.

Since efficiency is a major concern for on-line monitoring, we will investigate specific techniques that aim at minimizing overhead by allowing lossy observation, thus trading completeness of observation for efficiency, while preserving good detection power (i.e., while minimizing the probability of missing interesting

events).

Tool plans. It is planned to extend support for abstractions and parameters in the Next Generation LearnLib, taking advantage of the modular structure of the learning framework. We plan to provide an easily deployable “learning studio” available to project partners. Some central learning services may be provided using web-service technology, including hosting and maintenance. Tool-integration with monitoring tools is envisaged.

Bibliography

[1] F. Aarts, B. Jonsson, and J. Uijen. Generating models of communication protocols using regular inference with abstraction. Submitted, included as Appendix A.3.

[2] P. Abdulla, Y.-F. Chen, L. Holik, R. Mayr, and T. Vojnar. When simulation meets antichains (on checking language inclusion of NFA's). In *Proc. TACAS 2010*, 2010. to appear, available at http://user.it.uu.se/~parosh/publications/papers/tacas10.pdf.

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

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

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

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

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

[8] T. Berg, B. Jonsson, and H. Raffelt. Regular inference for state machines using domains with equality tests. In J. L. Fiadeiro and P. Inverardi, editors, *Proc. FASE '08,* 11^{th} *Int. Conf. on Fundamental Approaches to Software Engineering*, volume 4961 of *Lecture Notes in Computer Science*, pages 317–331. Springer, 2008.

[9] T. Bohlin, B. Jonsson, and S. Soleimanifard. Inferring compact models of communication protocol entities. In Preparation, included as appendix A.4.

[10] Y. Brun and M. D. Ernst. Finding latent code errors via machine learning over program executions. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 480–490, Washington, DC, USA, 2004. IEEE Computer Society.

[11] Y.-F. Chen, A. Farzan, E. M. Clarke, Y.-K. Tsay, and B.-Y. Wang. Learning minimal separating DFA's for compositional verification. In *Proc. TACAS '09,* 15^{th} *Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, volume 5505 of *Lecture Notes in Computer Science*, pages 31–45. Springer Verlag, 2009.

[12] T. S. Chow. Testing software design modeled by finite-state machines. *IEEE Transactions on Software Engineering*, 4(3):178–187, May 1978.

[13] E. Clarke, O. Grumberg, and D. Peled. *Model checking*. Springer, 1999.

[14] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. *Journal of the ACM*, 50(5):752–794, 2003.

[15] J. E. Cook and A. L. Wolf. Discovering models of software processes from event-based data. *ACM Trans. Softw. Eng. Methodol.*, 7(3):215–249, 1998.

[16] P. Cousot and R. Cousot. Abstract interpretation: A unified model for static analysis of programs by construction or approximation of fixpoints. In *Proc.* 4^{th} *ACM Symp. on Principles of Programming Languages*, pages 238–252, 1977.

[17] C. de la Higuera. A bibliographical study of grammatical inference. *Pattern Recognition*, 38(9):1332–1348, 2005.

[18] E. Elkind, B. Genest, D. Peled, and H. Qu. Grey-box checking. In *Proc. FORTE '06*, volume 4229 of *LNCS*, pages 420–435, 2006.

[19] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27:99–123, 2001.

[20] S. Fujiwara, G. von Bochmann, F. Khendek, M. Amalou, and A. Ghedamsi. Test selection based on finite state models. *IEEE Transactions on Software Engineering*, 17(6):591–603, 1991.

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

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

[23] O. Grinchtein, B. Jonsson, and M. Leucker. Learning of event-recording automata, 2010. To Appear in Theoretical Computer Science.

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

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

[26] F. Howar, B. Steffen, and M. Merten. A framework for hybrid automata learning. In progress.

[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] H. Hungar, O. Niese, and B. Steffen. Behavior-based model construction. *Springer International Journal of Software Tools for Technology Transfer*, 6(1):4–14, 2004.

[29] Information Sciences Institute, Universisy of Southern California. The network simulator - ns-2. http://www.isi.edu/nsnam/ns/index.html.

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

[31] C. Loiseaux, S. Graf, J. Sifakis, A. Boujjani, and S. Bensalem. Property preserving abstractions for the verification of concurrent systems. *Formal Methods in System Design*, 6(1):11–44, 1995.

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

[33] L. Mariani and M. Pezzè. A technique for verifying component-based software. In *International Workshop on Test and Analysis of Component Based Systems*, pages 17–30, Barcelona, Spain, March 27–28, 2004.

[34] O. Niese. An integrated approach to testing complex systems. Technical report, Dortmund University, 2003. Doctoral thesis.

[35] J. W. Nimmer and M. D. Ernst. Automatic generation of program specifications. *SIGSOFT Softw. Eng. Notes*, 27(4):229–239, 2002.

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

[37] H. Raffelt, T. Margaria, B. Steffen, and M. Merten. Hybrid test of web applications with webtest. In T. Bultan and T. Xie, editors, *TAV-WEB*, pages 1–7. ACM, 2008.

[38] H. Raffelt, M. Merten, B. Steffen, and T. Margaria. Dynamic testing via automata learning. *International Journal on Software Tools for Technology Transfer (STTT)*, 11(4):307–324, 2009.

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

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

[41] M. D. Wulf, L. Doyen, T. A. Henzinger, and J.-F. Raskin. Antichains: A new algorithm for checking universality of finite automata. In *Proc.* 18^{th} *Int. Conf. on Computer Aided Verification*, volume 4144 of *Lecture Notes in Computer Science*, pages 17–30. Springer Verlag, 2006.

A Case Studies

A.1 The NetTech (PoTM) Case Study

Imagine a web-service for mobile payment transactions. The “Pay On The Move” (PoTM) service provides the functionality to manage payments between customers and owners of web-shops. A company can login into the system, users (or, better, shops acting on the behalf of users) can begin transactions providing thereby wiring details for a payment and credentials for the used account. They may also suggest a transaction id. The service exposes two additional primitives: one to check the status of a transaction and one to let a bank confirm a transaction (in case the planned withdrawal is possible).

Assume this service runs on an infrastructure that is to be extended by some high availability features. Now it shall be proven that enabling the new features does not have any impact on the service’s functional behavior in order to guarantee a seamless update. As users of the service are only provided with the services interface description (in form of a WSDL document, an excerpt of which is shown below) and some scarce explanation of the service’s intention, a model generated only using this information will resemble the behavior experienced by users of the service.

```
...
<xs:complexType name="Login">
 <xs:sequence>
  <xs:element minOccurs="0" name="companyName" type="xs:string"/>
  <xs:element minOccurs="0" name="companyPassword" type="xs:string"/>
 </xs:sequence>
</xs:complexType>
<xs:complexType name="BeginTransaction">
 <xs:sequence>
  <xs:element minOccurs="0" name="companyName" type="xs:string"/>
  <xs:element minOccurs="0" name="companyPassword" type="xs:string"/>
  <xs:element minOccurs="0" name="details" type="tns:transactionDetails"/>
 </xs:sequence>
</xs:complexType>
<xs:complexType name="transactionDetails">
 <xs:sequence>
  ...
  <xs:element minOccurs="0" name="customerPassword" type="xs:string"/>
  <xs:element minOccurs="0" name="customerUsername" type="xs:string"/>
  <xs:element minOccurs="0" name="externalTransactionId" type="xs:string"/>
  <xs:element minOccurs="0" name="paymentType" type="xs:string"/>
  <xs:element minOccurs="0" name="transactionId" type="xs:string"/>
 </xs:sequence>
</xs:complexType>
<xs:complexType name="CheckTransactionStatus">
 <xs:sequence>
  <xs:element minOccurs="0" name="transactionId" type="xs:string"/>
 </xs:sequence>
</xs:complexType>
<xs:complexType name="AknowledgeTransaction">
 <xs:sequence>
  <xs:element minOccurs="0" name="transactionId" type="xs:string"/>
  <xs:element minOccurs="0" name="transactionRef" type="xs:string"/>
 </xs:sequence>
</xs:complexType>
...
```

To verify functional equivalence two test environments were created, one running the original version of the service and one running the service with the additional high availability features enabled. The experiments were performed in an active black box learning approach against the web-service interface. To ensure the absence of feature related impacts on functional behavior several aspects of the web-service (internal logical consistency, conformance between old and new service) were tested.

A.1.1 Experiments goals and hypothesis

The case study aims at several goals. The main target was the verification of the assumed absence of functional impacts due to cross cutting high availability technology in the PoTM service. Apart from that the design of the case study can also be understood as a prototypical application of learning and model checking in the situation of service integrators, that use off-the-shelf or closed source third party components, and must provide the (provably correct) integration of the service with possibly other such services or own developments, provided only with the mere interface description and some intuition about the provided functionality. To verify the functional equality of the two variants of the PoTM service, different aspects of both instantiations were inferred and compared to each other with respect to conformance. The aspects examined were:

- model inference
  The functional behavior of the services was extrapolated by active learning techniques. To apply those, some modifications to the original theoretical approach had to been made (see below).
- conformance of models
  The models of the different versions of the PoTM service were tested for conformance. Conformance was verified by testing the obtained models for isomorphism.
- internal logical consistency
  The inferred models where searched for unwanted / unexpected behavior by means of model checking techniques.

For the active learning we use “LearnLib”. To connect the web-service under test to LearnLib we generated a web-service client from the provided WSDL and wrote a test driver that translates LearnLib’s queries into invocations of the service’s primitives and routes them through the generated client. The resulting setup, which resembles the one outlined in Section 2.2, is shown in Fig. A.1. The learning algorithm generates queries using an abstract language. The test driver translates those into actual actions on the system under test and records the system’s reactions when exposing it to the actions. The inferred models were processed with a graphical modeling and model checking tool.

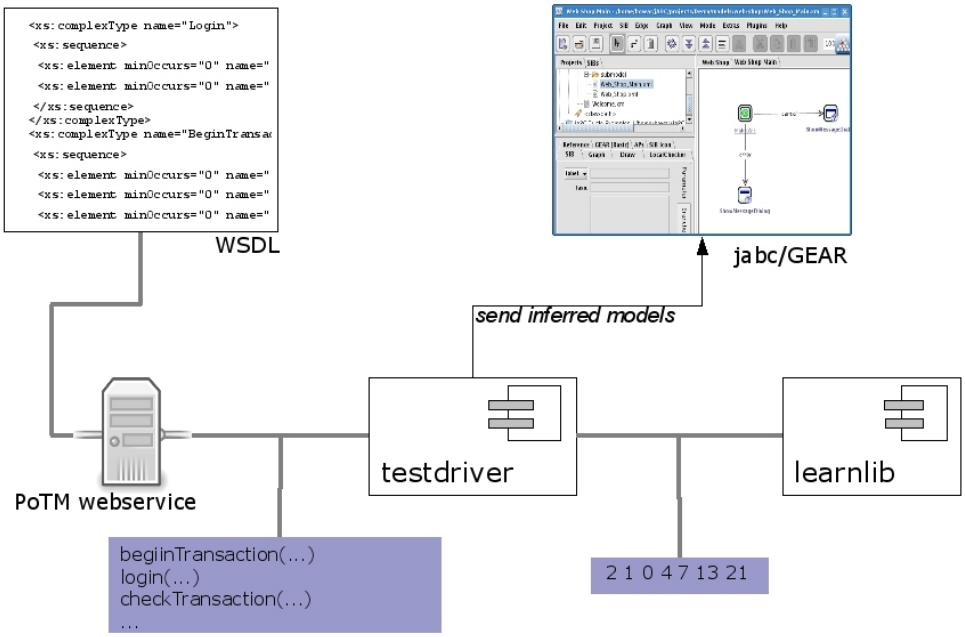


Figure A.1: Experimental setup for learning the PoTM service

A.1.2 Methodology

Handling Data Values One of the key problems when dealing with systems is that primitives typically have data parameters. To learn the service we need to provide an alphabet of the system under analysis. This is the set of input actions from which input sequences are formed. In presence of parameters the set of actions that can be performed on the system is the cross product of the different valuations of each primitive's parameters. The sets of possible valuations for some of the parameters are very big (e.g., in the case of character sequences it is exponential in the length of the word). As the number of queries needed to infer a system under test linearly depends on the size of the input alphabet, it is not feasible to test each input symbol.

We therefore used symbolic alphabets whose letters are (state-aware) translated into concrete symbols - and back. The abstraction mapping (cf. Section 3.3) was constructed manually. The login primitive for example had two parameters, which represent a pair of username and password. So we introduced an abstract `login(c)` symbol (where $c \in$ `valid, invalid`). During the execution of the experiments the two possible symbolic valuations of c are replaced by the test driver by known concrete valuations with the corresponding properties (i.e., granting access or not). For the `login(c)` primitive the needed valuations are taken from piece of example source code provided with the service. The used credentials are the same in all experiments.

Handling Reset For the symbols that require a unique transaction id, finding an adequate abstraction is a bit more complicated. Learning relies on the possibility of running numerous experiments on a system under test and requires that in each experiment the system's behavior is consistent with all other experiments. This is typically achieved by means of some kind of reset-mechanism in research setups. The existence of such a mechanism however is not typical in real-world scenarios. As in this case we did not have any way of accessing the service other than via the given web-service interface, we simulated resetting the system in the following way. A relaxed version of the requirement to reset the system into "the same state" is to reset the system into "an observationally equivalent state". The absence of differences in observable behavior has to include not only the state the system assumes at reset but has also to hold for all states possibly reached during experiments. As the service has no primitives to add, modify or delete users and companies, it may be assumed that the used valuations for those parameters will behave in the same way over time. Thus the behavior of the systems can only be different for the same (or different) transaction identifiers in different experiments (say, if one tries to use the same transaction identifier twice). The relaxation can thus be achieved by using "fresh" transaction identifiers in every experiment. Fresh here means transactions identifiers that have not been used in the system so far. Freshness was tested using the `checkTransaction(tid)` primitive which returns differently for used and unused transaction identifiers and does not change the state of the system (at least as far as we could observe).

Table A.1: abstraction used for the transaction details

Parameter	Type	Abstract Value	Concrete representative
amount	float	-	1.0
charge type	int	-	2
client ip	string	-	hermes
customer email	string	-	test@test.de
customer full name	string	-	Mustermann
customer mobile	string	-	0191058989
customer password	string	valid, invalid	test1, test2
customer username	string	-	test1
payment type	string	valid, invalid	8826a7bc-ff6e-4e06-8ef3-26dad264dfbd,-
transaction id	string	TID	*context dependent*

Levels Of Abstraction The valuations used for the details-parameter of the beginTransaction primitive are taken partly from the same piece of source code that already contained the company credentials, and they are partly chosen experimentally, in a trial-and-error kind of approach. They are chosen so that every valuation would reveal different behavior. The abstract and concrete valuations chosen are shown in Table A.1. This results in the following abstract set of parameterized input actions.

$$\begin{aligned} begin_{a1}(c, d1, d2, d3) c &\in \{valid, invalid\} && companycred. \\ d1 &\in \{valid, invalid\} && userpasswd. \\ d2 &\in \{valid, invalid\} && paymenttype \\ d3 &\in TID \\ login(c) c &\in \{valid, invalid\} && companycred. \\ check(t) t &\in TID \\ ack(t) t &\in TID \end{aligned}$$

The abstract input alphabet is the set of possible valuations of these actions. As is seen easily, there are 12 different valuations if the set TID of transaction identifiers is of size one. For $|TID| = 2$ it are 22 possible valuations. To study the influence of the alphabet size on the complexity of the approach we use a second abstraction in which we further restrict the allowed valuations of the beginTransaction's parameters: $begin_{a2}(c, t) = begin_{a1}(c, valid, valid, t)$ As during the first experiments the login primitive produced reflexive transitions only, we removed the corresponding action from the second tested abstract alphabet. This leads to the following two sets of abstract actions.

$$A1 = begin_{a1}(c, d1, d2, d3), check(t), ack(t), login(c) A2 = begin_{a2}(c, t), check(t), ack(t)$$

For each set of actions we did the learning twice, (once with $|TID| = 1$ and once with $|TID| = 2$). All four setups are run against both the original and the new service.

As we are dealing here with a real black-box scenario the equivalence queries have to be approximated. We did this using the W-method conformance tests [12], for which we choose the upper bound on the number of states to be the size of the conjecture (i.e. intermediate hypothesis) plus two. We use the LearnLib's learning algorithm for Mealy machines, which can be understood as performing a W-method test with a "plus one" bound during each learning phase already. This on the one hand reduces the number of equivalence queries, but on the other hand leads to more membership queries during the learning phase.

A.1.3 Results

The key parameters of the four different learning setups are shown in Table A.2, Where "abstraction" denotes the used abstract alphabet, "tid" the number of different transaction identifiers per experiment, $|\Sigma|$ and $|\Omega|$ the sizes of the input- and the output-alphabet, "states" the number of states the learned model has and "mqs" and "eqs" the number of the membership queries and equivalence queries performed to infer the model.

Table A.2: key parameters of the learning experiments on the PoTM

abstraction	tids	$\|\Sigma\|$	$\|\Omega\|$	states	mqs	eqs
A1	2	22	8	4	1.936	1
A1	1	12	8	2	288	1
A2	2	8	4	4	256	1
A2	1	4	4	2	32	1

In all four cases the obtained models are very small. In the setups with only one (abstract) transaction identifier the learned models have two states, in the setups with two transaction identifiers the models consist of four states. In all setups only one conformance test was performed at the end of the learning and none found a counterexample.

Fig. A.2 shows a graphical representation of the model the learning algorithm created exploring the payment service for abstraction level A2 and using one transaction id. The transitions are labeled with the according input and output symbols. The shown output symbols are the return values of the primitives' invocations. A return value of "1002" denotes a successful return (thus when using models for model checking this value was translated to "ok"). In the case of an error or exception the cause given by the web-service is shown as output-symbol.

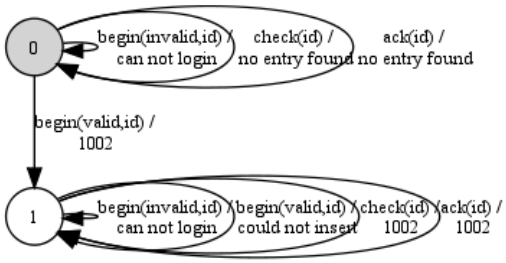


Figure A.2: Inferred model of the "plain" PoTM service [A2, 1 tid]

According to the obtained model the web-service behaves as follows: using the checkTransaction or acknowledgeTransaction primitive on a transaction identifier that is unknown to the system will result in an error ("no entry found"). Using invalid credentials will result in "can not login". A beginTransaction using an unknown transaction identifier and valid credentials will return "1002" (resp. "ok"). Using the checkTransaction or acknowledgeTransaction primitive with a transaction identifier that is already known to the system will return "ok" while trying to begin a new transaction with an already known identifier will return 'could not insert'. The models for abstraction level A1 exhibit the same behavior. All parameter valuations that were not used for abstraction level A2 result in reflexive transitions. Especially for the login primitive this is quite strange as it obviously has no functional impact on the system. The only parameters of this primitive are company credentials (a pair of username and password). No useful (state influencing) data is passed to the web-service. Fig. A.3 shows the model inferred for abstraction level A1 using two transaction identifiers. In the Figure only numerical labels are used for input- and output-symbols.

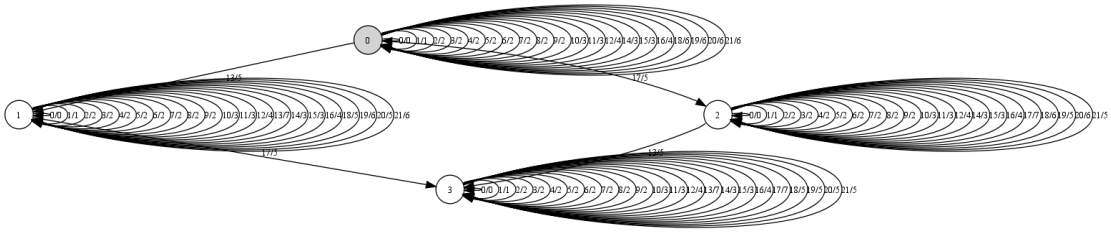


Figure A.3: Inferred model of the 'plain' PoTM service [A1, 2 tids]

Internal logical consistency It is easy to see that by starting a new transaction with valid information and an unused transaction identifier the state of the system changes permanently - there is no transition that leads back to the initial state. The same holds for the other models. This however contradicts the interpretation of the service's semantics given above.

The missing of a roll back in the models could have been due to the missing ability to cover time controlled behavior properly by state of the art automata learning techniques. To rule out this possibility, we performed a series of experiments focusing timeout behavior (as written above). The results are discussed below.

In this case the strange respectively unwanted behavior was easy to locate. One would not have to perform over 200 invocations of the service's primitives (as our learning required) to be sure that this does not work as expected. In larger scale scenarios however failure might not be so easy to identify. For example, we ran a series of learning experiments on a parametric network router which led to a learned model with over 22.000 states: searching for unwanted or unexpected behavior in models of this size can not be accomplished without tools. This is then the right situation to apply formal methods. Testing properties on the learned system for instance can be done by model checking.

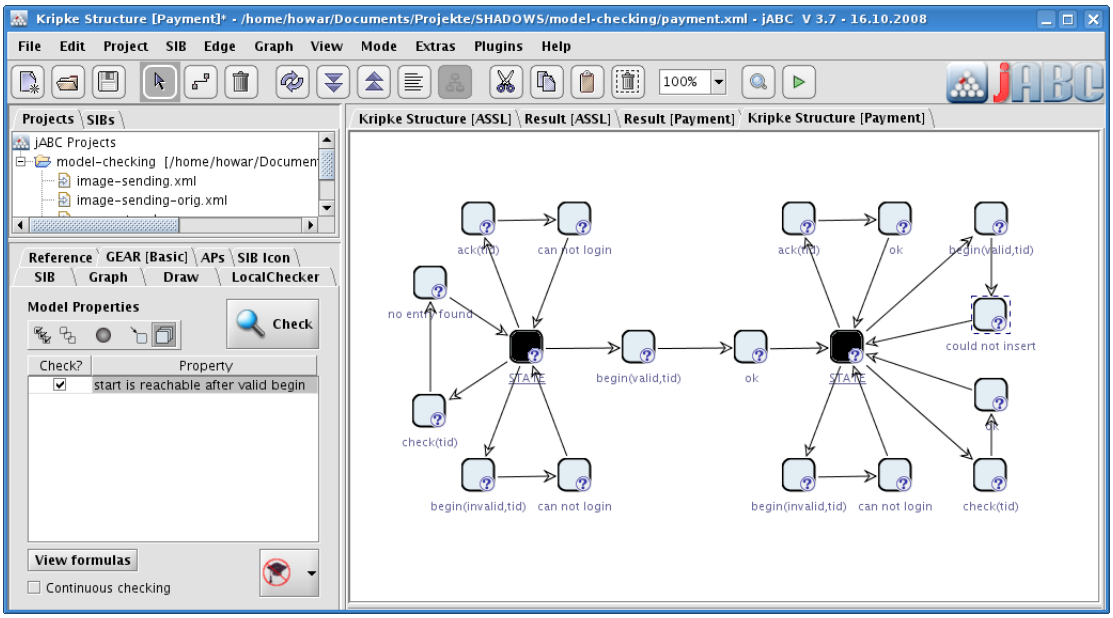


Figure A.4: Kripke representation of the PoTM model [A2, 1 tid] in GEAR

To apply model checking we used the jABC and its integrated model checker GEAR. The learned models were imported into the jABC and then transformed into Kripke structures, which is shown in Fig. A.4. To ease the readability, all the transitions of the learned Mealy automata are represented finer granularly, with two nodes (for the input and output symbols, respectively), carrying as atomic propositions the names of the alphabet-symbols. States with input symbols are additionally labeled valid or invalid depending on the concrete valuation of the symbol's parameters in that location. The initial state has the atomic proposition "start". The (in our case study) missing discard or timeout functionality can then be proven by the following CTL [13] formula.

$$((begin \vee valid \vee EXok) \vee (EFstart))$$

The former half holds for all valid beginTransaction input symbols from which in the next step an (output) state with the proposition "ok" is reached. The latter half holds for all states from which the initial state can be reached. The jABC shows which states of the Kripke structure satisfy a (sub)-formula by selecting the (part of the) formula: Fig. A.5 shows all states that satisfy the latter half of the formula.

When confronted with the resulting model, the provider of the web-service confirmed that the reason for the unexpected behavior was due to the test setup. This was configured to acknowledge transactions immediately after they are started.

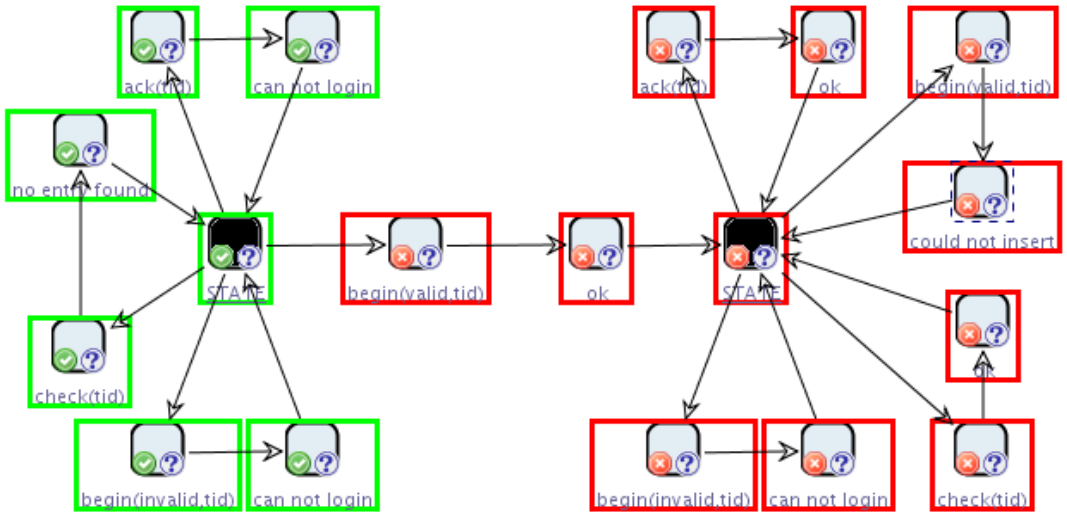


Figure A.5: Kripke structure, colored according to a formula

Conformance of original and new service For no experiment there was a difference in the observable behavior between the two implementations of the web-service. The absence of observable differences in the behavior is proved by testing the models of both services for isomorphism. Due to the small sizes of the models this could be done without the help of any tool. The models obtained from the two versions of the web-service being isomorphic is to be understood as a passed regression test: enabling the high availability technology had no influence on the application's functional behavior (which is what is exposed by learning).

A.1.4 Discussion

Though this is only a very small scenario in which most of the results achieved by learning (and model checking) could also have been achieved manually, there are some lessons learned here. It was proven (under the always to be made constraints) the conformance of the original and the new version of the PoTM web-service. The constraints to be named are:

1. active learning relies on equivalence queries which can only be approximated in practice. The value of the results thus generally depends on the confidence level of the performed equivalence query approximation. The exact confidence level however can only be determined with respect to an upper bound on the number of states the system under test may have. We chose an upper bound that seems sound in the given case study. With respect to this bound the confidence level of the performed approximation is 100%.

2. the abstraction we introduced was chosen carefully and validated in a series of pre-experiments to reveal the complete behavioral model of the web-service. As for the equivalence queries this has to be considered an approximation. It may well be that some parameter valuation leads to behavior the abstraction we used does not cover. In general abstractions made manually will rely on profound domain knowledge with an idea of what the goal of the performed experiments is.

Some other insights may be drawn from the case study: networked components exposing an accessible interface can be dealt with as black boxes (at least regarding the functional properties) that lack any kind of reset mechanism. The problem of missing reset functionality may be addressed in two ways: (1) by setting up services in learning-enabled environments which add the missing functionality on top of

the real service, (2) by finding ways around the hard reset requirement as the “not observable different” relaxation we introduced.

While deploying existent real life web-services in a special environment before learning seems quite expensive, using “fresh” values in every experiment may be impractical in some cases. Though the correct decision regarding this will always depend on the scenario, the second approach is probably more light weighted and more flexible. As web-services will often be designed to provide the same functionality to different, independent users there seems to be a fair chance of getting along with the approach taken here quite well.

The effects of the different levels of abstractions on the number of queries needed can be read from Table A.5. The number of parameters has an influence on the number of queries needed that is exponential. When choosing the right level of abstraction the trade off between making things too expensive to be dealt with and missing relevant behavior has to be considered. One big step forward will be moving the responsibility for supplying a good abstraction refinement from the user to the learning algorithm. The learning algorithm will then incrementally increase the number of abstract valuations by exploring and partitioning the concrete parameter-space in whatever fashion.

A.2 The Voyager Case Study

This section presents the learning experiments performed on the ASSL Voyager Case Study. The experiments were performed in an active learning approach on the Java binaries generated from the ASSL specification of NASA’s Voyager mission.

A.2.1 ASSL Voyager

ASSL is a language designed for the specification of systems of autonomous agents. It is developed by NASA and intended to be used for describing future space missions. For our case study we were provided with (1) a formal specification of NASA’s voyager mission re-modeled in ASSL and with (2) a corresponding set of java classes generated by an ASSL-to-Java code generator. Fig. A.6 contains a detail of the specification.

```
AESELF_MANAGEMENT {
  OTHER_POLICIES {
    POLICY IMAGE_PROCESSING {
      FLUENT inTakingPicture {
        INITIATED_BY { EVENTS.timeToTakePicture }
        TERMINATED_BY { EVENTS.pictureTaken }
      }
      FLUENT inProcessingPicturePixels {
        INITIATED_BY { EVENTS.pictureTaken }
        TERMINATED_BY { EVENTS.pictureProcessed }
      }
      MAPPING {
        CONDITIONS { inTakingPicture }
        DO_ACTIONS { ACTIONS.takePicture }
      }
      MAPPING {
        CONDITIONS { inProcessingPicturePixels }
        DO_ACTIONS { ACTIONS.processPicture }
      }
    }
  }
} // AESELF_MANAGEMENT
```

```
FLUENT inStartingGreenImageSession {
  INITIATED_BY { EVENTS.
            greenImageSessionIsAboutToStart }
  TERMINATED_BY { EVENTS.
            imageSessionStartedGreen }
}
FLUENT inCollectingImagePixelsBlue {
  INITIATED_BY { EVENTS.imageSessionStartedBlue }
  TERMINATED_BY { EVENTS.imageSessionEndedBlue }
}


EVENT greenImageSessionIsAboutToStart {
  ACTIVATION { SENT { AES.Voyager.
    AEIP.MESSAGES.msgGreenSessionBeginAus } }
}
EVENT imageSessionStartedBlue {
  ACTIVATION { RECEIVED { AES.Voyager.
    AEIP.MESSAGES.msgBlueSessionBeginAus } }
}
```

Figure A.6: Detail from the ASSL document specifying the Voyager mission

The ASSL description specifies four antennas (located on earth) and the voyager spacecraft. The spacecraft is equipped with two cameras, a narrow angle one and wide angle one. Every time the space craft passes by interesting objects or at least every 60 seconds it will take a picture with one of the cameras (the cameras are used in turn). The taken picture will then color by color and pixel by pixel be sent to each

of the antennas separately. After receiving the complete picture an antenna will send a message to the mission control center indicating the arrival of a new picture. All communication is realized through channels in a publish-subscribe manner: the antennas subscribe to the channel the voyager publishes to and then they publish to a channel mission control subscribes to. The specified mission is shown in Fig. A.7.

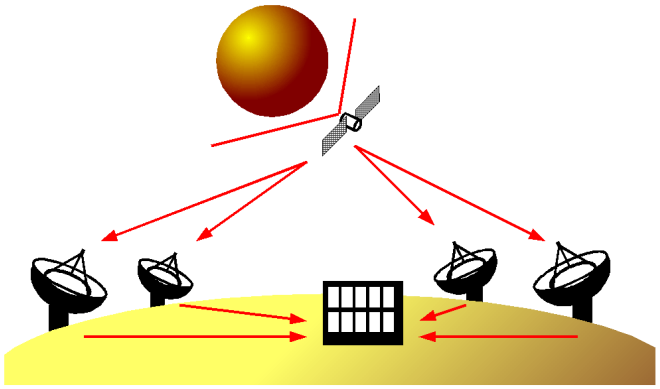


Figure A.7: Voyager mission (schematic)

A.2.2 Experiments goals and hypothesis

In this scenario the combination of learning and model checking can be used to study several things: one may search for unexpected behavior allowed by the specification by means of finding such behavior in the implementation. Drawing conclusions about the specification whilst looking at the implementation however implicitly makes the assumption that the implementation is a valid refinement of the specification. Thus testing the conformance of specification and implementation will be the second thing one is interested in. In the particular setup of this case study unconformant behavior will imply errors in the code generator. A third point of interest may be testing of the integratability of the different components of the system. This lead to the following experiments:

- *model inference*
  The functional behavior of the specified components was extrapolated by means of active learning. The problem of inferring Java code lead to some technical questions (see below).
- *conformance of models and specification*
  The models of the components were tested for conformance to the specification. Conformance was verified by going through models and specification manually.
- *internal logical consistency*
  The inferred models where searched for unwanted / unexpected behavior by means of model checking techniques.

To apply active learning techniques in the given scenario we used code instrumentation which however has its limitations (see PoTM Case Study). To test the advantages and potential of a heavyweight approach to tracing we developed a test driver basing on an extended Java virtual machine (JikesRVM).

To connect the system under test to LearnLib, which provides the means for learning for this case study, we generated test drivers that translate LearnLib's queries into invocations of the modeled entities' methods and that monitor the output of the system.

A.2.3 Methodology

Viewed from the learning perspective, this scenario comes with several challenges. The first challenge is to infer the behavior of Java code in general. It is not immediately clear how the notion of input and output alphabet is to be transferred to the learning of Java programs. As the ASSL code generator places each modeled entity (as cameras, spacecrafts, messages, channels etc.) into separate Java singletons which communicate by method invocation, it seemed adequate to use method invocations as input and output symbols. For input symbols this idea is quite easy to implement: after instantiating the system under test the learner has to get hold of a set of references to methods it may invoke as inputs.

In the case of output symbols we needed a technique for tracing all method invocations throughout the tested code. Such “cross cutting concerns” are well addressed by aspect oriented programming resp. code instrumentation. As stated above we used two slightly different approaches here. A lightweight using aspectj and a heavyweight one based using JikesRVM.

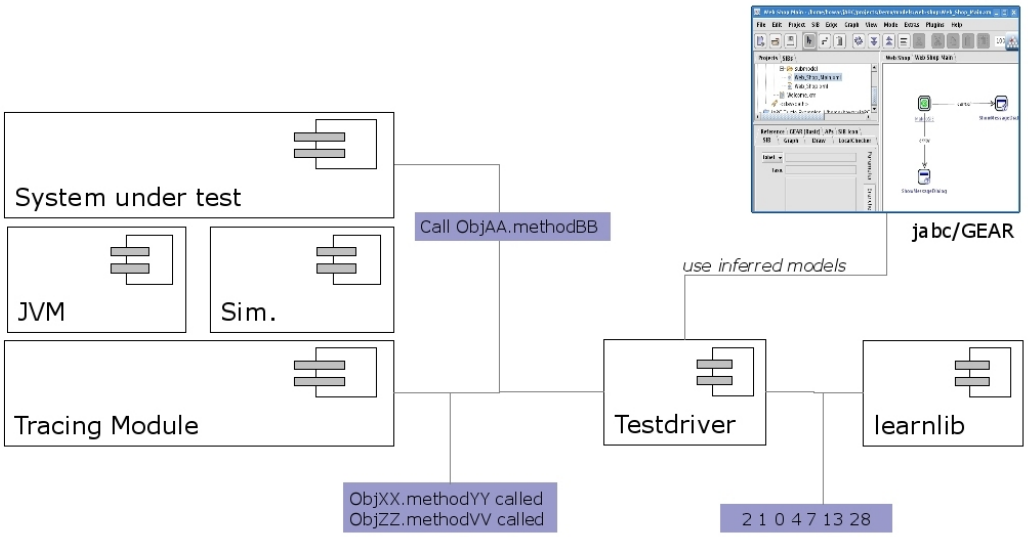


Figure A.8: schematic view of the experimentalsetup

The third challenge was the intense use of the singleton design pattern and multithreading in the code generated by the ASSL code generator. Extensive use of singletons in Java becomes a problem when separate copies of the system under test are instantiated for each experiment to be run. This however is a Java related issue. Singletons (static references) and every object referenced by one are never garbage collected, which pretty quick leads to lack of free memory. There are two possible solutions to this. One can either run each experiment in its own process or run all experiments on the same system under test. While the former is quite expensive (as every instantiation of the system under test costs several seconds) the latter requires a mechanism to reset the system into the initial state. We went for the latter approach and developed a homing sequence (of input actions) that would put the system into the initial state. The homing sequence essentially consisted of a sequence of actions consuming all the messages in the channels used by the modeled entities and leaving all components in their initial states.

This lead to the following experimental setup (see Fig. A.8): LearnLib was connected to the modified java virtual machine which on start would boot up the system under test and some test driver. The test driver would then stimulate the system under test on behalf of LearnLib, while the virtual machine monitored the call stack. In case a method assigned to an output symbol was called, the test driver would be informed and would at the end of each experiment report back to LearnLib.

The possibility to trace the complete call hierarchy makes this setup more grey-box than black-box.

It makes it possible to view parts of the otherwise hidden inner-workings of the components. In ASSL the hidden states of components for instance are designed as “fluents”. Those can be activated and terminated by external or internal events and may while active lead to some action. Monitoring calls of the methods of the fluents may then lead to a more detailed picture of components (fluent-level) which should be a valid refinement of a coarser model. We ran experiments with different levels of detail against one of the antennas. On the coarser level we would only monitor method calls that represent the insertion of messages into one of the channels (message-level).

Table A.3: stimuli used on different levels of abstraction

Entity	Input Symbols	Output Symbols
antenna (fluent-level)	$msgBegin_c$ $msgEnd_c$ $msgPixel_c$	$fluentInit_c$ $fluentDone_c$ $msgImageComplete$
Multiple antennas (message-level)	$msgBegin_{a,c}$ $msgEnd_{a,c}$ $msgPixel_{a,c}$	$msgImageComplete_a$
Voyager (message-level)	$timeToTakePicture$	$msgBegin_{a,c}$ $msgEnd_{a,c}$ $msgPixel_{a,c}$

$c \in \{red, blue, green\}$, $a \in \{aus, jpn, cal, spa\}$

The behavior of the voyager spacecraft was inferred only on the message-level. As the voyager has only one stimulus, which on the message-level changes the state anytime it is used, there is no chance to get a more detailed model by taking into account the inner-workings. To verify that the four antennas work completely independent from each other we did set up one scenario with two parallel antennas (for used stimuli see Table A.3).

The ASSL-to-Java code generator makes extensive use of threads. Each autonomous entity and most of its components are put into separate threads. Each thread runs its own main loop in which taking turns the application’s logic is run and the thread is paused for a fixed amount of time. To examine if timing and scheduling have any impact on the system under test the fluent-level version was run with simulated threads and a fixed scheduling among the simulated threads. All other experiments were run with real threads and the virtual machine’s scheduler.

As the Voyager’s output symbols are the antennas input symbols, it is not reasonable to test the Voyager together with the antennas. This leads to experiments shown in Table A.4.

Table A.4: performed experiments

Experiment	Entities	Level	Test driver
A_1	Voyager	message-level	JikesRVM
A_2	Antenna Australia	message-level	JikesRVM
A_3	Antenna Australia	fluent-level	simulated threads / man trace
A_4	Antennas Aus+Spa	message-level	JikesRVM

A.2.4 Results

Performance results for all experiments are shown in Table A.5. $|\Sigma|$ and $|\Omega|$ denote the sizes of the input- and the output-alphabet respectively, “states” denotes the number of states the learned model has. “mqs” and “eqs” indicate the number of the membership queries and equivalence queries performed to infer the model, respectively.

Table A.5: performance results for performed experiments

Experiment	$\|\Sigma\|$	$\|\Omega\|$	states	mqs	eqs
A_1	1	2	2	3	1
A_2	7	2	3	98	1
A_3	7	10	8	490	1
A_4	13	3	9	1.521	1

Model inference None of the experiments required more than one equivalence query. The model for experiment A_2 (one antenna, message level) is shown in Fig. A.9.

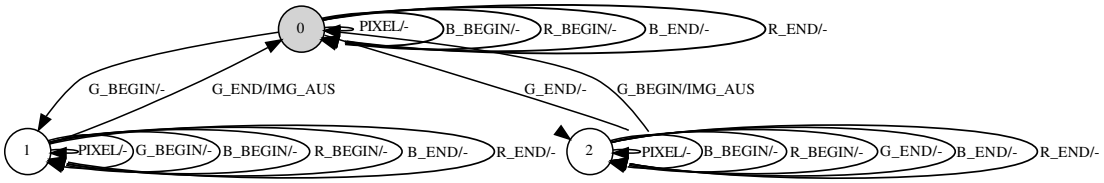


Figure A.9: Model from experiment A_2 (Antenna Aus., message-level, all primitives)

Due to the quadratic influence of the number of input symbols (compared to the linear influence of the number of states) in this case, it is not surprising that the setup with two antennas and 13 stimuli was the most expensive with regard to membership queries. As one antenna on the message-level has three states, it is reasonable that two antennas have 9 $(= 3^2)$ states (see Fig. A.11). Colored red and green are the states in which only one antenna is outside its initial state, the white states are those in which both antennas can be triggered to send the $msgImageComplete$ message to mission control in the next step.

On the fluent-level and using simulation code one antenna has eight states: each color can be activated for processing or not, which makes eight (2^3) states (see Fig. A.10).

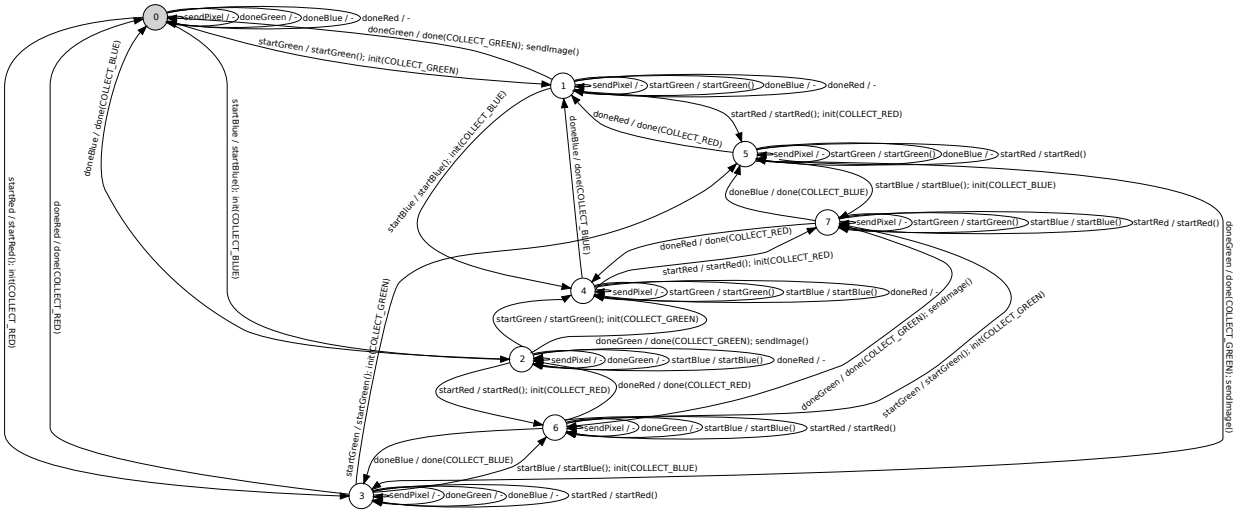


Figure A.10: Model from experiment A_3 (Antenna Aus., fluent-level, sim. threads)

The voyager takes a picture with each camera in turn. The two resulting states differ only in the number of pixels that are sent to each antenna afterwards as is shown in Fig. A.12. For ease of readability the Voyager was modified on the source code level for the experiments to send messages to only one

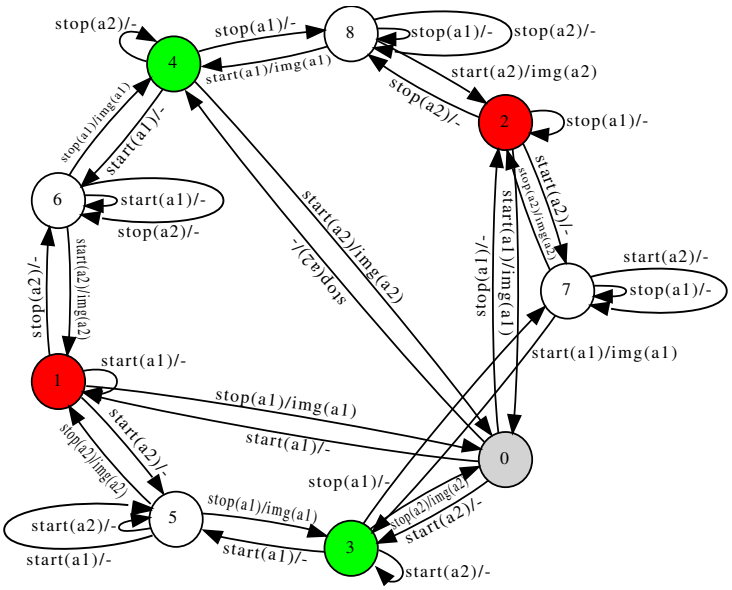


Figure A.11: Model from experiment A_4 (two antennas, message-level, green primitives only)

Figure A.12: Model from experiment A_1 (Voyager, sending to one antenna)

antenna. Otherwise the actions along each transition would have been performed for each of the four antennas.

Conformance of models and specification Fig. A.13 shows the learned model from the setup with one antenna on the message-level (A_2) imported to the jABC/GEAR. Only the green phase's primitives were imported as the primitives for the other colors do not reveal interesting behavior. The corresponding Kripke structure is shown in Fig. A.14. The Kripke structure used here represents names of input and output symbols as atomic propositions. In the cases where there is no output the state representing the output symbol is omitted. This seemed reasonable as there exist only two output symbols in this setup: the empty output and $msgImageComplete$. In most cases no additional proposition is needed and states with no proposition would blow up the size of the Kripke structure without any use.

In the textual version of the specification it is written that $msgImageComplete$ will be sent by an antenna if and only if the fluent representing the phase in which green image pixels are transferred is terminated (which is triggered by $msgEnd_{green}$):

```
FLUENT inSendingImage
  INITIATED_BY { EVENTS.imageSessionEndedGreen }
  TERMINATED_BY { EVENTS.imageAntCaliforniaSent }
```

As the model of the voyager (see Fig. A.12) shows and as it is specified in the ASSL document, the

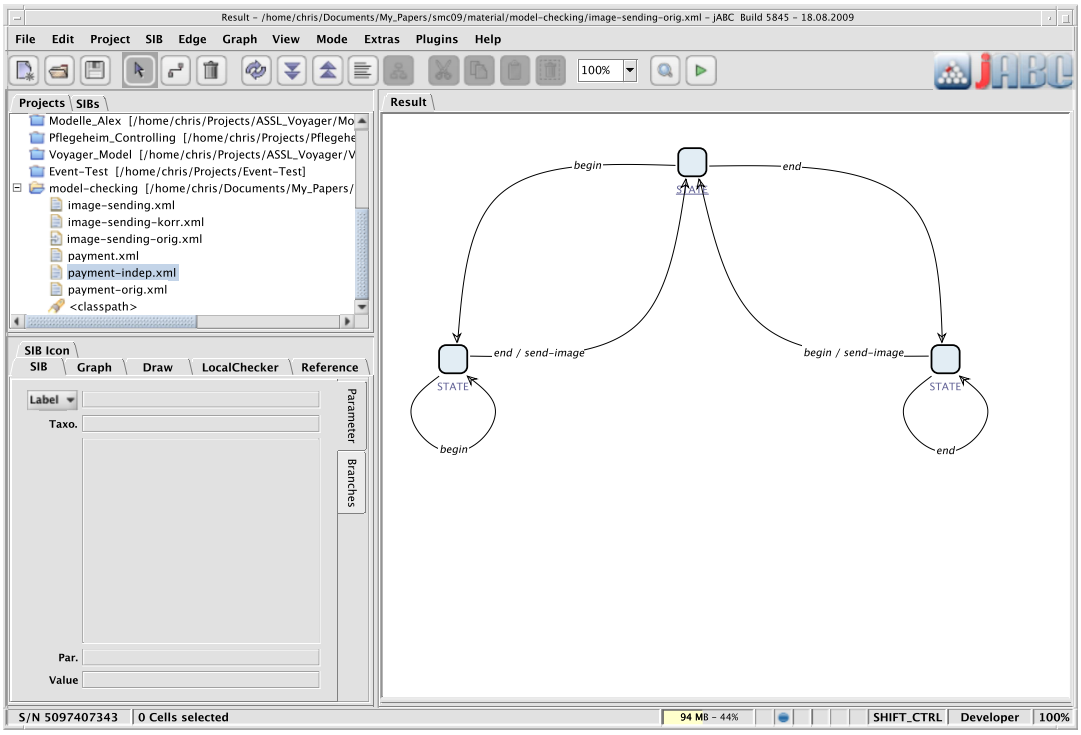


Figure A.13: jABC with imported model for A_2 (green primitives only)

transmission phase for the green image pixels is always the last one for a picture. Terminating the green phase leads automatically to the initiation of the phase (fluent) during which the $msgImageComplete$ will be sent. The general order of the three phases is not well defined for the antennas.

```
DO {
  CALL AEIP.FUNCTIONS.receiveImagePixelMsg
  numPixels = numPixels + 1
}
WHILE numPixels < AS.numPixelsPerImage
CALL AEIP.FUNCTIONS.receiveSessionEndMsg(filterName)
```

It is specified however (see above) that the actions performed when receiving an opening message for a phase will lead directly to the termination of the same phase.

Contradicting the specification Fig. A.14 shows that the sequence $< msgEnd_{green}, msgBegin_{green} >$ will lead to $msgImageComplete$ being sent also (in the figure it is $< end, begin >$). As all other than the messages related to the green phase resulted in reflexive transitions those are omitted in the imported model).

Using the GEAR model checker the violation of the “send message after green phase” property can be visualized in a fashion that shows exactly which states violate it. The used formula (CTL in GEAR dialect[1]) reads:

$$(\neg send - image \vee AX_b(end)) \wedge (\neg EX(send - image) \vee end)$$

The former half of the formula will fail for all states annotated with $send-image$ but preceded by something other than end. The latter half will fail for states that are not labeled end but are succeeded by a $send - image$.

[1] In the CTL dialect provided by GEAR AX_b denotes “for all predecessors”

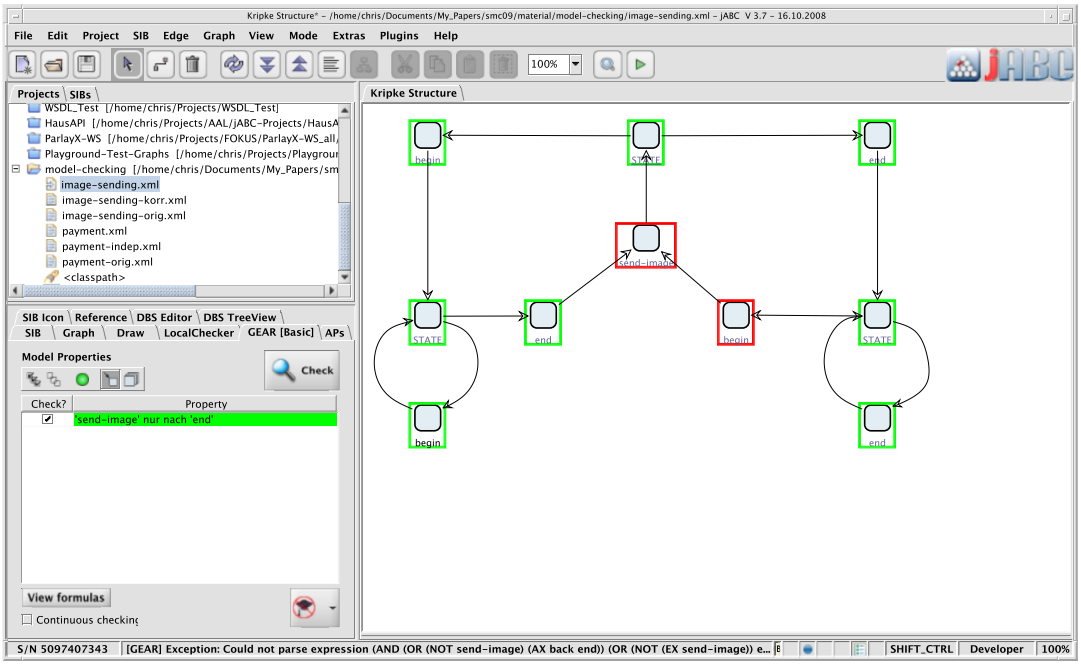


Figure A.14: Kripke structure of model from Fig. A.13, colored according to CTL formula

To substantiate the assumption that the behavior found is actually an example of unconformant behavior any influence of the experimental setup had to be disqualified. We thus ran the erroneous sequence on the system under test using the HotSpot Jvm without any modifications. The result was the same as observed during the learning. The system would send the message after receiving the two messages in the wrong order. As the system under test is generated directly from the specification by an ASSL-to-Java code generator the existence of unconformant behavior implies errors in the generator.

Apart from the presented error no other unconformity was found comparing the textual specification with the learned models.

Internal logical consistency The $< msgEnd_{green}, msgBegin_{green} >$ sequence produced unwanted behavior in all *JikesRVM* setups but not for the fluent-level setup in which simulation code was used. In the simulated code no threads were used and scheduling would take place in a well defined order. A $< msgEnd_{green}, msgBegin_{green} >$ sequence has no effect. After ruling out the modified VM as cause for the unconformant behavior, it had to be explainable by the differences between the simulation scenario and the ones using a modified VM.

One possible explanation for what happens here is timing and scheduling related. After sending both messages, due to some not expected race condition, the green transmission fluent is activated and then terminated directly afterwards. This possibility was eliminated by introducing long enough waits between the single steps done in each experiment. The erroneous behavior was not influenced.

This is only possible because messages are never lost in the generated code. The channels work like a set of buckets. There is one bucket for each message type and no bucket is ever emptied. Thus a $< msgEnd_{green}, msgBegin_{green} >$ sequence will lead to starting the green transmission phase and then using the stored message terminating it immediately. We found that in the simulation scenario channels were emptied by the test driver after each single step, while in the VM scenario only after each experiment the channels were emptied.

The cause for the unwanted behavior is due to the unrealistic implementation of message passing between the Voyager and the antennas. As in reality only loss of messages, but no modifications to the sequence, may occur we modified the experimental setup accordingly (by clearing channels before each

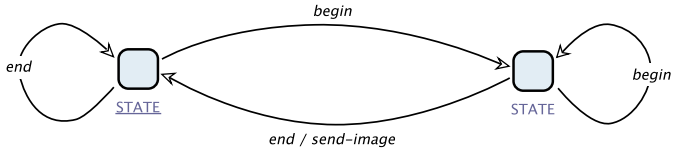


Figure A.15: jABC imported model for A_2, only green phase, corr. channel behavior

step) and rerun experiment A_2. The resulting model imported into jABC/GEAR is shown in Fig. A.15. Evidently the unwanted behavior is no longer existent.

A.2.5 Summary

The conformance of textual specification and generated code was tested successfully. The one error found lead to unexpected behavior of the generated message channels. Apart from this error the code generator works (considering the typical constraints to be made when applying active learning) as expected and produce code conforming to the specification. The constraints to be named are:

1. active learning relies on equivalence queries which can only be approximated in practice. The value of the results thus generally depends on the confidence level of the performed equivalence query approximation. The exact confidence level however can only be determined with respect to an upper bound on the number of states the system under test may have. We chose an upper bound that seems sound in the given case study. With respect to this bound the confidence level of the performed approximation is 100%.
2. active learning is not capable of representing time dependent behavior. The ASSL-to-Java code generator uses threads in many places. To cover the possible behavior completely scheduling had to be part of the learning process.

Though this case study is only dealing with quite small models some valuable conclusions may be drawn from it. For a complete discussion we refer to Chapter 2.

A.3 Generating Models of Communication Protocols using Regular Inference with Abstraction

This paper contains a worked-out presentation of the framework for using abstraction for bridging between different levels of abstraction, which is reported in Section 3.3. Such bridging is needed, not only for generating abstract models by the learning enabler, but also later when generating running code from an abstract specification of a synthesized connector. The paper also evaluates the applicability of this approach to real-world protocols by reporting on an implementation of the techniques by connecting the LearnLib tool for regular inference with the protocol simulator ns-2, and generating models of SIP and TCP protocol components as implemented in ns-2.

Generating Models of Infinite-State Communication Protocols using Regular Inference with Abstraction*

Fides Aarts†
Department of Computer Systems, Uppsala University,
f.aarts@cs.ru.nl

Bengt Jonsson
Department of Computer Systems, Uppsala University,
bengt@it.uu.se

Johan Uijen
Department of Computer Systems, Uppsala University,
johanuijen@student.ru.nl

ABSTRACT

To promote model-based verification and validation, it would be highly useful to develop techniques for generating models of communication system components from observations of their external behavior. Previous work on model generation has employed regular inference techniques which generate only finite-state models. However, typical protocol entities do not have finite-state models, since they use sequence numbers, buffers, and other unbounded data domains. We present a framework, which adapts regular inference to generating models of infinite-state components with infinite message alphabets. We adapt abstraction techniques, used in formal verification and program analysis, to the constraints of a black-box setting. A component interface is augmented by an abstraction layer, which produces a finite-state abstract view: by regular inference, a finite-state model is inferred, which can thereafter be mapped to an infinite-state model of the component. Since inference cannot access the internal state of a component, it is more difficult to define abstractions than in, e.g., model checking: therefore we present techniques for generating them systematically under restrictions on component behavior. We have implemented our techniques by connecting the LearnLib tool for regular inference with the protocol simulator ns-2, and generated models of SIP and TCP protocol components as implemented in ns-2.

1. INTRODUCTION

Model-based techniques for verification and validation of communication protocols and reactive systems, including model checking and model-based testing [6] have witnessed drastic advances in the last decades, and are being applied in industrial settings (e.g., [18]). They require formal models that specify the intended behavior of system components, which ideally should be developed during specification and design. However, the construction of models typically requires significant manual effort, implying that in practice often models are not available, or become outdated as the system evolves. Automated support for constructing models of the behavior of implemented components would therefore be extremely useful, e.g., for regression testing, for replacing manual testing by model based testing, for producing models of standardized protocols, for analyzing whether an existing system is vulnerable to attacks, etc. Techniques, developed for program analysis, that construct models from source code (e.g., [3, 17]) are often of limited use, due to the presence of library modules, third-party components, etc., that make analysis of source code difficult. We therefore consider techniques for constructing models from observations of their external behavior.

The construction of models from observations of component behavior can be performed using regular inference (aka automata learning) techniques [2, 9, 20, 27]. This class of techniques is now receiving increasing attention in the testing and verification community, e.g., for regression testing of telecommunication systems [16, 19], for integration testing [21, 21, 15], security protocol testing [29], and for combining conformance testing and model checking [25, 14]. Algorithms for regular inference pose a sequence of *membership queries*, each of which observes the component's output in response to a certain input string, and thereafter produce a minimal deterministic finite-state machine which conforms to the observations. After a sufficient number of membership queries, the produced machine will be a faithful model of the observed component.

Since regular inference techniques are designed for finite-state models, previous applications to model generation have been limited to generating a finite-state view of the system behavior, implying that, e.g., the alphabet must be made finite, e.g., by suppressing parameters. However, typical protocol entities do not have finite-state models, since they use sequence numbers, buffers, and other unbounded data domains. Modeling this part of behavior is often important: for instance the influence of data on control flow is used in model-based test generation tools, such as ConformiQ Qtronic [18]. Extensions of regular inference to handle simple forms of identifiers [5] or timed automata [11, 12, 13] have been proposed, using techniques specific to the proposed extensions, but there is no general framework for generating

*Supported in part by EC Proj. 231167 (CONNECT).
†Supported in part by the European Community's 7th Framework Programme No. 214755 (QUASIMODO). Fides Aarts and Johan Uijen are currently affiliated with the Inst. f. Comp. and Inf. Sciences, Radboud University

infinite-state models, incorporating data and control, by inference.

In this paper, we propose a framework for using regular inference to produce models of infinite-state components with infinite communication alphabets. The idea is to augment finite-state regular inference by an abstraction layer, which provides a finite-state abstract view to the inference engine, but allows to generate infinite-state models by reversing the effect of the abstraction. This technique is inspired by predicate abstraction [22, 7], which has been successful for extending finite-state model checking to large and infinite state spaces. In contrast to that work, however, we are now in a black-box setting, so we cannot define the abstraction directly on the source code of a component. Abstractions are typically history-dependent, so the abstraction must also maintain local state information. Since (in contrast to model checking) we cannot access the internal state of a component, finding suitable abstractions is more challenging, but it becomes feasible under restrictions on what operations the component may perform on data.

In this paper, we present a general framework for generating infinite-state models of components: an abstraction is constructed as consisting of a separately maintained local state and a mapping of interface symbols. By regular inference, a finite-state model of the abstracted interface is inferred. The abstraction can then be reversed to generate an infinite-state model of the component. Since inference cannot access the internal state of a component, it is more difficult to define abstractions than in, e.g., model checking. We therefore present techniques for generating abstractions systematically under restrictions on component behavior. We propose to realize the abstraction by an external component, which we call a *Mapper*. We have implemented our techniques by connecting the LearnLib tool for regular inference with the protocol simulator ns-2, which provides implementations of standard protocols. We have used it to generate models of the ns-2 implementations of entities in the SIP and TCP protocols.

Related Work.. Regular inference techniques have been used for several tasks in verification and test generation, e.g., to create models of environment constraints with respect to which a component should be verified [8], for regression testing to create a specification and a test suite [16, 19], to perform model checking without access to source code or formal models [14, 25], for program analysis [1], and for formal specification and verification [8]. Groz, Li, and Shahbaz [21, 28, 15] extend regular inference to Mealy machines with data values, for use in integration testing, but use only a finite set of the data values in the obtained model. In particular, they do not infer internal state variables. Shu and Lee [29] learns the behavior of security protocol implementations for a finite subset of input symbols, which can be extended in response to new information obtained in counterexamples. Mariani and Pezzé use inference in integration testing of commercial off the shelf components [23]. They infer two separate models, which are not closely related: one for the finite-state control, and the others being a relation on the parameters in each interaction. They use different inference techniques for each type of model. The approach we present in this paper unifies the control and data inference in one framework. Extensions of regular inference have been proposed to handle infinite-state systems, in which parameters of messages and state variables are from an unbounded domain, e.g., for identifiers [5], and timers [11, 10]. These extensions are specialized towards a particular data domain: this paper proposes a general framework for incorporating a range of such data domains, into which techniques specialized for different data domains can be incorporated. In previous work [4], we also presented an optimization of regular inference for models with boolean data parameters, containing techniques for refining guards which can be adapted to refining abstractions in the framework of this paper.

Organization.. In the next section, we give an overview of our approach by means of a simple example. Basic definitions of Mealy machine are in Section 3 and presentation of inference in Section 4. The abstraction technique is presented in Section 5, and the application to SIP and TCP are reported in Section 6. Section 7 contains conclusions and directions for future work.

2. AN INTRODUCTORY EXAMPLE

Let us introduce our techniques by means of a small example. Assume we are given a protocol entity (called SUT) which services requests to set up a connection. To us, SUT is a black box. We know its static interface: it receives input messages of form $REQ(id, sn)$ and $CONF(id, sn)$ and transmits output messages of form $REPL(id, sn)$, $ACK(id, sn)$, or REJ, where the parameters id and sn range over natural numbers. We intend to infer a model of its dynamic behavior by means of regular inference. However, SUT can not be modeled as a finite-state machine (since it handles sequence numbers and connection identifiers), whereas regular inference can only infer finite-state models. We overcome this discrepancy by introducing an abstraction, which maps the elements in the (infinite) static interface of SUT to a (small) finite alphabet. As in model checking, this abstraction mapping may depend on the previous sequence of symbols transmitted over the interface, which is typically summarized in state variables. In model checking this is made easier, since the given model provides access to the internal state of the SUT. However, we are in a black-box setting, where the information maintained by the SUT can only be inferred through observation. This becomes tractable if we may assume restrictions on what operations the SUT can perform on data parameters received in input messages, making its behavior "not too complicated".

Let us say that we can assume that the id parameter always represents a connection identifier, and that the sn parameter represents a sequence number (this information could be obtained using documentation, type description, reflection, or some other means). We assume that the only allowed binary operations on these parameters are to check for equality. In addition, sequence numbers may be incremented, but at most once between each reception of a message and the transmission of the corresponding response. By systematically observing the behavior of SUT during the regular inference (as described more precisely in Section 5), we can deduce which input parameters it remembers. Suppose that this investigation reveals that the id and sn parameters that

are received in the first *REQ* message are remembered by the *SUT* when processing subsequent messages, but thereafter forgotten. This conclusion leads us to define an abstraction mapping based on two state variables:

- *cur_id*, which is initially "undefined" (denoted $\perp$), and thereafter assigned to the *id* parameter of the first received *REQ* message.
- *cur_sn*, which is also initially "undefined" and thereafter assigned to the *sn* parameter of the first received *REQ* message.

Using these state variables, the abstraction is defined as follows. Each message of form $REQ(id, sn)$, $CONF(id, sn)$, $RESP(id, sn)$, or $ACK(id, sn)$ is mapped to a symbol of form $REQ(\text{ID}, \text{SN})$, $CONF(\text{ID}, \text{SN})$, $RESP(\text{ID}, \text{SN})$, or $ACK(\text{ID}, \text{SN})$, where ID is either CUR, denoting the *id* parameter value that is remembered, and OTHER, denoting any other parameter value. Analogously SN is either CUR, denoting the *sn* parameter value that is remembered, OTHER, denoting any other parameter value, or CUR + 1, denoting the result of incrementing the remembered value. Table 1 summarizes the mapping from "concrete" parameter values to corresponding abstract parameter values. Each entry in the table contains constraints under which the parameter of the row (*id* or *sn*) will be mapped to the abstract value of the column (CUR, CUR + 1, or OTHER). We use *mtype* to denote the type of the message considered (being either *REQ*, *CONF*, *RESP*, *ACK*, or *REJ*). Thus, in total there

par	CUR	CUR + 1	OTHER
id	$cur\_id = \perp$ $\wedge$ $mtype = REQ$ $\vee$ $id = cur\_id$ $\wedge$ $cur\_id \neq \perp$		$id \neq cur\_id$ $\wedge$ $cur\_id \neq \perp$
sn	$cur\_sn = \perp$ $\wedge$ $mtype = REQ$ $\vee$ $sn = cur\_sn$ $\wedge$ $cur\_sn \neq \perp$	$sn = cur\_sn + 1$ $\wedge$ $cur\_sn \neq \perp$	$sn = cur\_sn$ $\wedge$ $cur\_sn \neq \perp$

Table 1: Abstraction mappings for parameters

are 12 abstract input symbols and 13 abstract output symbols (the symbol *REJ* is mapped to itself). This abstraction mapping is naturally extended to a mapping from sequences of (concrete) messages to sequences of abstract symbols. It is then possible to infer the behavior over abstract symbols, modeled as a finite-state machine, using regular inference.

To actually perform the inference, we assume that we have a *Learner* module, which uses regular inference to infer finite-state models from observed responses to sequences of input symbols. We wrap the interface of *SUT* by a *Mapper* module, as shown in Figure 1. The *Mapper* maintains the local state information that is needed to perform the abstraction mapping (in this example the state variables *cur_id* and *cur_sn*). Intuitively, the *Mapper* hides the details of the "data part" of *SUT*, so that the *Learner* can be used to infer its "control part". Each symbol *a* sent by the *Learner* is translated by the *Mapper* to some message, which is mapped to *a* by the abstraction mapping, and sent to *SUT*. The corresponding

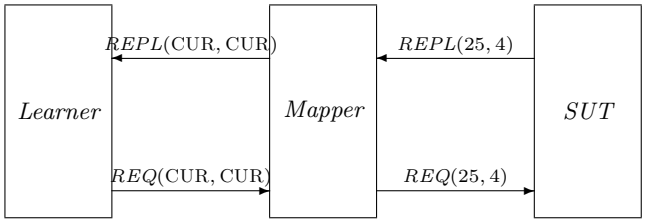
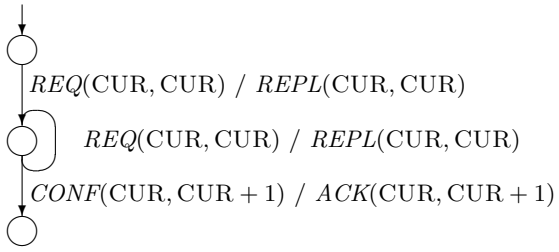


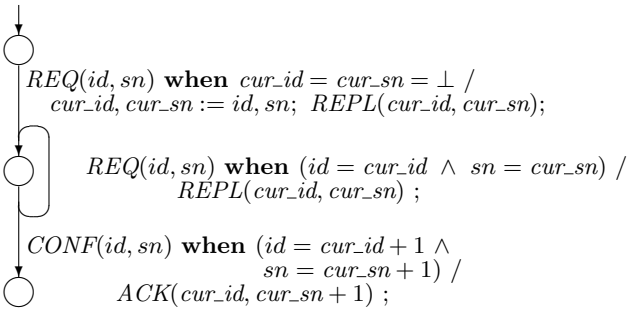
Figure 1: Introduction of *Mapper* module

reply by *SUT* is translated to an abstract symbol, using the mapping, and sent back to the *Learner*. Thus, the *Learner* perceives a Mealy machine which interacts using the alphabet of abstract symbols. She can now proceed to infer a finite-state machine that describes the possible sequences of abstract symbols transmitted between the *Learner* and the *Mapper*. A possible result is the Mealy machine in the figure below.



Each arc is labeled by an input symbol followed by the output symbol that the *Learner* observes in response. From the picture, we have excludes all arcs that contain the output symbol *REJ*: these all go to a terminal error state (also not shown).

Knowing the behavior of the *Mapper*, we can transform the above Mealy machine into a symbolically described infinite-state Mealy machine, which describes the behavior of the *SUT*. This is done by replacing each abstract input and output symbol by a symbolic expression that characterizes the corresponding concrete messages, and by adding assignments to maintain local variables of the *Mapper*. The result is shown below. The machine is initially in the uppermost control location, and variables *cur_id* and *cur_sn* are initialized to $\perp$. Each transition is triggered by the reception of a message, if parameter values satisfy the guard following **when**. If triggered, the transition will perform the assignments and transmission of output message occurring after the /. For brevity, we have omitted checks that *cur_id* and *cur_sn* are not $\perp$ in the guards of the two lower arcs. As for the abstract machine, we have suppressed all arcs where the machine replies with the output symbol *REJ* and lead to a terminal error state: their guards are satisfied by input symbols not covered by any of the displayed statements.



3. MEALY MACHINES

Basic Definitions. We will use Mealy machines to model communication protocol entities. Here we define the "flat" version of Mealy machines. A *Mealy machine* is a tuple $\mathcal{M} = \langle \Sigma_I, \Sigma_O, Q, q_0, \delta, \lambda \rangle$ where Σ_I is a nonempty set of *input symbols*, Σ_O is a nonempty set of *output symbols*, Q is a nonempty set of *states*, $q_0 \in Q$ is the *initial state*, $\delta : Q \times \Sigma_I \to Q$ is the *transition function*, and $\lambda : Q \times \Sigma_I \to \Sigma_O$ is the *output function*. Elements of Σ_I^* are called *input strings*, and elements of Σ_O^* are called *output strings*. The sets of states and symbols can be finite or infinite.

An intuitive interpretation of a Mealy machine is as follows. At any point in time, the machine is in some state $q \in Q$. It is possible to give inputs to the machine, by supplying an input symbol $a \in \Sigma_I$. The machine responds by producing an output symbol $\lambda(q, a)$ and transforming itself to the new state $\delta(q, a)$. We use the notation $q \xrightarrow{a/b} q'$ to denote that $\delta(q, a) = q'$ and $\lambda(q, a) = b$; in this case $q \xrightarrow{a/b} q'$ is called a *transition* of $\mathcal{M}$.

We extend the transition and output functions from input symbols to input strings in the standard way, by defining:

$$\begin{array}{lcllcl} \delta(q, \varepsilon) & = & q & \lambda(q, \varepsilon) & = & \varepsilon \\ \delta(q, ua) & = & \delta(\delta(q, u), a) & \lambda(q, ua) & = & \lambda(q, u)\lambda(\delta(q, u), a) \end{array}$$

The Mealy machines that we consider are *completely specified*, meaning that at every state the machine has a defined reaction to every input symbol in Σ_I, i.e., δ and λ are total. They are also *deterministic*, meaning that for each state q and input symbol a exactly one next state $\delta(q, a)$ and output string $\lambda(q, a)$ is possible.

Given a Mealy machine $\mathcal{M}$ with input alphabet Σ_I, output function λ, and initial state q_0, we define $\lambda_{\mathcal{M}}(u) = \lambda(q_0, u)$, for $u \in \Sigma_I^*$. Two Mealy machines $\mathcal{M}$ and $\mathcal{M}'$ with input alphabets Σ_I are *equivalent* if $\lambda_{\mathcal{M}}(u) = \lambda_{\mathcal{M}'}(u)$ for all input strings $u \in \Sigma_I^*$.

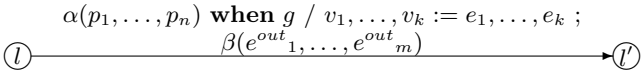
Symbolic Representation. When modeling entities of communication protocols, messages are typically described as consisting of a message type with a number of parameters, and states are typically defined by a combination of a control location and state variables. We will therefore define a symbolic way to represent Mealy machines, as a particular form of "Extended Finite State Machines" (EFSMs).

Input and output symbols will be represented using finite sets I and O of (input and output) *action types*. Each action type α has a certain *arity*, which is a tuple of *domains* (a domain is a set of allowed data values) $\mathcal{D}_{\alpha,1}, \ldots, \mathcal{D}_{\alpha,n}$ (where n depends on α). We will use d, d_1, d_2, etc. to range over data values. Let Σ_I be the set of *input symbols* of form $\alpha(d_1, \ldots, d_n)$, where $d_i \in \mathcal{D}_{\alpha,i}$ is in the appropriate domain for each i with $1 \leq i \leq n$. The set of *output symbols* Σ_O is defined analogously.

To represent states, we use a finite set L of locations, and a finite set V of *state variables*. Each state variable v has a domain of possible values, and a unique initial value. Let a *valuation* σ be a partial mapping from the set V of state variables to data values in their respective domains. Let σ_0 denote the valuation which maps each state variable to its initial value. The set of states of a Mealy machine is now the set of pairs $\langle l, \sigma \rangle$, where $l \in L$ is a location, and σ is a valuation. The initial state is $\langle l_0, \sigma_0 \rangle$.

Let us finally consider the representation of the transition and output functions. There are several possible ways to represent these symbolically. We have chosen a simple formalism based on guarded assignment statements. We will use a finite set of *formal parameters*, ranged over by $p_1, p_2, \ldots$, which will serve as local variables in each guarded assignment statement. We will assume some constants and operators to form expressions, and extend the definition of valuations to expressions over state variables in the natural way; for instance, if $\sigma(v_3) = 8$, then $\sigma(2 * v_3 + 4) = 20$.

A *symbolic transition* is a transition between locations, labeled by a guarded assignment statement. We denote it as



where

- l and l' are locations,
- $p_1, \ldots, p_n$ is a tuple of different formal parameters, In what follows, we will use $\overline{d}$ for $d_1, \ldots, d_n$ and $\overline{p}$ for $p_1, \ldots, p_n$,
- g is a boolean expression (the *guard*) over $\overline{p}$ and the state variables in V,
- $v_1, \ldots, v_k := e_1, \ldots, e_k$ is a multiple assignment statement, which to some (distinct) state variables $v_1, \ldots, v_k$ in V assigns the values of the expressions $e_1, \ldots, e_k$; here $e_1, \ldots, e_k$ are expressions over $\overline{p}$ and state variables in V,
- ${e^{out}}_1, \ldots, {e^{out}}_m$ is a tuple of expressions over $\overline{p}$ and state variables, which evaluate to data values ${d'}_1, \ldots, {d'}_m$ so that $\beta({d'}_1, \ldots, {d'}_m)$ is an output symbol.

Intuitively, the above guarded assignment statement denotes a step of the Mealy machine in which some input symbol of form $\alpha(d_1, \ldots, d_n)$ is received and the values $d_1, \ldots, d_n$ are assigned to the corresponding formal parameters $p_1, \ldots, p_n$. If now the guard g is satisfied, the state variables among $v_1, \ldots, v_k$ are assigned new values and an output symbol, obtained by evaluating $\beta({e^{out}}_1, \ldots, {e^{out}}_m)$, is generated. The statement does not denote any step in case g is not satisfied.

Formally, the guarded assignment statement above denotes that for the location l and the input symbols of form $\alpha(\overline{d})$ for which $\sigma(g[\overline{d}/\overline{p}])$ is true we have

- $\delta(\langle l, \sigma\rangle, \alpha(\overline{d})) = \langle l', \sigma'\rangle$, where σ' is the valuation such that $\sigma'(v) = \sigma(e_i[\overline{d}/\overline{p}])$ if v is among $v_1, \ldots, v_k$, and $\sigma'(v) = \sigma(v)$ otherwise
- $\lambda(\langle l, \sigma\rangle, \alpha(\overline{d})) = \beta(\sigma'(e_1^{out}[\overline{d}/\overline{p}]), \ldots, \sigma'(e_m^{out}[\overline{d}/\overline{p}]))$.

4. INFERENCE

In this section, we present the setting for inference of Mealy machines. It is formulated in the same setting as Angluin's L^* algorithm [2], in which a so called *Learner*, who initially knows nothing about the Mealy machine $\mathcal{M}$, is trying to infer $\mathcal{M}$, by asking queries to a so called *Oracle*. The queries are of two kinds.

- A *membership query* asks what the output is on a string $w \in (\Sigma_I)^*$.
- An *equivalence query* asks whether a hypothesized Mealy machine $\mathcal{H}$ is correct, i.e., whether $\mathcal{H}$ is equivalent to $\mathcal{M}$. The *Oracle* will answer *yes* if $\mathcal{H}$ is correct, or else supply a *counterexample*, which is a string $u \in (\Sigma_I)^*$ such that $\lambda_{\mathcal{M}}(u) \neq \lambda_{\mathcal{H}}(u)$.

The typical behavior of a *Learner* is to start by asking a sequence of membership queries until she can build a "stable" hypothesis $\mathcal{H}$ from the answers. After that she makes an equivalence query to find out whether $\mathcal{H}$ is equivalent to $\mathcal{M}$. If the result is successful, the *Learner* has succeeded, otherwise she uses the returned counterexample to perform subsequent membership queries until converging at a new hypothesized Symbolic Mealy machine, which is supplied in an equivalence query, etc.

For flat finite-state Mealy machines, the above problem is well understood, and there is an implementation in the LearnLib tool [26] of an adaptation of the L^* algorithm due to Niese [24]. In a black-box setting, equivalence queries can only be approximated: in LearnLib by test suites of user-controllable size.

5. INFERENCE USING ABSTRACTION

In this section, we present our technique for using regular inference to infer models of large- or infinite-state Mealy machines: the main idea is to transform the dynamic interface of the *SUT* into a finite-state interface by an abstraction mapping. We have adapted ideas from predicate abstraction [22, 7], which has been successful for extending finite-state model checking to larger and even infinite state spaces. However, we are in a black-box setting and so we must use abstraction in a different way.

We assume that we are given the task of inferring a symbolic representation of a Mealy machine $\mathcal{M} = \langle \Sigma_I, \Sigma_O, Q, q_0, \delta, \lambda\rangle$, where Σ_I, Σ_O, and Q may be large or even infinite. As in Section 2, this is done by defining an abstraction mapping, which transforms input and output strings of $\mathcal{M}$ into strings over a finite alphabet, so that the resulting behavior can be described by a (hopefully small) finite-state Mealy machine $\mathcal{M}^{\mathcal{A}}$, in which the alphabets and set of states are finite. We will often refer to the latter Mealy machine as an "abstract" one, and similarly for its input and output alphabets, etc. We will use the superscript $^{\mathcal{A}}$ when referring to the abstract version of alphabets, states, etc.

The abstraction is defined using an additional set R of local states, containing an initial state r_0. Whenever a new input symbol is observed, the local state is updated by a function $\delta^{\mathcal{R}} : R \times \Sigma_I \mapsto R$. We can extend $\delta^{\mathcal{R}}$ to a mapping from sequences of input symbols to the reached local state, by defining:

$$\delta^{\mathcal{R}}(\varepsilon) = r_0 \qquad \delta^{\mathcal{R}}(ua) = \delta^{\mathcal{R}}(\delta^{\mathcal{R}}(u), a),$$

The local state is used to define two state-dependent abstraction mappings:

- $abstr_I : R \times \Sigma_I \mapsto \Sigma_I^{\mathcal{A}}$ maps input symbols to a finite set $\Sigma_I^{\mathcal{A}}$ of abstract input symbols,
- $abstr_O : R \times \Sigma_O \mapsto \Sigma_O^{\mathcal{A}}$ maps output symbols to a finite set $\Sigma_O^{\mathcal{A}}$ of abstract output symbols. The mapping $abstr_O$ is required to be deterministic in the sense that for each $r \in R$ and $b^{\mathcal{A}} \in \Sigma_O^{\mathcal{A}}$ there is at most one output symbol $b \in \Sigma_O$ such that $abstr_O(r, b) = b^{\mathcal{A}}$.

These mappings ($abstr_I$ and $abstr_O$) together imply the behavior of $\mathcal{M}^{\mathcal{A}}$: if after the input sequence u it has reached state $q^{\mathcal{A}}$, its reaction $\lambda^{\mathcal{A}}(q^{\mathcal{A}}, u)$ is $abstr_O(r, \lambda(q, a))$, where $abstr_I(r, a) = abstr_I(r, a')$. Here, $r = \delta^{\mathcal{R}}(u)$ and $q = \delta(u)$ are the states reached after u in the abstraction and in $\mathcal{M}$, respectively.

Intuitively, one can think of the behavior of $\mathcal{M}^{\mathcal{A}}$ as follows: Whenever $\mathcal{M}^{\mathcal{A}}$ receives an abstract input symbol $a^{\mathcal{A}} \in \Sigma_I^{\mathcal{A}}$, this corresponds to $\mathcal{M}$ receiving some symbol a with $abstr_I(r, a) = a^{\mathcal{A}}$, where r is the maintained local state of the abstraction. Then $\mathcal{M}$ then responds by the output symbol $\lambda(q, a)$ which is mapped to $abstr_O(r, \lambda(q, a))$, which becomes the output from $\mathcal{M}^{\mathcal{A}}$. Since each abstract membership query in $(\Sigma_I^{\mathcal{A}})^*$ in general corresponds to several concrete queries to $\mathcal{M}$, a thorough inference process should perform each abstract query several times, with a strategy for covering the set of corresponding concrete input strings. This aspect is outside the scope of this paper, so we do not further elaborate on that here.

We see that the behavior of $\mathcal{M}^{\mathcal{A}}$ is deterministic only if

$$abstr_I(r, a) = abstr_I(r, a') \text{ implies } abstr_O(r, \lambda(q, a)) = abstr_O(r, \lambda(q, a'))$$

for all input symbols a, a', and (reachable combinations of) states q, r. Intuitively, this means that two input symbols (a and a' in this case) should be mapped to the same abstract input symbol only if they are "handled in the same way" by $\mathcal{M}$ (i.e., produce the same abstract output symbol). Later in this section, we present techniques to construct abstractions that satisfy this condition. We also present techniques, which can refine the input abstraction $abstr_I$ appropriately if it is discovered that the condition is violated for some input symbols a, a' and local state r of the abstraction.

Let us now describe how abstraction mappings can be conveniently defined symbolically:

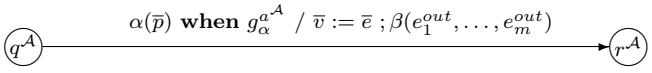
- Local states are defined by a set V' of state variables with initial values, and expressions for updating their values in response to each input symbol in Σ_I. The set R is then the set of valuations of the variables in V'.
- $abstr_I$ (the input abstraction mapping) can be defined by supplying, for each abstract input symbol $a^{\mathcal{A}}$ in $\Sigma_I^{\mathcal{A}}$, and each input action type α, a guard $g_\alpha^{a^{\mathcal{A}}}$ which characterizes when an input symbol of form $\alpha(p_1, \ldots, p_n)$ is mapped to the abstract symbol $a^{\mathcal{A}}$.
- $abstr_O$ (the output abstraction mapping) can be described by letting each abstract output symbol in $\Sigma_O^{\mathcal{A}}$ be an expression of form $\beta(e_1^{out}, \ldots, e_m^{out})$, where each e_i^{out} is an expression over state variables and parameters in the just received triggering input message, so that $\beta(e_1^{out}, \ldots, e_m^{out})$ evaluates to an output symbol $\beta(d_1, \ldots, d_m)$ in Σ_O. In Section 2 and 6, we use a variation of this, in the form of constraints on the parameters of β.

If the *Mapper* is designed properly, the *Learner* will be able to infer an abstract Mealy machine $\mathcal{M}^{\mathcal{A}} = \langle \Sigma_I^{\mathcal{A}}, \Sigma_O^{\mathcal{A}}, Q^{\mathcal{A}}, q_0^{\mathcal{A}}, \delta^{\mathcal{A}}, \lambda^{\mathcal{A}} \rangle$. We can then derive the corresponding behavior of $\mathcal{M}$, represented as the Mealy machine $\mathcal{M} = \langle \Sigma_I, \Sigma_O, Q^{\mathcal{A}} \times R, \langle q_0^{\mathcal{A}}, r_0 \rangle, \delta, \lambda \rangle$, where

- $\lambda(\langle q^{\mathcal{A}}, r \rangle, a)$ is an output symbol b such that $abstr_O(r, b) = \lambda^{\mathcal{A}}(q^{\mathcal{A}}, a)$, and
- $\delta(\langle q^{\mathcal{A}}, r \rangle, a) = \langle \delta^{\mathcal{A}}(q^{\mathcal{A}}, a), \delta^{\mathcal{R}}(r, a) \rangle$.

We note that the requirement that $abstr_O$ be injective is essential for obtaining a deterministic output function for $\mathcal{M}$.

To obtain a symbolic representation of $\mathcal{M}$, we simply let $Q^{\mathcal{A}}$ be the set of locations, and transform each transition $q^{\mathcal{A}} \xrightarrow{a^{\mathcal{A}}/b^{\mathcal{A}}} r^{\mathcal{A}}$, where $b^{\mathcal{A}}$ is $\beta(e_1^{out}, \ldots, e_m^{out})$, into the symbolic transition



where $\overline{v} := \overline{e}$ is the update defined in response to an input symbol of form $\alpha(\overline{p})$ when $g_\alpha^{a^{\mathcal{A}}}$ is satisfied.

As indicated in Section 2, we will typically succeed in building a suitable abstraction only if we can make restrictive assumptions on the behavior of $\mathcal{M}$. Typically, we assume that $\mathcal{M}$ can be modeled as a finite control skeleton which manipulates the data received in input messages. The abstraction mapping is then used to capture the potential "data aspects" of $\mathcal{M}$, leaving the control aspects to be inferred by regular inference. For different assumptions on how data are manipulated by $\mathcal{M}$, we can devise systematic techniques for using observations to define an initial abstraction: these observations can be made in a preliminary phase before the actual inference process, or can be integrated with the inference. An alternative to such an initial observation phase is that we have enough *a priori* knowledge about the behavior of $\mathcal{M}$ to define an initial abstraction.

Let us present how the definition of an abstraction can be generated for the case that parameters are data values from a potentially unbounded set, where the only allowed operations are to generate fresh values or compare existing ones for equality. They are adapted from our previous work [5]. For parameters of this form, the crucial decision in the design of a suitable abstraction is to decide when to store received parameters in state variables to allow comparisons with parameters that occur in the future. The techniques for finding out which parameters to store are, in short, as follows.

Consider an input string u, which contains a parameter value d. We intend to investigate whether $\mathcal{M}$ remembers d after having received u. This is done by observing the output of $\mathcal{M}$ in response to continuations following u. Let d' be a fresh data value that did not occur in u. If there is some continuation v of u such that the reponse $\lambda(\delta(q_0, u), v)$ to v and the reponse $\lambda(\delta(q_0, u), v[d'/d])$ to $v[d'/d]$ (i.e., v where all occurrences of d have been replaced by d') satisfy $\lambda(\delta(q_0, u), v[d'/d]) \neq \lambda(\delta(q_0, u), v)[d'/d]$ then u must be remembered after u: this is so, since $\mathcal{M}$ does not treat d in the same way as a fresh value d'.

Based on this principle, we systematically monitor membership queries to detect which parameters should be stored in state variables. Should the abstraction mapping violate the well-formedness property stated above, the input abstraction can be refined so that the partitioning on input symbols induced by the input abstraction mapping is refined, if necessary by adding more state variables to r. Techniques for this can be adapted from [4, 5].

For parameters which are of type sequence number, the above scheme can be modified to find out which data values must be remembered in the abstraction mapping. Analogous schemes have been developed for timers in the context of (subclasses of) timed automata [10, 13].

6. EXPERIMENTS

We have implemented and applied our approach to infer models of two implemented standard protocols: the Session Initiation Protocol (SIP) and the Transmission Control Protocol (TCP). In this section, we first describe our experimental setup, thereafter its application to the two protocols.

In order to have access to a large number of standard communication protocols, for development and evaluation of our inference techniques, we use the protocol simulator ns-2[1] to serve as *SUT*. The protocol simulator ns-2 provides implementations of a large number of communication protocols. Messages are represented as C++ structures, saving us the trouble of parsing messages represented as bitstrings. As *Learner*, we use the LearnLib tool [26], developed at the Technical University Dortmund, which has an efficient im-

[1] `http:/www.isi.edu/nsnam/ns/`

plementation of the L^* algorithm that can construct both finite automata and Mealy machines. LearnLib provides several different realizations of equivalence queries, including random test suites of user-controlled size.

To set up the experiments, we implemented a *Mapper* module, which performs a translation as described in Section 2. Our *Mapper* module also had to bridge between the asynchronous buffered interface of LearnLib and the synchronous interface of ns-2, which represents each received message as a method call. Each membership query issued by LearnLib must be transformed to a new protocol session with ns-2. The overhead for setting up and closing sessions imposed some limits on the number of membership queries that could be performed in reasonable time.

6.1 SIP

SIP is an application layer protocol for creating and managing multimedia communication sessions, such as voice and video calls. Although a lot of documentation is available, such as the `RFC 3261`, no proper reference behavior model, as a state machine, is available. Our first case study consisted of the behavior of the SIP Server entity when setting up and closing connections with a SIP Client. A message from the SIP Client to the SIP Server has the form *Request*(*Method*, *From*, *To*, *Contact*, *CallId*, *CSeq*, *Via*), where

- *Method* defines the type of request, either INVITE, PRACK or ACK.
- *From* contains the address of the originator of the request.
- *To* contains the address of the receiver of the request.
- *CallId* is a unique session identifier.
- *CSeq* is a sequence number that orders transactions in a session.
- *Contact* is the address on which the Client wants to receive *Request* messages.
- *Via* indicates the transport that is used for the transaction. The field identifies via which nodes the response to this request need to be sent.

A response from the Server to the Client has the form *Response*(*StatusCode*, *From*, *To*, *CallId*, *CSeq*, *Contact*, *Via*), where *StatusCode* is a three digit code status that indicates the outcome of a previous request from the Client, and the other parameters are as for a *Request* message. In the following, we will omit the *Response* and *Request* field, and denote the *Method* and *StatusCode* fields as message types, i.e., a *Response* will have the form *StatusCode*(*From*, *To*, *CallId*, *CSeq*, *Contact*, *Via*).

Abstraction Mapping. Let us describe how we arrived at an abstraction mapping for SIP. We consider separately the parameters. The parameters *From*, *To*, and *Contact* must be pre-configured in a session with ns-2, so they are set to constant values throughout the experiment. The *Via* parameter is a pair, consisting of a default address and a variable branch. The parameters *Via*, *CallId*, and *CSeq* are potentially interesting parameters. A priori, they can be handled as parameters from a large domain, on which test for equality and potentially incrementation can be performed, so we use the technique described in Section 5 to devise appropriate mapping functions. Monitoring reveals that the ns-2 SIP implementation for each of these parameters remembers the value which is received in the first *Invite* message (presumably, it is interpreted as parameters of the connection that is being established). For each of the three parameters, it also remembers the value received in the previous *Request* message when producing the corresponding *Response* reply, but thereafter forgets it.

Following this information, the abstraction mapping is based on six state variables. For the *CallId* parameter, the state variable *firstId* stores the *CallId* parameter of the first *Invite* message, and *lastId* stores the *CallId* parameter value of the most recently received message. The state variables *firstCSeq* and *lastCSeq* store the analogous values for the *CSeq* parameter, and the state variables *firstVia* and *lastVia* store the analogous values for the *Via* parameter. The mapping, based on these state variables, is shown in Tables 2 and 3. The tables say that the input parameters *Via* and *Cseq* are not tested by ns-2, whereas the input parameter *CallId* is compared with the variable *firstId* (in the first *Invite* message) to check if it already is defined. The variable *firstId* is simply assigned the value of the *CallId* parameter. In output *Response* messages, these parameters can take the value received in the first *Invite* message, or the value in the just received message, corresponding to the two abstract values FIRST INVITE and LAST.

The SIP Server does not always respond to each input message, and sometimes responds with more than one message. To stay within the Mealy machine formalism, we introduce the *nil* input symbol which denotes the absense of input, in order to allow sequences of outputs, and the *timeout* output symbol, denoting the absence of output.

	ANY	
$CSeq$	$isInteger(CSeq)$	
Via	$Via.Address = Default$; $isInteger(Via.Branch)$	
	FIRST	LAST
$CallId$	$firstId = \perp \wedge mtype = Invite \vee firstId \neq \perp \wedge CallId = firstId$	$isInteger(CallId)$

Table 2: Mapping table for input messages of SIP Server

Results for SIP. The inference performed by LearnLib needed about thousand membership queries and one equivalence query, and resulted in a model with 10 locations and 70 transitions. This is a minimal abstracted model. For presentation purposes, we have pruned the model as follows: (1) removing transitions triggered by abstract symbols that

	FIRST INVITE	LAST	OTHER
CSeq	$CSeq = firstCSeq$	$CSeq = lastCSeq$	*Other*
Via	$Via = firstVia$	$Via = lastVia$	*Other*
	FIRST	LAST	OTHER
CallId	$CallId = firstId$	$CallId = lastId$	*Other*

Table 3: Mapping table for output messages of SIP Server

have no corresponding concrete symbol: the Mapper will immediately reject these, and react with a distinguished error symbol, (2) removing transitions with empty input and output symbol, i.e., *nil/timeout*, (3) removing a location which has become unreachable after the previous steps. In Figure 2 in Appendix A, we show the abstract model resulting with 9 locations and 48 transitions. For readability, some transitions with same source location, output symbol and next location (but with different input symbols) are merged: the original input method types are listed, separated by a bar (|). Due to space limitations, we have suppressed the (abstract) parameter values. However, the *CallId* parameter is shown by coloring input messages with abstract value FIRST INVITE in blue, the remaining transitions are red. We suppressed all other parameters in the Figure. A full abstract model, showing the abstract values of other output parameters can be found at `http://www.it.uu.se/research/group/testing/sip`, together with a description of the corresponding concrete model.

This model shows how data parameters in messages influences control flow: therefore such a model would be more useful for thorough model-based test generation than a finite-state model where data aspects are suppressed.

Validation. We have compared the resulting models with the descriptions in the corresponding standard documents. According to `RFC 3261` and `RFC 3262`, a trace leading to an established connection must at least contain the transitions *INVITE/100, PRACK/ 200 and ACK/timeout*. As can be seen in Figure 2 in Appendix A, this is indeed the fact for the blue transitions representing the session being established. Moreover no red trace leads to the CONNECTED location, i.e. location 9, indicating that only one session can be established at the same time. The formal description mentions that after receiving a 100, 180 or 183 Response, no further *Invite* Requests should be retransmitted, but there are no strong constraints on noise.

Note that, having reached location 9, the Server does not react to any of the input symbols that we consider. We did not add any input message (e.g., sending some payload) to test whether the SIP Server actually has reached a connected location in location 9. Therefore, there are also other sequences that reach location 9, in which the SIP Server most likely has not opened a connection, but which the inference does not distinguish from a situation with an opened connection.

6.2 TCP

As a second case study, we chose the ns-2 implementation of the Transmission Control Protocol (`RFC 793`). TCP is a transport layer protocol, that provides reliable and ordered delivery of a byte stream from one computer application to another. It is one of the most widely used communication protocols. We consider the connection establishment and termination between Client and Server, but leave out the data transfer phase. As *SUT*, we consider the Server component of the protocol. We consider *Request* messages, which are input to *SUT*, and *Response* messages, which are the output from *SUT*. Messages in TCP are of form *Request/Response*(*SYN*, *ACK*, *FIN*, *SeqNr*, *AckNr*). Among parameters, *SYN*, *ACK*, and *FIN* are flags that define what type of message is sent: *SYN* synchronizes sequences numbers, *ACK* acknowledges the previous *SeqNr*, and *FIN* signals the end of the data transfer phase. *SeqNr* is a number that needs to be synchronized with both sides of the connection, and *AckNr* acknowledges a previous sequence number. Both Client and Server can send messages with the same parameters as defined above. We distinguish these messages by using *Request* for messages that are sent to *SUT* and *Response* for messages that come from *SUT*.

Abstraction Mapping. To define the abstraction mapping, we use information obtained from the standard `RFC 793`. For the flags *SYN*, *ACK*, and *FIN*, there are four valid combinations, shown in the first row of Table 4, thus defining four abstract values SYN, SYN+ACK, ACK, and ACK+FIN. The parameters *SeqNr* and *AckNr* are treated as sequence numbers. In this case, they will be incremented with each transmission round in a session. We therefore define four state variables: *last_SeqNr_sent* and *last_AckNr_sent* are the last values of *SeqNr* and *AckNr*, respectively, that have been transmitted to ns-2 in a *valid Request* message. *last_SeqNr_recd* and *last_AckNr_recd* are the last values of *SeqNr* and *AckNr*, respectively, that have been received from ns-2 in a *valid Response* message: they are initially $\perp$. By *"valid"*, we understand messages that follow the protocol and increment parameters *SeqNr* and *AckNr* appropriately. Using these state variables, the abstraction mappings for *Request* and *Response* messages are shown in the second and third rows of Table 4. The main information is that in VALID messages, each *AckNr* increments the previously sent *SeqNr* at both sides, and that each *SeqNr* should be that of the previously received *AckNr*.

Request Type	SYN	SYN+ACK	ACK	ACK+FIN
	$SYN = 1$	$SYN = 1 \wedge ACK = 1$	$ACK = 1$	$ACK = 1 \wedge FIN = 1$

Request	VALID	INVALID
SeqNr	$SeqNr = last\_AckNr\_recd$	$SeqNr \neq last\_AckNr\_recd$
AckNr	$AckNr = last\_SeqNr\_recd + 1$	$AckNr \neq last\_SeqNr\_recd + 1$

Response	VALID	INVALID
SeqNr	$last\_SeqNr\_recd = \perp \vee SeqNr = last\_SeqNr\_recd$	$last\_SeqNr\_recd \neq \perp \wedge SeqNr \neq last\_SeqNr\_recd$
AckNr	$AckNr = last\_SeqNr\_sent + 1$	$AckNr \neq last\_SeqNr\_sent + 1$

Table 4: Abstraction of parameters in input and output messages of TCP

Results. After inference, LearnLib produced a model with 12 locations and 204 transitions. In order to display the model in this paper, we suppressed transitions with input symbols that have INVALID abstract parameter values are suppressed, and removed transitions labeled by *nil/timeout*, resulting in 11 locations and 33 transitions, shown in Figure 3 of Appendix B. The model is displayed in a shorthand symbolic representation for readability reasons.

Validation. To validate the learned TCP model, we compared it to a reference model[2]. Our setup differs from this one, in that we do not explicitly use triggers, like CONNECT, SEND, LISTEN, CLOSE, and that we do not model a RST message. Furthermore, our learned model reflects only setup and closing of connections which are initiated by the Client communicating with the Server. To include setup and closing initiated by the learned Server, we should also have included the above triggers in the set of input symbols of membership queries. We therefore compare our learned model with the paths in the reference model that correspond to behavior triggered by the Client. We observe the following differences and similarities.

- Connection establishment corresponds between the two models.
- The reference model responds to FIN messages when closing a connection, but the learned model does not respond at all to FIN messages, only to FIN+ACK. This is a choice made by the implementors of the module in ns-2: it is our impression that this is a common practice in TCP implementations.
- In the ns-2 implementation, there are several ways to close connections, which are not represented in the reference model, e.g., by a transition with ACK(VALID,VALID)/*timeout* from location 9 to 11.

7. CONCLUSIONS AND FUTURE WORK

We have presented an approach to infer models of entities in communication protocols, which also handles message parameters. The approach adapts abstraction, as used in formal verification, to the black-box inference setting. This necessitates to define an abstraction together with the local state needed to define it. This makes finding suitable abstractions more challenging, but we have presented techniques for systematically deriving abstractions under restrictions on what operations the component may perform on data. We have shown the feasability of the approach towards inference of realistic communication protocols, by feasibility studies on SIP and TCP, as implemented in the protocol simulator ns-2. The work shows how regular inference can infer the influence of data parameters on control flow, and how data parameters are produced. Thus, models generated using our extension are more useful for thorough model-based test generation, than are finite-state models where data aspects are suppressed. In future work, we plan to supply a library of different inference techniques specialized towards different data domains that are commonly used in communication protocols.

[2]`http://en.wikipedia.org/wiki/File:Tcp_state_diagram_fixed.svg`

Acknowledgement. We are grateful to Falk Howar from TU Dortmund for his generous LearnLib support, and Falk Howar and Bernhard Steffen for fruitful discussions.

8. REFERENCES

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

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

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

[4] T. Berg, B. Jonsson, and H. Raffelt. Regular inference for state machines with parameters. In *FASE 2006*, *LNCS3922*, pages 107–121. Springer, 2006.

[5] T. Berg, B. Jonsson, and H. Raffelt. Regular inference for state machines using domains with equality tests. In *FASE 2008 LNCS 4961*, pages 317–331. Springer, 2008.

[6] M. Broy, B. Jonsson, J.-P. Katoen, M. Leucker, and A. Pretschner, editors. *Model-Based Testing of Reactive Systems*, volume 3472 of *LNCS*. Springer Verlag, 2004.

[7] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. *Journal of the ACM*, 50(5):752–794, 2003.

[8] J. Cobleigh, D. Giannakopoulou, and C. Pasareanu. Learning assumptions for compositional verification. In *Proc. TACAS '03*, *LNCS 2619*, pages 331–346. Springer Verlag, 2003.

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

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

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

[12] O. Grinchtein, B. Jonsson, and M. Leucker. Inference of timed transition systems. *Electr. Notes Theor. Comput. Sci.*, 138(3):87–99, 2005.

[13] O. Grinchtein, B. Jonsson, and P. Pettersson. Inference of event-recording automata using timed decision trees. In *Proc. CONCUR 2006*, *LNCS 4137*, pages 435–449, 2006.

[14] A. Groce, D. Peled, and M. Yannakakis. Adaptive model checking. In *Proc. TACAS '02*, *LNCS 2280*, pages 357–370. Springer Verlag, 2002.

[15] R. Groz, K. Li, A. Petrenko, and M. Shahbaz. Modular system verification by inference, testing and reachability analysis. In *TestCom/FATES*, *LNCS 5047*, pages 216–233, 2008.

[16] A. Hagerer, H. Hungar, O. Niese, and B. Steffen. Model generation by moderated regular extrapolation. In *Proc. FASE '02*, *LNCS 2306*, pages 80–95. Springer Verlag, 2002.

[17] T. Henzinger, R. Jhala, R. Majumdar, and G. Sutre.

Lazy abstraction. In *Proc.* 29^{th} *ACM Symp. on Principles of Programming Languages*, pages 58–70, 2002.

[18] A. Huima. Implementing conformiq qtronic. In A. Petrenko, M. Veanes, J. Tretmans, and W. Grieskamp, editors, *Proc. TestCom/FATES, 2007*, *LNCS 4581*, pages 1–12, 2007.

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

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

[21] K. Li, R. Groz, and M. Shahbaz. Integration testing of distributed components based on learning parameterized I/O models. In, *FORTE 2006*, *LNCS 4229*, pages 436–450, 2006.

[22] C. Loiseaux, S. Graf, J. Sifakis, A. Boujjani, and S. Bensalem. Property preserving abstractions for the verification of concurrent systems. *Formal Methods in System Design*, 6(1):11–44, 1995.

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

[24] O. Niese. An integrated approach to testing complex systems. Technical report, Dortmund University, 2003. Doctoral thesis.

[25] D. Peled, M. Y. Vardi, and M. Yannakakis. Black box checking. In FORTE/PSTV 1999, pages 225–240, Beijing, China, 1999. Kluwer.

[26] H. Raffelt, B. Steffen, and T. Berg. Learnlib: a library for automata learning and experimentation. In *FMICS '05*: pages 62–71, New York, NY, USA, 2005. ACM Press.

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

[28] M. Shahbaz, K. Li, and R. Groz. Learning and integration of parameterized components through testing. In *TestCom/FATES 2007*, *LNCS 4581*, pages 319–334. Springer, 2007.

[29] G. Shu and D. Lee. Testing security properties of protocol implementations - a machine learning based approach. In *Proc. ICDCS'07*. IEEE Computer Society, 2007.

APPENDIX

A. SIP MODEL

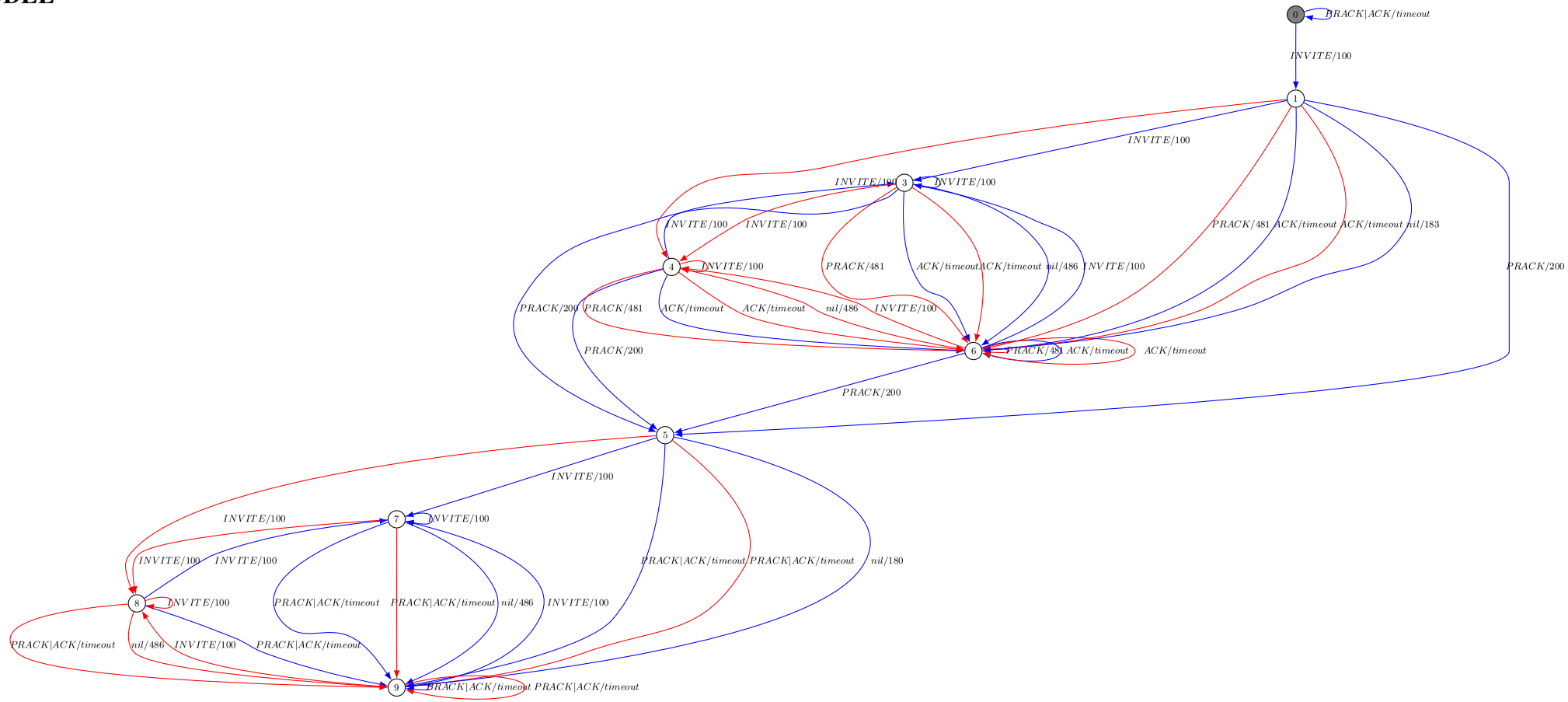


Figure 2: Full SIP model

B. TCP MODEL

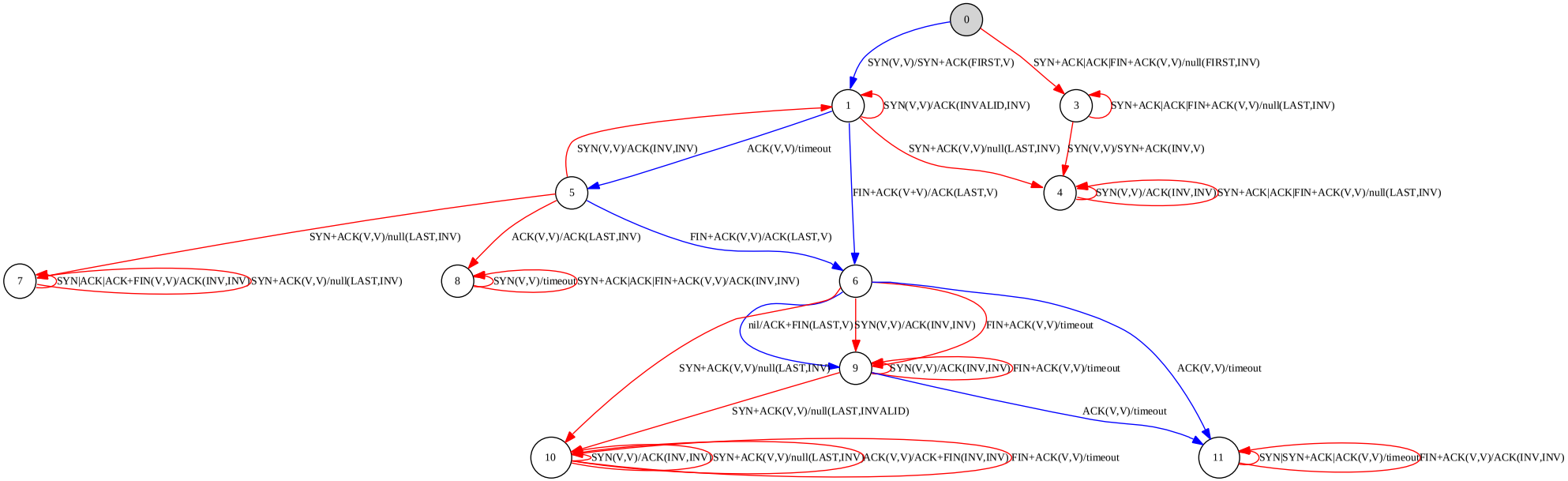


Figure 3: TCP model, V represents the VALID equivalence class and INV represents the INVALID equivalence class. The null message describes that none of the bits SYN, ACK or FIN is set

A.4 Inferring Compact Models of Communication Protocol Entities

This paper contains a worked-out presentation of techniques, mentioned in Section 3.3, for restructuring the representation of an unstructured finite-state machine, in order make it more compact, possibly as a means to abstract less important details. In the paper, we report on an evaluation of this technique by applying it to the Mobile Arts Advanced Mobile Location Center (A-MLC) protocol, which is a commercially available middleware protocol that allows mobile network operators to provide presence information from the GSM/UMTS network.

Inferring Compact Models of Communication Protocol Entities

Therese Bohlin
Dept. IT, Uppsala University
Box 337, SE-751 05, Sweden
Email: thereseb@it.uu.se

Bengt Jonsson
Dept. IT, Uppsala University
Box 337, SE-751 05, Sweden
Email: bengt@it.uu.se

Siavash Soleimanifard
Dept. IT, Uppsala University
Box 337, SE-751 05, Sweden
Email: siavashs@kth.com

***Abstract*—An obstacle to wider adoption of model-based approaches to verification and validation of communication protocols is that modeling is a labor-intensive activity. To provide automated support for modeling, we aim to develop techniques that construct models of communication protocol entities from observations of their external behavior. Such techniques are useful when introducing a model-based approach in the test process, for regression testing, and for components with no access to source code. Models can be constructed using regular inference (aka automata learning). In this paper, we address the problem that existing regular inference techniques produce "flat" state machines, whereas practically useful protocol models structure the internal state in terms of control locations and state variables, and describes dynamic behavior in a suitable (abstract) programming notation. We present techniques for restructuring the representation of an unstructured finite-state machine, Our transformation introduces state variables, based on a certain amount of user guidance. Thereafter it groups states with "similar control behavior" into control locations, and finally construct a description of dynamic behavior, whose control structure is obtained by a decision tree generation algorithm. We have applied parts of our approach to an executable state machine specification of the Mobile Arts Advanced Mobile Location Center (A-MLC) protocol and evaluated the results by comparing them to the original executable specification.**

This work supported in part by EC Proj. 231167 (CONNECT). Siavash Soleimanifard is currently affiliated with Dept. TCS, Royal Institute of Technology , SE-100 44, Stockholm, Sweden

***Keywords*-protocol modeling; regular inference; protocol testing**

I. Introduction

Model-based techniques for verification, testing, and validation of commmunication protocols, including model checking and model-based testing [1], have witnessed drastic advances in the last decades. They require access to a formal model that specifies the behavior of protocol entities, which ideally should be developed during specification and design. However, the construction of models typically requires significant manual effort, implying that in many cases no such model is available, or becomes outdated as the system evolves over time. It is therefore important to develop automated techniques that support the task of producing models, e.g., models that reflect the behavior of an existing protocol implementation. Such techniques would be highly useful for producing models of standardized protocols, for introducing model based testing techniques to replace manual testing of an existing product, for regression testing, etc. A potential approach is to use program analysis to construct models from source code (e.g., [2], [3]). However, many system components, such as library modules, or third-party components, often do not allow analysis of source code. We will therefore focus on techniques for constructing models from observations of their external behavior.

The construction of models from observations of component behavior can be performed using regular inference (aka automata learning) techniques [4], [5], [6], [7]. This class of techniques has recently started to get attention in the testing and verification community, e.g., for regression testing of telecommunication systems [8], [9], for integration testing [10], [10], [11], and for combining conformance testing and model checking [12], [13]. In regular inference, a finite-state machine (or a regular language) is constructed from the answers to a set of *membership queries*, each of which observes the component's output in response to a certain input string. Given "enough" membership queries, the constructed automaton will be a correct model of the observed component.

Our overall goal is to construct models of entities in communication protocols, which can be readily understood and maintained by protocol designers and test engineers. Manually constructed models of protocol behavior facilitate understanding by describing messages as consisting of a message type with a number of parameters, by representing the internal states of the entity in terms of control locations and state variables, and by describing the reaction to incoming messages by a change of location and variable transformation in some suitable language. This style of modeling is supported by several formalisms, such as UML state diagrams [14].

A serious obstacle to constructing structured models from observations is that existing regular inference techniques produce "flat" state machines, in which neither states nor transitions have any structure. In this paper, we therefore present techniques for restructuring the representation of an unstructured finite-state machine, in order to make it readily understandable by humans. Since there are many ways to restructure state-machine descriptions, and since most likely

there is no unique optimal restructuring, our techniques will need some light guidance by a user or expert, giving general principles for forming state variables and control locations. Based on such principles, our transformation first equips the "flat" state machine with state variables. Thereafter it groups states with similar control behavior into control locations. Finally, the "flat" description of the reaction to received messages is transformed into a compact description in the chosen coding language; we have chosen the intuitive formalism of decision trees, which can be generated by well-developed tools.

We evaluate our techniques by applying them to the Mobile Arts Advanced Mobile Location Center (A-MLC) protocol, which is a commercially available middleware protocol that allows mobile network operators to provide presence information from the GSM/UMTS network. We have access to an executable specification of A-MLC, which is structured for human readability by developers and testers of the protocol. This makes it a suitable object for evaluation, since we can both observe its reaction to a large number of input sequences, as well as compare the results of our restructuring to the structure of the executable specification. We present the results our comparison.

In summary, the main contributions of this paper are:

- A novel approach to construct structured state machine models of communication protocol entities from observations, based on regular inference techniques. techniques
- The construction of a model of a large industrial protocol by regular inference techniques: our model has 1560 different messages and 44 states.

Related Work: Regular inference techniques have been used for verification and test generation, e.g., to create models of environment constraints with respect to which a component should be verified [15], for regression testing to create a specification and a test suite [8], [9], to perform model checking without access to source code or formal models [13], [12], for program analysis [16], and for formal specification and verification [15]. Groz, Li, and Shahbaz extend regular inference to Mealy machines with a finite subset of input and output symbols from the possible infinite set of symbols [10], [17], [11]. Mariani and Pezzé use inference in integration testing of commercial off the shelf components [18]. They infer two separate models, which are not connected: one for the finite-state control, and the others being a relation on the parameters in each interaction. They use different inference techniques for each type of model. In previous work [19], we presented an optimization of regular inference to cope with models where the domains of the parameters are booleans. We have also presented an approach using regular inference, in which systems have input parameters from a potentially infinite domain and parameters may be stored in state variables for later use [20].

Organization of Paper: In next section, we review Mealy machines and a formalism for structured representation of Mealy machine models. In Section III we review regular inference algorithm for Mealy machines by Niese [21], and in Section III-B we present our transformation for generating a structured description of a Mealy machine. In Section IV we describe how we implemented our techniques, and in Section V we describe their application so the A-MLC protocol, which is evaluated in Section VI. Conclusions and proposed future work is in Section VII.

II. Mealy Machines

A. Flat Mealy Machines

We model communication protocols as Mealy machines. A *Mealy machine* is a tuple $\mathcal{M} = \langle \Sigma_I, \Sigma_O, Q, q_0, \delta, \lambda \rangle$ where Σ_I is a nonempty set of *input symbols*, Σ_O is a finite nonempty set of *output symbols*, Q is a nonempty set of *states*, $q_0 \in Q$ is the *initial state*, $\delta : Q \times \Sigma_I \to Q$ is the *transition function*, and $\lambda : Q \times \Sigma_I \to \Sigma_O$ is the *output function*. Elements of Σ_I^* and Σ_O^* are called *input strings* and *output strings*, respectively.

An intuitive interpretation of a Mealy machine is as follows. The machine interacts with its environment by receiving input symbols and producing output symbols. At any point in time, the machine is in some state $q \in Q$. When the machine receives an input symbol $a \in \Sigma_I$, it responds by producing an output symbol $\lambda(q, a)$ and moving to a new state $\delta(q, a)$. We let $q \xrightarrow{a/b} q'$ denote that $\delta(q, a) = q'$ and $\lambda(q, a) = b$. We call $q \xrightarrow{a/b} q'$ a *transition* of $\mathcal{M}$. For the implementation and experiments in Sections IV and V, we have slightly generalized Mealy machines to allow production of a *sequence* of output symbols in each transition. This generalization is straightforward: to keep the presentation simple, we here assume one output symbol per transition.

We extend the transition and output functions from input symbols to input strings in the standard way, by defining:

$$\begin{array}{lcllcl} \delta(q, \varepsilon) & = & q & \lambda(q, \varepsilon) & = & \varepsilon \\ \delta(q, ua) & = & \delta(\delta(q, u), a) & \lambda(q, ua) & = & \lambda(q, u)\lambda(\delta(q, u), a) \end{array}$$

We define $\lambda_{\mathcal{M}}(u) : \Sigma_I \to \Sigma_O$ by $\lambda_{\mathcal{M}}(u) = \lambda(q_0, u)$, for $u \in \Sigma_I^*$. Two Mealy machines $\mathcal{M}$ and $\mathcal{M}'$ with the same input alphabets are *equivalent* if $\lambda_{\mathcal{M}} = \lambda_{\mathcal{M}'}$.

Note that the Mealy machines that we consider are *completely specified*, meaning that at every state the machine has a defined reaction to every input symbol in Σ_I, i.e., δ and λ are total. They are also *deterministic*, meaning that for each state q and input a exactly one next state $\delta(q, a)$ and output symbol $\lambda(q, a)$ is possible.

B. Symbolic Representation

In order to be understandable, Mealy machine models of realistic protocols must be structured. Structure is typically

imposed by letting messages consist of a message type with a number of parameters, by describing the internal states of the entity in terms of control locations and state variables, and by describing the reaction to incoming messages as a combination of a change of location and a transformation on state variables.

Structuring of input and output symbols will be achieved by assuming finite sets, I and O, of (input and output) *action types*. Each action type α has a certain *arity*, which is a tuple of finite domains $\mathcal{D}_{\alpha,1}, \ldots, \mathcal{D}_{\alpha,n}$ (where n depends on α), each of which contains the possible values of the corresponding parameter. We will use d, d_1, d_2, etc. to range over data values. The set Σ_I of input symbols is now the set of terms of form $\alpha(d_1, \ldots, d_n)$, where each $d_i \in \mathcal{D}_{\alpha,i}$ is in the appropriate domain. The set of *output symbols* Σ_O is defined analogously. We will use β to range over output action types. We write $\overline{d}$ for $d_1, \ldots, d_n$.

To structure the representation of states, we will use a finite set L of locations, one of which is the *initial location*, and a tuple $v_1, \ldots, v_k$ of *state variables*, each of which ranges over a finite domain and has a unique initial value. The set of states of a mealy machine is now the set of tuples $\langle l, \mathsf{v}_1, \ldots, \mathsf{v}_k \rangle$, where $l \in L$ is a location, and $\mathsf{v}_1, \ldots, \mathsf{v}_k$ is a tuple of values of the state variables $v_1, \ldots, v_k$. In the following, we will write $\overline{v}$ for $v_1, \ldots, v_k$ and $\overline{\mathsf{v}}$ for $\mathsf{v}_1, \ldots, \mathsf{v}_k$

To provide a structured representation of the transition and output functions, we must employ a suitable formalism to express transformation of state variables and generation of output symbols. We will here use a simple formalism with constructs for selection, assignment, and output. We will use a finite set of *formal parameters*, ranged over by $p_1, p_2, \ldots$, which will serve as local variables to which values of parameters in input symbols are bound. Let an *expression* be either a formal parameter, a state variable or a data value. Let an *action expression* be an expression of one of the following forms.

- **output** $\beta(d_1, \ldots, d_m)$; **nextloc** l;
  produces the output symbol $\beta(d_1, \ldots, d_m)$ and changes the current location to l.
- $v_1, \ldots, v_k := e_1, \ldots, e_k$; *actexp*
  simultaneously assigns the values of the expressions e_i to the variables v_i for $i = 1, \ldots, k$, and thereafter carries out whatever the action expression *actexp* does.
- **case** e **of** d_1 : *actexp*$_1$ $\cdots$ d_k : *actexp*$_k$
  evaluates the expression e, and if the result is d_i for some i in $1, \ldots, k$, it carries out whatever the action expression *actexp*$_i$ does. It is assume that $d_1, \ldots, d_k$ are distinct, and cover the possible values of e. Sometimes we use **if** e **then** *actexp*$_1$ **else** *actexp*$_2$ instead of
  case e **of** true : *actexp*$_1$ false : *actexp*$_2$

The reaction to input symbols can now be described by providing for each location $l \in L$ and each input action type $\alpha \in I$ an expression of form

in location l **when** $\alpha(p_1, \ldots, p_m)$ *actexp*

where *actexp* is an action expression which may use the formal parameters $p_1, \ldots, p_m$. For each input symbol $\alpha(\overline{d})$, the action expression will follow exactly one branch leading to an action expression of form **output** $\beta(\overline{d}')$; **nextloc** l'; this implies that for the transition and output functions we have

- $\delta(\langle l, \mathsf{v}_1, \ldots, \mathsf{v}_k \rangle, \alpha(\overline{d})) = \langle l', \mathsf{v}'_1, \ldots, \mathsf{v}'_k \rangle$, and
- $\lambda(\langle l, \mathsf{v}_1, \ldots, \mathsf{v}_k \rangle, \alpha(\overline{d})) = \beta(\overline{d}')$,

for all tuples $\mathsf{v}'_1, \ldots, \mathsf{v}'_k$ of values of $v_1, \ldots, v_k$, where $\mathsf{v}_1, \ldots, \mathsf{v}'_k$ is the result of performing the assignment statements on that branch.

To summarize, a Mealy machines can be presented symbolically by providing finite sets I of *input action types*, O of *output action types*, and L of *locations*, an *initial location* $l_0 \in L$, a tuple $v_1, \ldots, v_k$ of state variables, and an expression of form

in location l **when** $\alpha(p_1, \ldots, p_m)$ *actexp*

for each location $l \in L$ and each input action type $\alpha \in I$.

In Figure 1, we show a possible action expression, from an idealized version of the receiver in the alternating bit protocol, in which we have action types `Data` and `Ack`, each of which has a bit (either 0 ot 1) as a sequence number.

```
in location rec
    when Data(sn)
        case (sn) of
            0 : output Ack(0); nextloc rec
            1 : output Ack(1); nextloc rec
end
```

Figure 1. Example syntax defining part of reciever in alternating bit protocol.

III. Inference of Symbolic Mealy Machines

In this section, we present our approach for inferring a symbolic representation of a Mealy machine model of the behavior of an entity in a communication protocol, by observing its responses to selected input strings. We will hereafter refer to the given protocol entity as the System Under Test (SUT). We assume that the SUT can be modeled as a Mealy machine, and that its input and output action types as well as their arities, are known.

The problem of inferring a model of the SUT naturally decomposed into two subproblems:

- inferring a flat Mealy machine $\mathcal{M}$ which models the behavior of SUT, and
- generating a symbolic representation of $\mathcal{M}$.

For the first subproblem we use an adaptation of the L^* algorithm [4] to Mealy machines, due to Niese [21]. For the second subproblem, we have developed a technique for transforming a Mealy machine into an equivalent symbolic description by introducing state variables, control locations, and action expressions. Each subproblem is described in more detail in the following subsections.

A. Inference of Mealy Machines

In this subsection, we review the L^* algorithm [4], which was originally formulated for inferring regular languages, in the form of its adaptation for Mealy machines [21]. The L^* algorithm initially knows nothing about the Mealy machine $\mathcal{M}$ except its input and output alphabets. It infers a Mealy machine model of $\mathcal{M}$ by asking a sequence of queries. There are two kinds of queries.

- A *membership query* consists in asking what is $\mathcal{M}$'s output in response to an input string $u \in (\Sigma_I)^*$.
- An *equivalence query* consists in asking whether a hypothesized Mealy machine $\mathcal{H}$ supplied, in the query, is correct, i.e., whether $\mathcal{H}$ is equivalent to $\mathcal{M}$. The query will be answered by answer yes if $\mathcal{H}$ is correct, or else by a *counterexample*, which is a string $u \in (\Sigma_I)^*$ such that $\lambda_{\mathcal{M}}(u) \neq \lambda_{\mathcal{H}}(u)$.

The typical behavior of the L^* algorithm is to ask a sequence of membership queries until it can build a "stable" hypothesis $\mathcal{H}$ from the answers. It then makes an equivalence query to find out whether $\mathcal{H}$ is equivalent to $\mathcal{M}$. If the result is successful, the algorithm has succeeded, otherwise it uses the returned counterexample to perform subsequent membership queries until converging at a new hypothesized Mealy machine, which is supplied in an equivalence query, and so on. The algorithm is guaranteed to terminate after at most n such equivalence queries, where n is the number of states of $\mathcal{M}$, having posed in total $O(m|\Sigma_I|n^2)$ membership queries, where m is the length of the longest counterexample returned in some equivalence query [4].

When transforming the conceptual framework of the L^* algorithm to a practical setting, some issues need to be considered. In order to perform a sequence of membership queries, one must be able to reliably reset the SUT to an initial state between each query. A perfect realization of an equivalence query must involve analysis of the source code, or equivalent, of the SUT. In a black-box setting, where source code is not available, there is in general no perfect implementation of equivalence queries. In the case that there is a known upper bound on the number of states of $\mathcal{M}$ is known, (typically large) conformance test suites (as described in, e.g., [22], [23]) can conclusively settle equivalence queries. In practice, equivalence queries are often approximated by large random test suites and/or by monitoring the SUT under a long period of time.

B. Generating Symbolic Representation of Mealy Machines

We have developed a transformation that transforms a Mealy machine $\mathcal{M}$ into a symbolic representation, as described in Section II-B, in order that the model be readily understood by human designers or testers. Our transformation must

- represent the states of $\mathcal{M}$ in terms of control locations and state variables, and
- represent the transition and output function of $\mathcal{M}$ in terms of action expressions.

A given "flat" Mealy machine has several equivalent symbolic representation, among which there is probably no unique "most intuitive" one. Therefore, our tranformation needs user-supplied guidance, which typically require a certain amount of understanding of the protocol, but not at all a complete knowledge about its behavior. Let us first describe how we transform states, thereafter how we generate action expressions.

1) Transforming the Representation of States: Intuitively, our transformation will construct state variables that record information in received parameters of input symbols that may influence future behavior, and control locations that capture "high level control" aspects of behavior. The transformation is based on some user-supplied guidance, as follows

- The user must supply a set $\overline{v} = v_1, \ldots, v_k$ of state variables. To describe the information captured by the state variables, the user should also, for each variable $v_i \in \overline{v}$ and input action type α, supply an expression $e_{v_i,\alpha}$ which describes how the variable v is updated when input symbols of form $\alpha(p_1, \ldots, p_n)$ are received. Typically, the user need not provide this information in detail; instead it can be derived automatically from a user-supplied general principle, e.g., to store, for each input action type α, the most recent values of the parameters of an input symbol of form $\alpha(d_1, \ldots, d_n)$. With this principle, the Alternating bit protocol in Figure 1 would have a variable for the parameter of the action type `Data` (called, say $v_{\texttt{Data.sn}}$), which is assigned the parameter value `sn` in action expressions triggered by the action type `Data`, and not updated in other action expressions.
- The user should supply a criterion for merging states with the "same control behavior" into control locations. We will here use the criterion that all states that may be reached by a particular sequence of input and output action types should be in the same location. (* Explain this principle and supply an example *)

Define an *extended state* as a pair $\langle q, \nabla \rangle$, where $q \in Q$ is a state of $\mathcal{M}$ and ∇ is a tuple of values of the variables $\overline{v}$. For each state q of $\mathcal{M}$, there are in general several reachable extended states of form $\langle q, \nabla \rangle$: differences in ∇ correspond to different input strings that cause q to be reached. Define an

extended transition as a transition between extended states of form $\langle q, \nabla \rangle \xrightarrow{\alpha(\overline{d})/\beta(\overline{d}')} \langle q', \nabla' \rangle$, such that $\mathcal{M}$ has a transition $q \xrightarrow{\alpha(\overline{d})/\beta(\overline{d}')} q'$ and ∇' is obtained from ∇ and $\alpha(\overline{d})$ by appropriately updating state variables.

We can now form control locations as sets of extended states with "same control behavior". Thie procedure will differ, depending on how "same control behavior" is defined. Let us consider the case that extended states that may be reached by the same sequence of input and output action types should be in the same location. We must then construct a set $Locs$ of locations, each of which is a set of extended states. For each location l in $Locs$, and each pair α, β of input and output action types, $Locs$ should contain a successor location containing the targets $\langle q', \overline{\nu}' \rangle$ of all extended transitions $\langle q, \overline{\nu} \rangle \xrightarrow{\alpha(\overline{d})/\beta(\overline{d}')} \langle q', \overline{\nu}' \rangle$ from l with action types α, β. We record this structure by generating a set of edges of form $l \xrightarrow{\alpha/\beta} l'$ between locations. The locations and edges can be constructed by a technique similar to the subset construction for nondeterministic finite automata: starting from an initial location, successors of already constructed locations are created based on the pair of input-output action types that reach them. Such an algorithm is described in Algorithm 1.

The algorithm maintains two sets of locations; $Locs$ accumulates the set of formed locations, whereas $TempLocs$ is a set of locations whose successor locations remain to be constructed, and a set $Edges$ of generated edges. Algorithm 1 starts by forming the initial location $l_0 \in L$, containing the extended state formed from the initial state q_0 and initial values $\overline{\nu}_0$ of variables. The algorithm then iteratively picks some location l from $TempLocs$; for each pair α, β of input and output action types it constructs a new location as containing the targets of all transitions $\langle q, \overline{\nu} \rangle \xrightarrow{\alpha(\overline{d})/\beta(\overline{d}')} \langle q', \overline{\nu}' \rangle$ with the same source location, input, and output action types, and adds it to $TempLocs$, and also adds $l \xrightarrow{\alpha/\beta} l'$ to $Edges$. The process of forming locations continues iteratively until all locations in $TempLocs$ have been used for forming successor locations. The process is guaranteed to terminate since the set of extended states is finite.

During Algorithm 1, we additionally merge locations which are "similar", in the sense that they share an extended state, since presumably their future behavior is rather similar. Such new formed locations are added to $TempLocs$ to properly generate their successors. However, we must not merge locations if as a result they will contain two extended states $\langle q, \overline{\nu} \rangle$ and $\langle q', \overline{\nu} \rangle$ with the same variable values but different control state, since action expressions (which can only test values of variables) will not be able to distinguish the difference in future behavior between q and q'.

2) Generating Action Expressions: It remains to generate an action expression for each location l and input action type α, which distinguish between the (slightly) different behaviors of different extended states in tha location. Our transformation generates action expressions as decision tree structures of **case** expressions, each of which tests some input parameters in $p_1, \ldots, p_n$ or state variable in $\overline{v}$, reaching appropriate leaves of form

$$v_1, \ldots, v_k := e_{v_1,\alpha}, \ldots, e_{v_k,\alpha} \ ; \ \textbf{output} \ \beta(\overline{d}') \ ; \ \textbf{nextloc} \ l' \ .$$

Here $v_1, \ldots, v_k := e_{v_1,\alpha}, \ldots, e_{v_k,\alpha}$ is a multiple assignment statement that updates each v_i by the appropriate user-supplied expression $e_{v_i,\alpha}$, $\beta(\overline{d}')$ is an output symbol, and where l' is a next location.

The decision tree structure of the **case** expressions in the action expression of location l and input action type α should be constructed so that whenever is is presented with values $\overline{d}$ of input parameters $\overline{p}$ and values $\overline{\nu}$ of state variables $\overline{v}$, such that

$$\langle q, \overline{\nu} \rangle \xrightarrow{\alpha(\overline{d})/\beta(\overline{d}')} \langle q', \overline{\nu}' \rangle$$

is an extended transition from some $\langle q, \overline{\nu} \rangle \in l$ to some $\langle q', \overline{\nu}' \rangle \in l'$ with $l \xrightarrow{\alpha/\beta} l' \in Edges$, then the decision tree should reach the output symbol $\beta(\overline{d}')$ and location l'. There are well-developed algorithms to generate decision trees from a set of such constraints, among the most well-known being ID3 [24], [25]. The ID3 algorithm generates a minimal decision tree from a given set of examples (in our case generated from extended transitions as above).

The generated decision tree structures are typically much more compact than the set of "flat" Meay machine transitions that they cover, in particular if the input alphabet is large. The generated action expression can be further optimized by simple transformations. For instance, the insertion of a multiple assignment statement at each leaf of the structure of **case** expressions may blow up its size.

Algorithm 1 MAKELOCATIONS

```
1:  Locs := ∅;
2:  Edges := ∅;
3:  TempLocs := {⟨q_0, ν̄_0⟩};
4:  while TempLocs ≠ ∅ do
5:      choose l ∈ TempLocs;
6:      for all pairs α, β do
7:          l' := {⟨q', ν̄'⟩ : ∃⟨q, ν̄⟩ ∈ l , ∃d̄, d̄' .
8:                              ⟨q, ν̄⟩ --α(d̄)/β(d̄')--> ⟨q', ν̄'⟩};
9:          if (l' ≠ ∅ ∧ l' ∉ (Locs ∪ TempLocs)) then
10:             TempLocs := TempLocs ∪ l';
11:             Edges := Edges ∪ l --α/β--> l';
12:         end if
13:     end for
14:     TempLocs := TempLocs \ l;
15:     Locs := Locs ∪ l;
16: end while
```

This size can mostly be reduced by standard code motion transformations that move assignments towards the root of the tree structure while merging them and removing unnecessary assignments.

IV. Implementation

We have implemented the technique for generation of symbolic representation of Mealy machines described in Section III-B. Our implementation gets a description of a "flat" Mealy machine together with user-supplied criteria for forming state variables and control locations, and generates a symbolic representation of the Mealy machine.

Algorithm 1 forms locations as sets of extended states that may be reached by the same sequence of input and output action types, implying that the set of targets $\langle q', \overline{\mathrm{v}}' \rangle$ of extended transitions $\langle q, \overline{\mathrm{v}} \rangle \overset{\alpha(\overline{d})/\beta(\overline{d}')}{\longrightarrow} \langle q', \overline{\mathrm{v}}' \rangle$ from a formed location with a particular pair α, β of input and output action types should be in the same successor location. In addition to this criterion for forming successor control locations, our implementation also accepts the criterion that sucessor locations of extended transitions $\langle q, \overline{\mathrm{v}} \rangle \overset{\alpha(\overline{d})/\beta(\overline{d}')}{\longrightarrow} \langle q', \overline{\mathrm{v}}' \rangle$ with the same output action type β should be in the same location, as well as the criterion that extende transitions with the same output symbol $\beta(\overline{d}')$ should be in the same location. For these criteria, Algorithm 1 and the generation of action expressions are change accordingly. Users can try these alternative criteria to see whether the resulting structure better suits their purpose.

In our tool we use an implementation of ID3 provided by Weka (Waikato Environment for Knowledge Analysis) a data mining tool developed at the University of Waikato, New Zealand, and distributed under the Gnu Public Licence. It includes a wide variety of state-of-the-art algorithms of data mining and machine learning which are implemented in Java [26].

V. Experiments

In this section, we described the use of our implemented technique for generating a model of the Mobile Arts Advanced Mobile Location Center (A-MLC) protocol. We have chosen A-MLC because we have access to an executable specification, which has been created to be understood by developers and testers. This makes A-MLC suitable for our experimentation since we can both execute large numbers of membership queries and can compare our resulting model with the provided one.

A. The A-MLC Protocol and its Executable Specification

The A-MLC protocol is a middle-ware product that allows Mobile Network Operators to provide presence information from the GSM/UMTS network. The supported presence information includes details about the location, present status, and capabilities of mobile devices. For example, a taxi switchboard application may want to know where a calling customer is located to send the closest available taxi car to the customer. A-MLC is commercially available and has been deployed at several Telecom operators within Europe.

Applications using the A-MLC communicate with A-MLC via the Mobile Location Protocol (MLP), a standard XML-based application-level protocol for obtaining the position of mobile devices utilizing HTTP over IP. On a request from an application to A-MLC to provide presence information, A-MLC uses the Mobile Application Part (MAP) layer in the global standard for telecommunications (SS7) protocol stack to communicate with the GSM/UMTS network, from which the information is retrieved.

The implementation of A-MLC was made mainly in Erlang, utilizing Erlang Open Telecom Platform [27] - a large collection of libraries for Erlang. It consists of approximately 130,000 lines of Erlang code and 5,500 lines of C code.

The originators of the A-MLC protocol have written a functional specification of the protocol in order to generate high-quality test suites [28]. The specification essentially has the form of a symbolically represented Mealy machine, and captures all traffic sequences through A-MLC via the MLP protocol towards an application, and all relevant MAP operations towards the GSM network. Lower level protocols in the IP and SS7 stacks are not part of the specification. Likewise, no operation and maintenance interface (counters, alarms, GUI etc.) are part of the specification. The specification models the behavior of an individual protocol request. Moreover, does not model the reception of "illegal" or traffic sequences that should not be provided by the environment of the modeled A-MLC protocol. The interaction between concurrent requests is also not modeled, since it is minimal, and since Erlang's light-weight threads make it easy to reliably handle large numbers of concurrent requests.

We have used an executable version of this specification, implemented using the Erlang behavior module *gen_fsm* (Generic Finite State Machine Behavior), as the SUT. This executable specification will simply crash, if an "illegal" input string, which should not be provided by the protocol environment, is received.

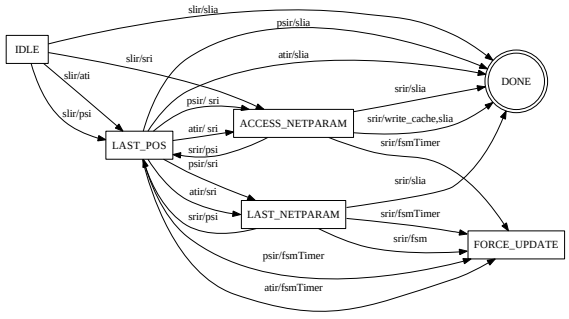


Figure 2. Part 1 of executable specification's structure related to inferred Mealy machine shown in figure 5.

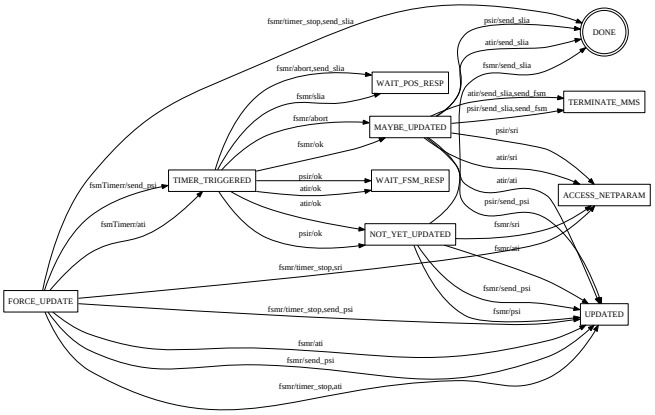


Figure 3. Part 2 of executable specification's structure related to inferred Mealy machine shown in figure 5.

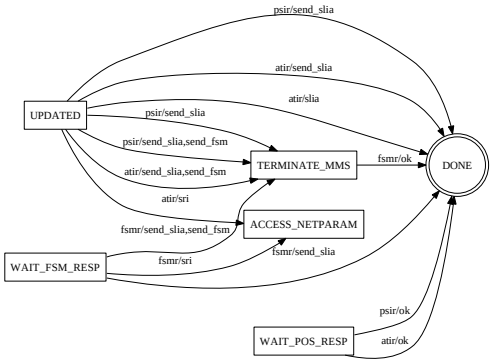


Figure 4. Part 3 of executable specification's structure related to inferred Mealy machine shown in figure 5.

An overview of the executable specification is shown in Figures 2, 3 and 4. In these figures, we show the control locations, and for each location l' which can be reached from location l by receiving an input action type α and producing and output action type β, we display an edge $l \xrightarrow{\alpha/\beta} l'$, i.e., the figures show the graph corresponding to the set $Edges$ generated by Algorithm 1. We have divided the overview of the specification into three sub-figures for readability. The specification consists of 13 control states, 13 state variables, and 10 input action types with different arities. In addition, there are 10 variables that are assigned only when configuring a session. We model these by adding 10 extra parameters to an initial input symbol, which are used to assign values to these variables. This causes number of state variables to become 23.

B. Creating a Model of A-MLC by Regular Inference

For the inference experiment, we defined small domains for the values of parameters in input symbols, in order to be able to carry out enough membership queries to complete the inference process. For most parameters, these domains were already small in the original specification (typically $2--4$ values), and for others, we could choose a representive sample that would allow coverage of the entire specification. In one case, however, this reduction made a part of the model unreachable (as described in Section VI): for input parameter `status` of `atir` action type which can assume values `not_reachable`, `reachable` and `undefined`, we only used the value `not_reachable`. In all, this resulted in an input alphabet of 1560 input symbols.

To construct a Mealy machine model of the executable specification of the A-MLC protocol by regular inference, we use the LearnLib tool [29], developed at the University of Dortmund, which has an efficient implementation of the L^* algorithm. This tool provides several different realizations of equivalence queries, including conformance tests suite generated by the Vasilevsky-Chow algorithm[22], [23]), and random test suites of user-controlled size.

To enable communication between LearnLib and the executable specification, we implemented an intermediate layer, between LearnLib and the executable specification, which monitors the executable specification and capture the occasions when it crashes on illegal input strings. The intermediate layer simply forwards messages from LearnLib to the executable specification, and vice versa, unless it is notified that the executable specification crashed, which it models to LearnLib by means of an introduced error output symbol.

To generate a symbolic representation, we then applied our implementation of the transformation in III-B to the flat Mealy machine generated by LearnLib.

VI. Results

In this section we describe the results of our experiment, ans evaluate the results by comparing the model we obtained with the original executable specification.

Applying LearnLib to the executable specification resulted in a Mealy machine with 44 states. It took about 43 hours to complete the inference, during which LearnLib asked about 175 million membership queries. As equivalence oracle, LearnLib used a test suite of 1000 randomly generated tests of length 10. The Mealy machine model passed the test suite in the first (and only) equivalence query, meaning that the randomly generated test suite found no deviation, and the inference algorithm halted.

The generated "flat" Mealy machine exhibits 59 different sequences of symbols on its transitions, formed from 11 distinct sequences of output action types. A part of the Mealy machine is shown in Figure 5 as an example (the complete Mealy machine can not been shown because of space limitation). In the figure states $q0$ till $q8$ can be seen with some of their transitions, which we have labeled by pairs of input-output action types, e.g., the dashed transition from state $q0$ to state $q1$, shown in Figure 5, contains `slir` input action type and `slia` and `ati` output action types.

Later in this section in table I we correlate the states of this figure with control states of executable specification.

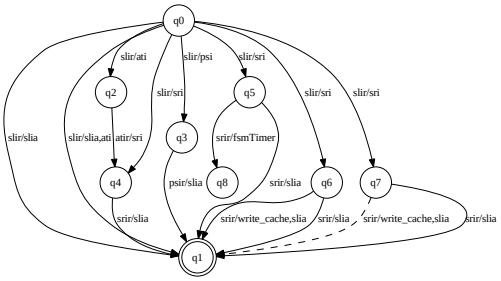


Figure 5. The inferred Mealy machine, states $q0 - q8$.

Most of the transitions of the Mealy machine output the error symbol (representing that the corresponding input symbol is "illegal"). Before generating a symbolic representation using our tool, we removed these, since we are interested in being equivalent with respect to the legal input strings. The structure of control locations and edges generated by our transformation is shown in Figure 6, where we show the edges in the set $Edges$ of Algorithm 1. We used the same criterion for forming control locations as used in Algorithm 1.

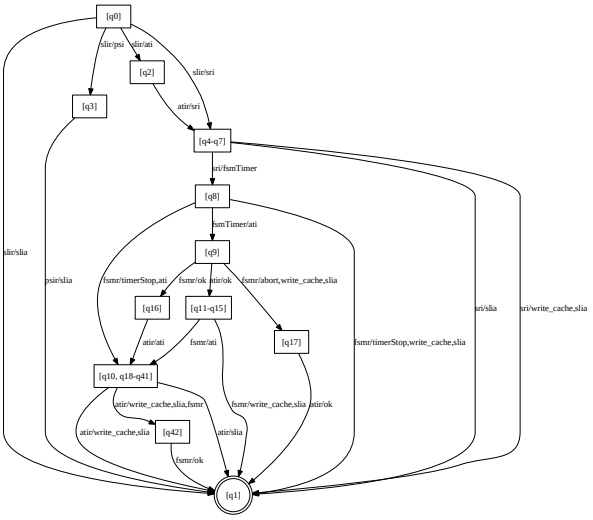


Figure 6. The inferred Mealy machine.

In Figure 6, boxes represent locations. Each location is labeled with the set of states of that "flat" Mealy machine that occured in forming this location; e.g., the location with label [$q4$-$q7$] contains extended states with states $q4$, $q5$, $q6$, $q7$ of the "flat" Mealy machine, since, as can be seen in Figure 5, they can all be reached from state $q0$ by the pair `slir/sri` of input and output action types

A. Evaluation

To evaluate our transformation we compare

- *coverage*: how many of the control locations and edges in the executable specification are captured in our symbolic representation,
- *similarity*: of the locations in our symbolic representation and of those in the executable specification,
- *readability* of the action expressions of our symbolic representation, as compared with those in the executable specification.

Coverage: 12 of the 13 control locations of the executable specification have been reached in our symbolic representation. The control location LAST_NETPARAM in Figure 2 could not be reached, since we had reduced the range of parameter `status`, as described in the beginning of Section V-B.

The executable specification has 60 edges. The described reduction of the parameter range of `status` causes 20 of these to become unreachable. Of the remaining 40, our model captured 26. The missing 14 edges are all missing for the same reason, namely that LearnLib incorrectly merged two particular states in the flat Mealy machine. Let us explain how. The state $q5$ in Figure 5 is reached by (among others) `slir` messages with both the values `psi` and `ati` of one particular parameter. The effect of these parameters is not externally observable immediately in the behavior of the SUT, but shows up only two transitions later. However, the L^* algorithm sees that the message following the `slir` message with parameter value `psi` triggers the same output as the message following the `slir` message with parameter value `ati`. L^* then assumes that all the replies to all following messages does not depend on whether the parameter value `psi` or `ati` was supplied with the `slir` message. It then continues to explore continued behavior of the SUT only for longer input strings that start with the `ati` value. This problem can be avoided by having more powerful test suites in equivalence oracles. Our equivalence test used only 1000 randomly chosen input strings; we conjecture that a larger equivalence test would discover the differences between the two parameter values.

Similarity: Table I shows how the locations of our symbolic representation correspond to those of the executable specification The locations NOT_YET_UPDATED and WAIT_FSM_RESP are not distinguished in our symbolic representation, since they are reached by the same sequence of input-output action types. Also, locations [q2] and [q3] correspond to location LAST_POS, which can be reached by two different pairs of input-output action types.

Readability: Since we cannot compare the two models in their entirety, we have chosen to compare two typical action expressions, representing the same behavior to each other. Figure 8 shows a part of our generated action expression from the initial location when a message of form `Slir` with formal parameters (`msis`, `loct`, `maxage`, `netp`, `epsi`) is received, and Figure 7 shows the corresponding part of the executable spec-

Location	Control State
[$q0$]	IDLE
[$q1$]	DONE
[$q2$]	LAST_POS
[$q3$]	LAST_POS
[$q4$-$q7$]	ACCESS_NETPARAM
[$q8$]	FORCE_UPDATE
[$q9$]	TIMER_TRIGGERED
[$q11$-$q15$]	NOT_YET_UPDATED, WAIT_FSM_RESP
[$q16$]	MAYBE_UPDATED
[$q17$]	WAIT_POS_RESP
[$q10$,$q18$-$q41$]	UPDATED
[$q42$]	TERMINATE_MMS

Table I
CORRESPONDENCE BETWEEN LOCATIONS IN OUR SYMBOLIC REPRESENTATION AND IN THE EXECUTABLE SPECIFICATION

```
    in location IDLE
1)
2)  when Slir(msis, loct, netp, epsi, frc, lra)
3)    if(epsi)
4)      if(frc)or((!frc)and((lra)and(loct==last)))
5)        case (netp) of
6)          false → output Psi(netpar);nextloc LAST_POS;
7)          true → output Sri(msis); nextloc ACCESS_NETPARAM;
8)        endcase
9)      else if((!frc)and(!lra)){ output Slia(netp, msis);
                                  nextloc DONE; }
10)     else { output ErrMsg;nextloc ErrLoc;}
11) ...
12) MSIS=msis;LOCT=loct;NETP=netp;EPSI=epsi;FRC=frc; LRA=lra;
13) end
```

Figure 7. Small extract of executable specification

ification. In the figures, the values of input parameters are assigned to state variables, shown by upper-case letters, in line 12. To simplify the comparison between the action expression and executable specification we have replaced the parameter values of output symbols by the parameters' name of received input symbol. For this we carefully matched the values of the parameters in output symbols with the input action type's parameter names and found the corresponding parameter name for each parameter value.

(* the last 7 lines above should be fixed *)

We see that the action expression is more compact in the executable specification. One reason is that it uses complex boolean expressions (e.g., Figure 7 line 4), whereas our representation only uses a simple decision tree structure which tests one parameter or variable at a time. This makes the executable specification smaller than our representation, but sometimes more difficult to understand.

Another difference is that our representation does not explicitly return an error message on illegal input. This allows our action expressions to sometimes omit distinctions. In this example, the `loct` parameter is tested in Figure 7 line 4, but not in Figure 8.

VII. CONCLUSIONS AND FUTURE WORK

We have presented an approach using regular inference to infer models of communication protocol entities. We

```
    in location IDLE
1)
2)  when Slir(msis, loct, netp, epsi, frc, lra)
3)    if(epsi)
4)      case netp of
5)        false →
6)          if(!frc)
7)            if(!lra){output Slia(netp, msis);nextloc DONE;}
8)            else if(lra) { output Psi(netp);nextloc LAST_POS;}
9)          else if(frc) { output Psi(netp);nextloc LAST_POS;}
10)       true →
11)         if(!frc) {
12)           if(!lra) {output Slia(netp, msis);nextloc DONE;}
13)           else if(lra){output Sri(msis);
                           nextloc ACCESS_NETPARAM;}
14)         else if(frc){output Sri(msis);nextloc ACCESS_NETPARAM;}
15)     endcase
16) ...
17) MSIS=msis;LOCT=loct;NETP=netp;EPSI=epsi;FRC=frc;LRA=lra;
18) end
```

Figure 8. Small extract of action expression related to specification part in Figure 7

recognize that communication protocols tend to have their behavior structured into control states. Furthermore, our approach aims to infer a model with locations that are similar to control states in the protocol. Our technique consists of two phases; in the first phase we apply existing regular inference techniques to construct a Mealy machine model of the protocol, and in the second phase we fold this model into a Symbolic Mealy machine.

We have applied our approach to an executable specification of the A-MLC protocol developed by Mobile Arts. We used LearnLib to generate a flat Mealy machine, which was then transformed into a symbolic representation by our implementation, and evaluated the result by comparing it to the original executable specification.

(* say something about the size of the generated model *) The two models had many similarities, but differed in some respects. Our model did not cover all the locations and transitions of the SUT, due to an incorrect merging of two states by the L^* algorithm, which caused a part of the behavior to be unexplored. We conjecture that this problem would go away if we would have used a larger test suite for checking the generated model; at the time of the experiment our time and space resources did not allow this. Our structure of locations was surprisingly similar to that of the manually generated executable specification. Our action expressions has a rather simple form, and thus they become longer than corresponding hand-generated ones. This suggests to look at more advanced ways to generate action expressions in a richer syntax, and to employ code transformations that reduce redundancies.

We would like to apply the approach to other communication protocols in order to further evaluate the approach. This would hopefully also give us more insights of how to create structures in terms of locations.

VIII. Acknowledgments

We are grateful to Harald Raffelt, Bernhard Steffen and Falk Howar for generous support when using the LearnLib tool, and for helpful discussions.

References

[1] M. Broy, B. Jonsson, J.-P. Katoen, M. Leucker, and A. Pretschner, Eds., *Model-Based Testing of Reactive Systems*, ser. LNCS Springer Verlag, 2004, vol. 3472.

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

[3] T. Henzinger, R. Jhala, R. Majumdar, and G. Sutre, "Lazy abstraction," in *Proc. 29^{th} ACM Symp. on Principles of Programming Languages*, 2002, pp. 58–70.

[4] D. Angluin, "Learning regular sets from queries and counterexamples," *Information and Computation*, vol. 75, no. 2, pp. 87–106, 1987.

[5] E. M. Gold, "Language identification in the limit," *Information and Control*, vol. 10, no. 5, pp. 447–474, 1967.

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

[7] R. Rivest and R. Schapire, "Inference of finite automata using homing sequences," *Information and Computation*, vol. 103, pp. 299–347, 1993.

[8] A. Hagerer, H. Hungar, O. Niese, and B. Steffen, "Model generation by moderated regular extrapolation," in *Proc. FASE '02, 5^{th} Int. Conf. on Fundamental Approaches to Software Engineering*, ser. LNCS, vol. 2306. Springer Verlag, 2002, pp. 80–95.

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

[10] K. Li, R. Groz, and M. Shahbaz, "Integration testing of distributed components based on learning parameterized I/O models," in *FORTE*, ser. LNCS, vol. 4229, 2006, pp. 436–450.

[11] R. Groz, K. Li, A. Petrenko, and M. Shahbaz, "Modular system verification by inference, testing and reachability analysis," in *TestCom/FATES*, ser. LNCS, vol. 5047, 2008, pp. 216–233.

[12] D. Peled, M. Y. Vardi, and M. Yannakakis, "Black box checking," in *Formal Methods for Protocol Engineering and Distributed Systems, FORTE/PSTV*, J. Wu, S. T. Chanson, and Q. Gao, Eds. Beijing, China: Kluwer, 1999, pp. 225–240.

[13] A. Groce, D. Peled, and M. Yannakakis, "Adaptive model checking," in *Proc. TACAS '02*, ser. LNCS, vol. 2280. Springer Verlag, 2002, pp. 357–370.

[14] M. Fowler, *UML Distilled – A Bried Guide to the Standard Object Modeling Language*. Addison-Wesley, 2008, 3rd Ed.

[15] J. Cobleigh, D. Giannakopoulou, and C. Pasareanu, "Learning assumptions for compositional verification," in *Proc. TACAS '03*, ser. LNCS, vol. 2619. Springer Verlag, 2003, pp. 331–346.

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

[17] M. Shahbaz, K. Li, and R. Groz, "Learning and integration of parameterized components through testing," in *TestCom/FATES*, ser. LNCS, vol. 4581. Springer, 2007, pp. 319–334.

[18] L. Mariani and M. Pezz, "Dynamic detection of COTS components incompatibility," *IEEE Software*, vol. 24, no. 5, pp. 76–85, September/October 2007.

[19] T. Berg, B. Jonsson, and H. Raffelt, "Regular inference for state machines with parameters," in *FASE*, ser. LNCS, L. Baresi and R. Heckel, Eds., vol. 3922. Springer, 2006, pp. 107–121.

[20] ——, "Regular inference for state machines using domains with equality tests," in *FASE*, ser. LNCS, J. L. Fiadeiro and P. Inverardi, Eds., vol. 4961. Springer, 2008, pp. 317–331.

[21] O. Niese, "An integrated approach to testing complex systems," Dortmund University, Tech. Rep., 2003, doctoral thesis.

[22] T. S. Chow, "Testing software design modeled by finite-state machines," *IEEE Trans. on Software Engineering*, vol. 4, no. 3, pp. 178–187, May 1978, special collection based on COMPSAC.

[23] M. P. Vasilevski, "Failure diagnosis of automata," *Cybernetic*, vol. 9, no. 4, pp. 653–665, 1973.

[24] J. Quinlan, "Induction of decision trees," *Machine Learning*, vol. 1, no. 1, pp. 81–106, March 1986.

[25] T. M. Mitchell, *Machine Learning*. McGraw Hill, 1997.

[26] I. H. Witten and E. Frank, *Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations*. Morgan Kaufmann, 1999.

[27] J. Armstrong, *Programming ERLANG: software for a concurrent world*, ser. Pragmatic programmers. pub-PRAGMATIC-BOOKSHELF:adr: pub-PRAGMATIC-BOOKSHELF, 2007. [Online]. Available: http://www.oreilly.com/catalog/

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

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