Learning of Automata Models Extended with Data

Bengt Jonsson

Learning of Automata Models Extended with Data *

Bengt Jonsson

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

Abstract. One of the challenges in the CONNECT project is to develop techniques for learning models of networked components from exploratory interaction with the component, based on analyzing messages exchanged between the component and its environment. Many approaches to this problem employ regular inference (aka. automata learning) techniques which generate modest-size finite-state models. Most communication with real-life systems involves data values being relevant to the communication context and thus influencing the observable behavior of the communication endpoints. When applying methods from the realm of automata learning, it is desirable to handle such data-occurrences. It is therefore important to extend inference techniques to handle message alphabets and state-spaces with structures containing data parameters, often with large domains. After very briefly mentioning several approaches to the problem, we give a longer account of an approach proposed by Aarts et al, which adapts ideas from of predicate abstraction, successfully used in formal verification. We illustrate the techniques by application to a simple running example, which models a simple booking service.

Acknowledgment This paper builds on joint work with several present and former collaborators, including Fides Aarts, Therese Bohlin, Sofia Cassel, Olga Grinchtein, Falk Howar, Maik Merten, Bernhard Steffen, Johan Uijen, and Frits Vaandrager. Mistakes and inconsistencies are caused by the author.

1 Introduction

Interoperability remains a fundamental challenge when connecting heterogeneous systems which encounter and spontaneously communicate with one another in pervasive computing environments. The CONNECT Integrated Project [22] aims at overcoming the interoperability barrier by 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

* Supported in part by EC Proj. 231167 (CONNECT).

behavior of networked systems, together with (iv) synthesizing new interaction behaviors out of the ones exhibited by the systems to be made interoperable.

One of the challenges in the CONNECT project is to develop techniques for learning models of exploratory interaction with the component, based on analyzing messages exchanged between the component and its environment. Generation of models by exploratory interaction can be useful also in other contexts. A large source of application might be found in model-based verification and validation, including model checking and model-based testing [7]. Such techniques have witnessed drastic advances in the last decades, and are being applied to verification and valiation of communication protocols, hardware systems, embedded controllers, etc., also in industrial settings (e.g., [20]). They require 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 useful also, 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.

The construction of models from observations of component behavior can be performed using automata learning (aka. regular inference) techniques [4, 11, 13, 23, 30, 33]. This class of techniques is now receiving increasing attention in the testing and verification community, e.g., for regression testing of telecommunication systems [18, 21], for integration testing [17, 24], security protocol testing [32], and for combining conformance testing and model checking [29, 16]. One of the most used algorithms for regular inference, L^*, is thoroughly explained in the Chapter *Introduction to Active Automata Learning from a Practical Perspective* by Steffen, Howar, and Merten. This algorithm poses a sequence of *membership queries*, each of which observes the component's output in response to a certain input word, and produces a minimal deterministic finite-state machine which conforms to the observations. If the sequence of membership queries is sufficiently large, the produced machine will be a model of the observed component.

Since regular inference techniques are designed for finite-state models, most previous applications to model generation have been limited to generating control flow skeletons, suppressing data which appear, e.g., as parameters of messages. However, data parameters have a significant impact on control flow and behavior in typical networked components and protocol entities. they can be sequence numbers, configuration parameters, agent and session identifiers, etc.; a model of a networked service is considerabely less informative if information about exchanged data is suppressed. It is therefore important to extend inference techniques to handle message alphabets and state-spaces with structures containing data parameters with large domains.

In this chapter, we will consider the problem of extending learning to automata with data, by presenting a particular approach, introduced in the work by Aarts, Jonsson, Uijen, and Vaandrager [1, 2]. We first define a model for

symbolic representation of protocols. Thereafter, we present a technique for using the L^* algorithm, designed for inference of finite-state Mealy machines, to infer also symbolically defined protocol models. The technique is inspired by predicate abstraction [26, 9], 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, where an abstraction cannot be defined based on the source code or model of a component, since it is not accessible. Instead, we must construct an externally supplied abstraction, which translates between a large message alphabet of the component to be modeled and a small finite alphabet of the regular inference algorithm. Via regular inference, a finite-state model of the abstracted interface is inferred. The abstraction can then be reversed to generate a faithful model of the component.

The presented approach was used to learn models of reduced versions of the SIP and TCP protocols, in [1], and also to learn a model of the new generation of biometric passports in [2]. A We will also describe how to construct a suitable abstraction, utilizing pre-existing knowledge about which operators are sufficient to express guards and operations on data in a faithful model of the component.

On 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 [10], for regression testing to create a specification and test suite [18, 21], to perform model checking without access to source code or formal models [16, 29], for program analysis [3], and for formal specification and verification [10].

In several approaches, the challenge of including data parameters of message have been addressed. In the work of Shu and Lee [32], parameters are essentially suppressed in order to obtain a finite subset of input symbols when learning the behavior of security protocol implementations. This subset can be extended in response to new information obtained in counterexamples. Groz, Li, and Shahbaz [24, 31, 17] extend regular inference to Mealy machines with data values, for use in integration testing. In their work, they select a finite set of representative data values to be supplied together with the input to ta component.

An influential approach to learning properties of data in programs is represented by the Daikon system [12]. Its basic technique is to observe executions of a component, and extract invariants over program variables that are observed to hold. The invariants can be chosen from a predefined collection. The Daikon system does not immediately consider to extract control structures of components. There are several approaches that combine regular inference for learning control structures, and the Daikon tool (or similar) for inferring constraints on data parameters. One of the questions to be solved in such a combination is how to correlate the two types of models.

Lorenzoli, Mariani, and Pezzé infer models of software components that consider both sequence of method invocations and their associated data parameters [27, 28]. They use a passive learning approach where a finite control structure that captures possible sequences of method invocations is infer by an extension of the k-tails algorithm (a passive learning algorithm), and using Daikon [8] to

infer guards and relations on method parameters. This allows to infer constraints on data parameters that are exchanged after specific sequences of method invocations, but not to analyze the influence of data parameter on subsequent control behavior. The same basic combination is also employed by Lo and Maoz [25], which infer a more refined view on constraints over data parameters, in that different constraints are generated for different scenarios, if a need for this is detected.

In previous work, we have considered extensions of regular inference to handle data parameters. In [5], we show how guards on boolean parameters can be refined lazily. This technique for maintaining guards have inspired the more general notion of abstractions on input symbols presented in this chapter. We have also proposed techniques to handle infinite-state systems, in which parameters of messages and state variables are from an unbounded domain, e.g., for identifiers [6], and timers [15, 14]. These extensions are specialized towards a particular data domain, and their worst-case complexities do not immediately suggest an efficient implementation.

Organization. In the next section, we first introduce a simple running example that will serve to illustrate the techniques presented in this tutorial. Thereafter, we introduce Mealy machines, and our symbolic extension of Mealy machines, that include data. The technique of using abstraction to adapt finite-state learning algorithms to symbolically defined Mealy machines is presented in Section 5. This techniques requires an abstraction which in general must be constructed manually. A technique for systematic construction of such abstractions is presented in 7. We illustrate the application of this technique to the running example in Section 6.

2 A Running Example

Let us introduce a small example to illustrate the techniques that will be introduced in later sections. Imagine a service for booking seats in a concert or similar event. A user of this service has to provide his credentials and can then browse through a list of seats. From the list of seats, a single seat can be booked, which will be confirmed in a corresponding receipt.

Imagine further that an *a priori* interface description of the service is provided, specifying a specific set of messages that are understood by the service, containing

- openSession with two parameters, a user name and a password, supplies credentials, and if they are accepted the service provides a session identifier in response,
- getSeats with a session identifier as parameter, asks for a list of avaible seats that can be booked,
- getSeat with a session identifier and a seat as parameters, asks to book a specific seat, and if accepted, the service will confirm by a positive reply.

The exchange of interface primitives during a typical session can be informally depicted in sequence chart in Figure 1. In the following sections, we will con-

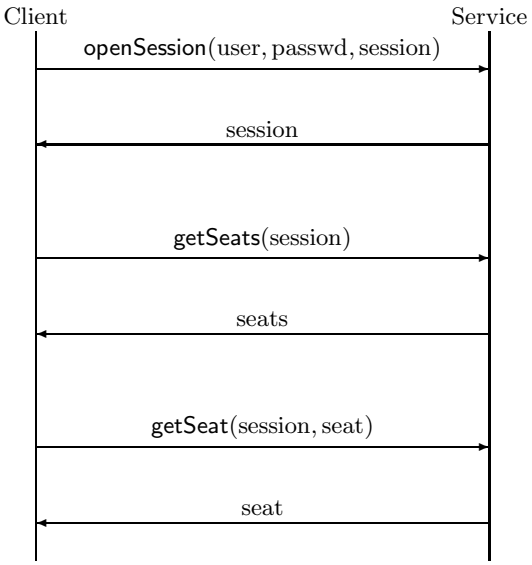


Fig. 1. Message Flow in the Running Example

sider how active automata learning, which is able to generate finite-state Mealy machines from queries, can be used to generate a model of the booking service.

3 Mealy Machines

Throughout the presentation, we will use *Mealy machines* to model the behavior of communication protocol entities, networked services, etc.

Definition 1. 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*. □

The sets of states and symbols can be finite or infinite: if they are both finite we say that the Mealy machine is *finite-state*. Elements of Σ_I^* are called *input words*, and elements of Σ_O^* are called *output words*.

Intuitively, a Mealy machine behaves as follows. At any point in time, the machine is in some state $q \in Q$. When supplied with an input symbol $a \in \Sigma_I$, it responds by producing an output symbol $\lambda(q, a)$ and transforms itself to a 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 can depict Mealy machines as directed edge-labeled graphs, where Q is the set of vertices. The outgoing edges from a state $q \in S$ lead to $\delta(q, a)$ for all $a \in \Sigma_I$, and they are labeled "a/b", where a is the input symbol and b is the output symbol $\lambda(q, a)$. As an example, Figure 2 shows a Mealy machine that receives a sequence of symbols of form a or b. Whenever an a-symbol is received, it outputs the number of received a-symbols modulo 2, and whenever a b-symbol is received, it outputs the number of received a-symbols modulo 4. The initial state is q_0.

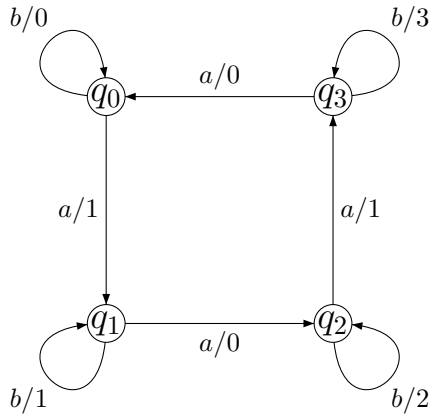


Fig. 2. A Mealy machine $\langle \Sigma_I, \Sigma_O, Q, q_0, \delta, \lambda \rangle$ with states $Q = \{q_1, q_2, q_3, q_4\}$, input alphabet $\Sigma_I = \{a, b\}$, and output alphabet $\Sigma_O = \{0, 1, 2, 3\}$. For instance, applying a starting in q_0 produces output $\lambda(q_0, a) = 1$ and moves to next state $\delta(q_0, a) = q_1$.

Applying a word $a_1 a_2 \cdots a_k \in \Sigma_I^*$ of input symbols starting in a state q_0 results in the sequence of states $q_0, q_1, \ldots, q_k$ with $q_j = \delta(q_{j-1}, a_j)$ for $j = 1, \ldots, k$. We extend the transition function to $\delta(q_0, a_1 a_2 \cdots a_k) \stackrel{\text{def}}{=} q_k$ and the output function to $\lambda(q_0, a_1 a_2 \cdots a_k) \stackrel{\text{def}}{=} \lambda(q_0, a_1)\lambda(q_1, a_2) \cdots \lambda(q_{k-1}, a_k)$, i.e., the concatenation of all outputs. We define $\lambda_{\mathcal{M}}(u) = \lambda(q_0, u)$ for $u \in \Sigma_I^*$. Two Mealy machines $\mathcal{M}$ and $\mathcal{M}'$ with the same set of input symbols are *equivalent* if $\lambda_{\mathcal{M}}(u) = \lambda_{\mathcal{M}'}(u)$ for all input words u.

Mealy machines are *completely specified*, meaning that at every state there is a next state for every input (δ and λ are total). They are also *deterministic*, because only one next state is possible.

4 Symbolic Mealy Machines

Finite-state Mealy machines, introduced in the previous section, cannot represent all aspects of the behavior of protocols and networked components, where the interplay between control and data is significant. Typical such models have an infinite number of states and infinite communication alphabets that span domains of data values that are very large or infinite. Typical examples of such data domains are integers to represent sequence numbers, session identifiers, etc., strings to represent exchanged data, etc.

In this section, we introduce *Symbolic Mealy Machines*. They can be seen as a symbolic representation of large or infinite-state Mealy machines, in that input and output symbols have parameters which are data values, e.g., to represent messages in a typical communication protocol. Often, data parameters are from rather large (in practice "infinite") domains, and on which rather simple operations and tests are applied, e.g., equality tests between elements of the same domain, or a check whether an element is a member of some set. It seems reasonable to be able to extend automata learning to such models, in analogy with the way automated verification techniques have been extended from finite-state models to extensions that cover, e.g., clocks as in timed automata.

Data Values We first consider the data values that occur as parameters of input and output symbols and stored in state variables that are part of the representation of the internal state of a Mealy machine. We will use d, d_1, d_2, etc. to range over data values. To describe the data values that are relevant for a symbolic Mealy machine, we assume a finite set of *domains*, each of which is a (finite or infinite) set of data values. We also assume a finite set of *functions* and a finite set of *predicates*. Each function f has an *arity*, denoted $\mathcal{D}_1 \times \cdots \times \mathcal{D}_n \mapsto \mathcal{D}$, where $\mathcal{D}_1, \ldots, \mathcal{D}_n$ and $\mathcal{D}$ are domains, meaning that the arguments to f must be an n-tuple of data values $d_1, \ldots, d_n$, where $d_i \in \mathcal{D}_i$ for $i = 1, \ldots, n$, and then $f(d_1, \ldots, d_n)$ is an element in $\mathcal{D}$. We write $f : \mathcal{D}_1 \times \cdots \times \mathcal{D}_n \mapsto \mathcal{D}$ to denote that f has arity $\mathcal{D}_1 \times \cdots \times \mathcal{D}_n \mapsto \mathcal{D}$. A predicate r has an arity, denoted $\mathcal{D}_1 \times \cdots \times \mathcal{D}_n$, meaning that it can be thought of as a function from $\mathcal{D}_1 \times \cdots \times \mathcal{D}_n$ to boolean values.

Input and Output Symbols Input and output symbols will be represented using finite sets I and O of (input and output) *ations*. Each ation α has a certain *arity*, which is a tuple of *domains* $\mathcal{D}_{\alpha,1}, \ldots, \mathcal{D}_{\alpha,n}$ (where n depends on α). 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.

Example For the service introduce in Section 2, we use the following domains to represent the data values that occur in input and output symbols.

- STRING contains data values for user names and passwords.
- SESSION contains identifiers of sessions: it could be, e.g., the set of natural numbers.

– SEAT contains the possible seats (e.g., represented by seat numbers) that are available in the event, and
– SEATS contains sets of seats in SEAT.

We use the following predicates.

– $\in$: SEAT $\times$ SEATS is the test for membership, and
– has_passwd : STRING $\times$ STRING tests for valid combinations of usernames and passwords.

In addition, we include the equality predicate $=$ on the domains SESSION and SEAT.

The set of input ations with corresponding arities is described in the following table.q

Input ation	arity
openSession :	STRING, STRING, SESSION
getSeats :	SESSION
getSeat :	SESSION, SEAT

To model the response from the service, we could use one output ation for each kind of response, e.g., an ation returnSession with arity SESSION for replies to input symbols of form openSession(u, p, s). To save space, we will simply just let the reply be modeled by a data element from the appropriate domain. □

Symbolic Mealy Machines We can now define symbolic Mealy machines. We assume a set of domains, functions, and predicates, as described in the previous paragraphs, which will be used to form expressions denoting data values, and boolean expressions to denote tests on data values. We assume that expressions always follow the restrictions of the relevant arities.

We assume a set of *formal parameters*, ranged over by $p_1, p_2, \ldots$, to be used as placeholders for parameters of symbols in symbolic transitions. We also assume a set of *state variables*, each with a domain of possible values, and a unique initial value.

Definition 2. A *Symbolic Mealy machine* (SMM for short) is a tuple $\mathcal{SM} = \langle I, O, L, l_0, X, \longrightarrow \rangle$, where

– I and O are disjoint finite sets of actions (*input ations* and *output ations*),
– L is a finite set of *locations*,
– $l_0 \in L$ is the *initial location*,
– X is a finite set of state variables; each state variable x has a domain $\mathcal{D}_x$ of possible values, and a unique initial value, and
– $\longrightarrow$ is a finite set of *symbolic transitions*, each of form

$$l \xrightarrow{\alpha(p_1,\ldots,p_n)\ \textbf{when}\ g\ /\ x_1,\ldots,x_k := e_1,\ldots,e_k\ ;\ \beta(e^{out}{}_1,\ldots,e^{out}{}_m)} l'$$

in which
  • l and l' are locations,
  • $\alpha \in I$ and $\beta \in O$ are input and output actions,

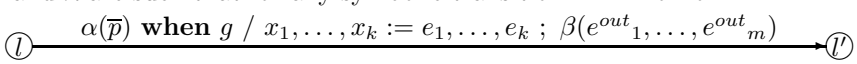
- $p_1, \ldots, p_n$ are distinct formal parameters,
- $x_1, \ldots, x_k$ are distinct state variables in X,
- g (the *guard*) is a boolean expression over the formal parameters $p_1, \ldots, p_n$ and the state variables in X, and
- $e_1, \ldots, e_k$ and ${e^{out}}_1, \ldots, {e^{out}}_m$ are tuples of expressions over $p_1, \ldots, p_n$ and X. We assume that the arities of α and β and the domains of $x_1, \ldots, x_k$ are respected. □

Intuitively, a symbolic transition of the above form denotes steps of the Mealy machine in which some input symbol of form $\alpha(d_1, \ldots, d_n)$ is received, whereby the formal parameters $p_1, \ldots, p_n$ are bound to the received data values $d_1, \ldots, d_n$; in case the guard g is evaluated to true, the state variables among $x_1, \ldots, x_k$ are assigned new values by the assignment $x_1, \ldots, x_k := e_1, \ldots, e_k$, and an output symbol, obtained by evaluating $\beta({e^{out}}_1, \ldots, {e^{out}}_m)$, is generated. In case the guard g is evaluted to false, then the symbolic transition does not denote any step.

Semantics of SMM We can give a precise meaning to an SMM by letting it denote a Mealy machine with possibly infinite sets of input and output symbols and states. Such a Mealy machine can be defined as follows. Assume an SMM $\mathcal{SM}$ defined as the tuple $\langle I, O, L, l_0, X, \longrightarrow \rangle$. A *valuation* is an assignment σ which maps each location variable x in X to a data value in $\mathcal{D}_x$. Valuations are extended to expressions in the natural way: for instance, if $\sigma(x_3) = 8$, then $\sigma(2 * x_3 + 4) = 20$. We let σ_0 denote the valuation which maps each variable to its initial value. In the following, we will use $\overline{p}$ for $p_1, \ldots, p_n$ and $\overline{d}$ for $d_1, \ldots, d_n$.

Definition 3. We define a SMM $\mathcal{SM} = \langle I, O, L, l_0, X, \longrightarrow \rangle$ as denoting a (typically infinite-state) Mealy machine $\mathcal{M}_{\mathcal{SM}} = \langle \Sigma_I, \Sigma_O, Q, q_0, \delta, \lambda \rangle$, where

- Σ_I is obtained from I as described when introducing actions, and similarly for Σ_O,
- Q is the set of pairs $\langle l, \sigma \rangle$ consisting of a location $l \in L$ and a valuation σ,
- q_0 is the pair $\langle l_0, \sigma_0 \rangle$, and
- δ and λ are such that for any symbolic transition in $\longrightarrow$ of form



for any valuation σ and data values $\overline{d}$ such that $\sigma(g[\overline{d}/\overline{p}])$ is true (i.e., under the valuation σ, the guard g is evaluated to true when the formal parameters $\overline{p}$ are replaced by the received data values $\overline{d}$), it holds that

- $\delta(\langle l, \sigma \rangle, \alpha(\overline{d})) = \langle l', \sigma' \rangle$, where σ' is the valuation such that
  - $\sigma'(x_i) = \sigma(e_i[\overline{d}/\overline{p}])$ for $1 \leq i \leq k$, and
  - $\sigma'(x) = \sigma(x)$ if x is not among $x_1, \ldots, x_k$,
- $\lambda(\langle l, \sigma \rangle, \alpha(\overline{d})) = \beta(\sigma'(e_1^{out}[\overline{d}/\overline{p}]), \ldots, \sigma'(e_m^{out}[\overline{d}/\overline{p}]))$. □

Here the four last lines say that the state is updated to a new pair $\langle l', \sigma' \rangle$, where l' is the target location of the symbolic transition and σ' is obtained by performing the multiple assignment $x_1, \ldots, x_k := e_1, \ldots, e_k$ simultaneously to all variables

that are among $x_1, \ldots, x_k$, and that an output symbol, obtained by evaluating the expression $\beta(e^{out}{}_1, \ldots, e^{out}{}_m)$ in σ, is generated.

Having defined the meaning of an SMM through translation to an ordinary Mealy machine, we can inherit some definitions. We use $\lambda_{\mathcal{SM}}$ to denote $\lambda_{\mathcal{M}_{\mathcal{SM}}}$, and say that $\mathcal{SM}$ and $\mathcal{SM}'$ are equivalent if $\lambda_{\mathcal{SM}}(u) = \lambda_{\mathcal{SM}'}(u)$ for all input words u. We can similarly say that an SMM is equivalent to a Mealy machine.

Symbolic Mealy machines are required to be deterministic, just like ordinary Mealy machines. We say that $\mathcal{SM}$ is deterministic if $\mathcal{M}_{\mathcal{SM}}$ is deterministic: a sufficient condition under which $\mathcal{M}_{\mathcal{SM}}$, is deterministic is that for each input action $\alpha \in I$, each location $l \in L$, and each valuation σ, the set $\longrightarrow$ contains exactly one symbolic transition such that $\sigma(g[\overline{d}/\overline{p}])$ is true.

Example 1. Model a simple booking system

5 Inference Using Abstraction

Let us consider the problem of extending finite-state automata learning (as realized. e.g., by th L^* alglorithm) to the learning of infinite state automata, as represented by Symbolic Mealy machines. More precisely, given a *SUT*, whose behavior can be modeled as an SMM $\mathcal{SM}$, we should describe how a component, called the *Learner*, which communicates with the *SUT*, can infer an SMM equivalent to $\mathcal{SM}$ by query learning. In this setup, the *Learner* initially knows the static interface of $\mathcal{SM}$, i.e., the sets I and O of input and output actions together with their arities. It may then ask a sequence of *membership queries*; each one supplying a chosen input word $u \in (\Sigma_I)^*$ and observing the response $\lambda_{\mathcal{SM}}(u)$. After a "sufficient" number of membership membership queries the *Learner* can build a "stable" hypothesis $\mathcal{H}$ from the obtained information. The hypothesis $\mathcal{H}$ should of course agree with $\mathcal{SM}$ on the performed membership queries (i.e., $\lambda_{\mathcal{SM}}(u) = \lambda_{\mathcal{H}}(u)$ whenever u was supplied in a membership query), but must make suitable generalizations for other input words.

The L^* algorithm is designed for finite-state Mealy machines and cannot construct infinite-state models. In order to use it for inferring models of large or infinite-state SMMs, we must somehow transform the behavior of an SMM so that it becomes the behavior of some finite-state Mealy machine. In this section, we present an approach, which has been elaborated in the work by Aarts, Jonsson, Uijen, and Vaandrager [1, 2]. The approach adapts ideas from predicate abstraction [26, 9], which has been successful for extending finite-state model checking to large and infinite state spaces.

In order to introduce our ideas, consider an SMM $\mathcal{SM} = \langle I, O, L, l_0, X, \longrightarrow \rangle$ for which the sets Σ_I and Σ_O of input and output symbols, and the set of valuations of X may be large or even infinite. To apply regular inference to $\mathcal{SM}$, we here propose to define an abstraction from Σ_I and Σ_O to (small) finite sets of *abstract* input and output symbols. The overall idea can be schematically depicted as in Figure 3. The abstraction $\mathcal{A}$ interacts with the *SUT* using the alphabets Σ_I and Σ_O; it interacts with the *Learner* using finite alphabets of

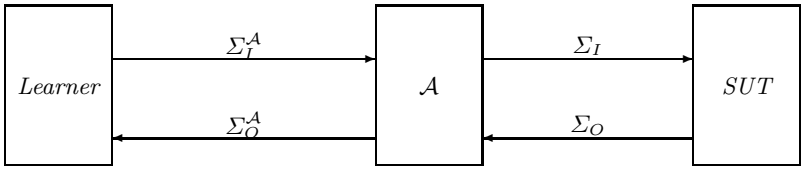


Fig. 3. Introducing an abstraction $\mathcal{A}$ between the SUT and the *Learner*

symbols $\Sigma_I^{\mathcal{A}}$ and $\Sigma_O^{\mathcal{A}}$. Thus, $\mathcal{A}$ can transform sequences of symbols in Σ_I or Σ_O to sequences of abstract symbols in $\Sigma_I^{\mathcal{A}}$ or $\Sigma_O^{\mathcal{A}}$. If $\mathcal{A}$ is suitable defined, it can also the (possibly) infinite-state behavior of SUT into a behavior which can be represented by a finite-state Mealy machine. The *Learner* can then use standard techniques to learn this Mealy machine. Having done this, we can finally "reverse" the effect of the abstraction $\mathcal{A}$ to obtain an SMM which is equivalent to the original SUT.

As a concrete example, in the SMM in our running example, symbols of form getSeats(s) in which the parameter s belongs to the large domain of session identifiers, can be abstracted to symbols of form getSeats(S), where S is a value from a small domain. A natural choice for such a small domain could be the set $\{CUR, BAD\}$, where the value CUR denotes that s is the "current" session identifier (supplied in the relevant openSession interaction), and the value BAD denotes that s has some other value. Thus, the abstraction of a symbol, such as getSeats(s), in general depends on the previous history of symbols. In model checking using abstraction [26, 9], this dependency is taken into account by letting the abstraction depend on internal state variables. For instance, the SUT may have a state variable to remember the "current" session identifier; a predicate abstraction will then only represent whether this internal state variable is equal to the session identifier that is received in the input symbol that is being processed. However, automata learning is performed in a black-box setting where the state variables of the SMM are not accessible. Therefore, these state variables must be recreated in the the abstraction, and be updated to record relevant history information. In our example, the recreated state variables can be *cur_session* and *cur_seats*, where *cur_session* is assigned the session identifier supplied to the SUT in the relevant openSession interaction, and *cur_seats* is returned by the SUT in response to the relevant getSeats interaction. Typically, these additional state variables must be defined by a user who has some insight into the functioning of the SUT. In general, this is a nontrivial task, but in Section 7 we discuss approaches for systematically constructing abstractions for situations where the operation on data is not overly complex.

We can define an abstraction formally as follows.

Definition 4. Let I and O be disjoint finite sets of (input and output) actions. An $\langle I, O\rangle$-*abstraction* is a tuple $\mathcal{A} = \langle \Sigma_I^{\mathcal{A}}, \Sigma_O^{\mathcal{A}}, R, r_0, abstr_I, abstr_O, \delta^{\mathcal{R}}\rangle$, where

- $\Sigma_I^{\mathcal{A}}$ and $\Sigma_O^{\mathcal{A}}$ are finite sets of *abstract* input and output symbols,

- R is a (possibly infinite) set of *local states*,
- $r_0 \in R$ is an *initial local state*,
- $abstr_I : R \times \Sigma_I \mapsto \Sigma_I^{\mathcal{A}}$ maps input symbols of the SUT to abstract input symbols,
- $abstr_O : R \times \Sigma_O \mapsto \Sigma_O^{\mathcal{A}}$ maps output symbols of the SUT to abstract output symbols, and
- $\delta^{\mathcal{R}} : R \times (\Sigma_I \cup \Sigma_O) \mapsto R$ updates the local state when a new input or output symbol occurs. □

Intuitively, an abstraction $\mathcal{A}$ maps input and output symbols of the SUT to abstract input and output symbols, and updates its local state immediately after the occurrence of each symbol.

Let us, as was done for Mealy machines, extend the definitions of the abstraction functions to sequences of input and output symbols. We will use u to range over sequences of input symbols, v to range over sequences of output symbols, and w to range over sequences of pairs (of form a/b) of input and output symbols. Since a Mealy machine interacts with both an input symbol and an output symbol at each transition, we extend the definitions of $\delta^{\mathcal{R}}$, $abstr_I$, and $abstr_O$ by defining:

$$\begin{aligned} \delta^{\mathcal{R}}(r, a/b) &= \delta^{\mathcal{R}}(\delta^{\mathcal{R}}(r, a), b) \\ abstr(r, a/b) &= abstr_I(r, a)/abstr_O(\delta^{\mathcal{R}}(r, a), b) \end{aligned}$$

where a/b is a pair of input and output symbol, and $abstr$ maps pairs of input and output symbols to corresponding abstract ones. In the last formula, the abstraction of the input symbol a is performed in the local state r, and the abstraction of the output symbol b is performed wrp. to the local state $\delta^{\mathcal{R}}(r, a)$ reached after having processed the input symbol a.

We thereafter extend $\delta^{\mathcal{R}}$ to sequences of pairs of input and output symbols, by

$$\delta^{\mathcal{R}}(r, \varepsilon) = r \qquad \delta^{\mathcal{R}}(r, w\ a/b) = \delta^{\mathcal{R}}(\delta^{\mathcal{R}}(r, w), a/b)$$

We can similarly extend the mapping $abstr$ from pairs of input and output symbols to sequences of such pairs.

$$\begin{aligned} abstr(r, \varepsilon) &= \varepsilon \\ abstr(r, w\ a/b) &= abstr(r, w)\ abstr(\delta^{\mathcal{R}}(r, w), a/b) \quad , \end{aligned}$$

In particular, $abstr(r_0, w)$ is the abstraction of an arbitrary sequence w of input-output pairs.

In a concrete setup for learning a model of the SUT, we envisage that the abstraction is performed by introducing a *Mapper* module between the *Learner* and the SUT, which carries out the transformations of the abstraction. The *Learner* can then interact with the combination of the *Mapper* and the SUT, using the finite sets $\Sigma_I^{\mathcal{A}}$ and $\Sigma_O^{\mathcal{A}}$, whereas the *Mapper* and the SUT interact using the alphabets Σ_I and Σ_O. The *Mapper* maintains the local state r of the abstraction. Note that the *Mapper* must transform between original and abstract symbols in two different directions, depending on whether the symbol

is an input or output symbol. Each abstract input symbol $a^{\mathcal{A}}$ supplied by the *Learner* is translated by the *Mapper* to a concrete input symbol a such that $a^{\mathcal{A}} = abstr_I(r, a)$, and sent to the *SUT*, while also updating the local state r to $\delta^{\mathcal{R}}(r, a)$. The corresponding reply b by *SUT* is translated to the abstract symbol $abstr_O(\delta^{\mathcal{R}}(r, a), b)$ and sent back to the *Learner*. Finally the local state r is updated to $\delta^{\mathcal{R}}(r, a/b)$.

An example of a possible round of exchanged symbols is depicted in Figure 4. In this round, the abstract symbol openSession(USR, OK) is received by the *Mapper*. Here the combination USR, OK represents a valid combination of user and password. The *Mapper* chooses appropriate concrete data values as parameters, including to choose a session identifier to create an input symbol for the *SUT*. It also stores the chosen session identifier into a local variable. The *SUT* recognizes the input symbol as a valid start of a session, and so acknowledges this by returning the provided session identifier. The *Mapper* compares the session identifier returned with its stored local variable containing the session identifier in the preceding input symbol, finds out that they are equal, and therefore transforms 42 into the abstract symbol *CUR*, representing "current session identifier".

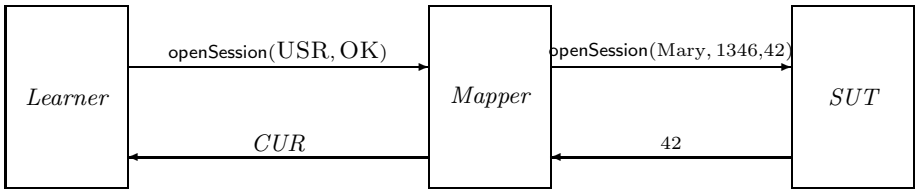


Fig. 4. Introduction of *Mapper* module

Example Let us suggest an abstraction that could be applied in order to learn a model of the seat booking service. In Section 4, we already described the domains and predicates for modeling data parameters. Let us first suggest a representation of the local state of the abstraction. This can be represented by two state variables:

- *cur_session* which stores the value of the "current" session, and and
- *cur_seats* which stores the set of seats that has been proposed by the service.

The initial values of both variables is $\perp$ (undefined). Thus, a state of the abstraction is a valuation ρ which maps *cur_session* and *cur_seats* to values. Initially, ρ maps these variables to the undefined value.

Let us then define the set of abstract input symbols and the state-dependent mapping $abstr_I$ from input symbols of the service to abstract input symbols. The abstraction of an input symbol depends on whether certain guards, that

may be evaluated when such an input symbol is received, hold. For each combination of input symbol and applicable guard, we create a suitable abstract input symbol. In Table 1, we show the different combinations of input symbols and guards, and the corresponding abstract input symbols. For instance, if

Table 1. Mapping from combinations of input symbols and guards to abstract input symbols

input	**guard**	**abstract symbol**
openSession(u, p, s)	has_passwd(u, p)	openSession(USR, OK)
	$\neg$has_passwd(u, p)	openSession(USR, NOK)
getSeats(s)	$s = cur\_session$	getSeats(CUR)
	$s \neq cur\_session$	getSeats(BAD)
getSeat$(s, seat)$	$s = cur\_session \wedge seat \in cur\_seats$	getSeat$(CUR, SEAT)$
	$s = cur\_session \wedge seat \notin cur\_seats$	getSeat(CUR, NO_SEAT)
	$s \neq cur\_session \wedge seat \in cur\_seats$	getSeat$(BAD, SEAT)$
	$s \neq cur\_session \wedge seat \notin cur\_seats$	getSeat(BAD, NO_SEAT)

$\rho(cur\_session) = 42$, and $\rho(cur\_seats) = \{C, D, G\}$, then $abstr_I(\rho, \mathsf{getSeat}(42, F))$ is getSeat(CUR, NO_SEAT).

Let us next define the set of abstract output symbols and the mapping $abstr_O$. As described in Section 4, the set of output symbols correspond to data values in domains SESSION, SEATS, and SEAT. In addition, there is an output symbol *error* which is returned on input that does not make the current session progress. These output symbols are mapped to abstract output symbols as follows.

- Data values in SESSION are mapped to CUR or BAD, depending on whether they are equal to *cur_session* or not.
- Data values in SEATS are mapped to $OFFERED$ or $NOT\_OFFERED$, depending on whether they are equal to the set of seats offered by the service. Here, we have performed a modeling trick in order to be able to model the service as a (deterministic) Mealy machine, in spite of the fact that the set of seats it may return cannot be predicted from the past sequence of input and output symbol. In order not to have to model the return of a set of seats using nondeterminism, we invent a constant, named *offered* (say), which represents the set of seats offered by the service in the session considered.
- Data values in SEAT are mapped to $SEAT$ or NO_SEAT, depending on whether the seat is a member of the set *cur_seats* or not.
- The output symbol *error* is left unchanged by the abstraction.

Let us finally consider how the state of the abstraction is updated on the occurrence of an input or output symbol. This state is unchanged, except for the following cases.

- When an input symbol of form openSession(u, p, s) is received, such that has_passwd(u, p) and when *cur_session* is previously undefined, then *cur_session* is assigned the value s received in the input symbol.

– When an output symbol in domain SEATS is produced in a situation where *cur_seats* is previously undefined, then *cur_seats* is assigned the value of the output symbol. □

In order to better understand what behavior is obtained by wrapping the SUT with the *Mapper*, and which is observed by the *Learner*, let us model the behavior of the combination of the *Mapper* and SUT, which we denote by $\mathcal{A}\langle\langle\mathcal{SM}\rangle\rangle$. Unfortunately, $\mathcal{A}\langle\langle\mathcal{SM}\rangle\rangle$ cannot in general be modeled as a (deterministic) Mealy machine. The reason is that each (abstract) input symbol $a^{\mathcal{A}}$ can be translated by the *Mapper* (in state r) to any input symbol a with $a^{\mathcal{A}} = \mathit{abstr}_I(r, a)$: different choices of a will, in general, cause the SUT to move to different states and subsequently cause different (abstract) output symbols to be generated. In addition, the *Mapper* should have a defined reaction for the case that there is no input symbol a with $a^{\mathcal{A}} = \mathit{abstr}_I(r, a)$. We therefore need to introduce a generalization of Mealy machines that allows nondeterminism, and represent the behavior of $\mathcal{A}\langle\langle\mathcal{SM}\rangle\rangle$ as a nondeterministic Mealy machine. A nondeterministic Mealy machine differs from a Mealy machine as defined in Definition 1 in that the reception of an input symbol can result in several possible combinations of output symbols and next states. For this situation, it is more suitable to use only the notation $q \xrightarrow{a/b} q'$ to denote that when the machine is in state q and receives input symbol a, a possible reaction is to emit output symbol b and move to state q'.

Let $\mathcal{M}_{\mathcal{SM}} = \langle \Sigma_I, \Sigma_O, Q, q_0, \delta, \lambda\rangle$ denote the Mealy machine model of $\mathcal{SM}$, let $(\Sigma_O^{\mathcal{A}})^{\top} = \Sigma_O^{\mathcal{A}} \cup \{\top\}$ and $R^{\top} = R \cup \{r_{\top}\}$, where $\top$ is an output symbol denoting that the provided abstract input symbol cannot be translated by the *Mapper*. Then the behavior of $\mathcal{A}\langle\langle\mathcal{SM}\rangle\rangle$ can be modeled as a nondeterministic Mealy machine in which

– $\Sigma_I^{\mathcal{A}}$ and $(\Sigma_O^{\mathcal{A}})^{\top}$ are the sets of input and output symbols,
– $Q \times R^{\top}$ is the set of states,
– $\langle q_0, r_0\rangle$ is the initial state, and
– whenever $\mathcal{A}\langle\langle\mathcal{SM}\rangle\rangle$ is in state $\langle q, r\rangle$ and receives an abstract input symbol $a^{\mathcal{A}}$, then for any concrete input symbol a such that $a^{\mathcal{A}} = \mathit{abstr}_I(r, a)$
  - the state $\langle q, r\rangle$ can be updated to $\langle \delta(q, a), \delta^{\mathcal{R}}(r, a/b)\rangle$, where $b = \lambda(q, a)$ is the output symbol returned by $\mathcal{SM}$, and
  - the abstract output symbol $\mathit{abstr}_O(\delta^{\mathcal{R}}(r, a), b)$ can be produced.

  We denote this possible symbol exchnage by

$$\langle q, r\rangle \xrightarrow{\mathit{abstr}_I(r,a)/\mathit{abstr}_O(\delta^{\mathcal{R}}(r,a),\lambda(q,a))} \langle \delta(q, a), \delta^{\mathcal{R}}(r, a/\lambda(q, a))\rangle \quad .$$

  For the case where there is no concrete input symbol a such that $a^{\mathcal{A}} = \mathit{abstr}_I(r, a)$, the output symbol $\top$ is produced and the state $\langle q, r\rangle$ is updated to $r_{\top}$, where it remains, i.e.,
  - $\langle q, r\rangle \xrightarrow{a^{\mathcal{A}}/\top} \langle q, r_{\top}\rangle$, and
  - $\langle q, r_{\top}\rangle \xrightarrow{a^{\mathcal{A}}/\top} \langle q, r_{\top}\rangle$ for any $a^{\mathcal{A}} \in \Sigma_I^{\mathcal{A}}$.

Although the behavior of $\mathcal{A}\langle\langle\mathcal{SM}\rangle\rangle$ must, in general, be modeled as a Mealy machine, which is internally nondeterministic, it is still possible that its external behavior will appear to the *Learner* as being deterministic. The *Learner* can only observe the sequences of abstract output symbols that are produced in response to provided input sequences. So, for a sequence $a_1^{\mathcal{A}} \cdots a_n^{\mathcal{A}}$ of abstract input symbols, define $\lambda^{\mathcal{A}\langle\langle\mathcal{SM}\rangle\rangle}(\langle q, r\rangle, a_1^{\mathcal{A}} \cdots a_n^{\mathcal{A}})$ as the set of sequences of output symbols of form $b_1^{\mathcal{A}} \cdots b_n^{\mathcal{A}}$ such that for some sequence of states $\langle q_1, r_1\rangle \cdots \langle q_n, r_n\rangle$ we have

$$\langle q, r\rangle \xrightarrow{a_1^{\mathcal{A}}/b_1^{\mathcal{A}}} \langle q_1, r_1\rangle \xrightarrow{a_2^{\mathcal{A}}/b_2^{\mathcal{A}}} \cdots \xrightarrow{a_n^{\mathcal{A}} co/b_n^{\mathcal{A}}} \langle q_n, r_n\rangle$$

Intuitively, $\lambda^{\mathcal{A}\langle\langle\mathcal{SM}\rangle\rangle}(\langle q, r\rangle, a_1^{\mathcal{A}} \cdots a_n^{\mathcal{A}})$ is the set of abstract output sequences that may be generated by $\mathcal{A}\langle\langle\mathcal{SM}\rangle\rangle$ in response to $a_1^{\mathcal{A}} \cdots a_n^{\mathcal{A}}$, starting from state $\langle q, r\rangle$. In particular, $\lambda^{\mathcal{A}\langle\langle\mathcal{SM}\rangle\rangle}(\langle q_0, r_0\rangle, u)$ is the set of sequences that may result from the input sequence u.

If tha input-output behavior of $\mathcal{A}\langle\langle\mathcal{SM}\rangle\rangle$ is equivalent to that of a deterministic Mealy machine, and if this (deterministic) Mealy machine is finite-state, it will be possible to use L^* for learning a model of its external behavior. Indeed, a well-designed abstraction will preserve the determinism of the *SUT* so that the input-output behavior of $\mathcal{A}\langle\langle\mathcal{SM}\rangle\rangle$ behaves deterministically, i.e., each sequence of supplied (abstract) input symbols uniquely determines the subsequently produced (abstract) output symbol.

We make a formal definition of the condition under which the abstraction will present a deterministic view to the *Learner*.

Definition 5. Let $\mathcal{SM} = \langle I, O, L, l_0, X, \longrightarrow\rangle$ be an SMM, and let $\mathcal{A} = \langle \Sigma_I^{\mathcal{A}}, \Sigma_O^{\mathcal{A}}, R, r_0, abstr_I, abstr_O, \delta^{\mathcal{R}}\rangle$ be an $\langle I, O\rangle$-abstraction. Then $\mathcal{A}$ is *adequate* for $\mathcal{SM}$ if for any sequence $u \in (\Sigma_I^{\mathcal{A}})^*$ of abstract input symbols, the set $\lambda^{\mathcal{A}\langle\langle\mathcal{SM}\rangle\rangle}(\langle q_0, r_0\rangle, u)$ of correspondingly generated output sequences has at most one element. □

Intuitively, adequacy means that $\mathcal{A}\langle\langle\mathcal{SM}\rangle\rangle$ exhibits a deterministic mapping from sequences of abstract input symbols received by the *Mapper* to sequences of abstract output symbols produced by the *Mapper* after abstracting the output of the *SUT*. If $\mathcal{A}$ is adequate for $\mathcal{SM}$, then the *Learner* will perceive that $\mathcal{A}\langle\langle\mathcal{SM}\rangle\rangle$ is equivalent to a (deterministic) Mealy machine (which may or may not be finite-state). For any deterministic mapping from sequences of abstract input symbols to sequences of abstract output symbols, there is a minimal Mealy machine which generates it. This Mealy machine can be defined by a Nerode-like quotient construction, as follows.

Let $Q^{\langle\mathcal{SM},\mathcal{A}\rangle}$ denote the set of states of $\mathcal{A}\langle\langle\mathcal{SM}\rangle\rangle$ that are reachable. More precisely, $Q^{\langle\mathcal{SM},\mathcal{A}\rangle}$ is the smallest subset of $Q \times R$ which includes $\langle q_0, r_0\rangle$ and such that $\langle q, r\rangle \in Q^{\langle\mathcal{SM},\mathcal{A}\rangle}$ implies $\langle \delta(q, a), \delta^{\mathcal{R}}(r, a/\lambda(q, a))\rangle \in Q^{\langle\mathcal{SM},\mathcal{A}\rangle}$ for all $a \in \Sigma_I$). Note that we have excluded states where the *Mapper* has reached $r_\top$.

Define the equivalence $\simeq$ on $Q^{\langle\mathcal{SM},\mathcal{A}\rangle}$ by $\langle q, r\rangle \simeq \langle q', r'\rangle$ if $\lambda^{\mathcal{A}\langle\langle\mathcal{SM}\rangle\rangle}(\langle q, r\rangle, u) = \lambda^{\mathcal{A}\langle\langle\mathcal{SM}\rangle\rangle}(\langle q', r'\rangle, u)$ for any sequence of abstract input symbols $u \in (\Sigma_I^{\mathcal{A}})^*$. Intuitively, two elements of $Q^{\langle\mathcal{SM},\mathcal{A}\rangle}$ are equivalent if they cannot be distinguished by the *Learner*, i.e., any two subsequent sequences of input symbols that are

identified by $abstr_I$ trigger two subsequent output words that are identified by $abstr_O$.

If $\mathcal{A}$ is adequate for $\mathcal{SM}$, then the input-output behavior of $\mathcal{A}\langle\langle\mathcal{SM}\rangle\rangle$ is equal to that of a deterministic Mealy machine $\mathcal{M}^{\mathcal{A}}$ (in the sense that $\lambda^{\mathcal{A}\langle\langle\mathcal{SM}\rangle\rangle}(\langle q_0, r_0\rangle, u) = \{\lambda_{\mathcal{M}^{\mathcal{A}}}(u)\}$ for any sequence $u \in (\Sigma_I^{\mathcal{A}})^*$ of abstract input symbols), which is defined by $\mathcal{M}^{\mathcal{A}} = \langle \Sigma_I^{\mathcal{A}}, \Sigma_O^{\mathcal{A}}, Q^{\mathcal{A}}, q_0^{\mathcal{A}}, \delta^{\mathcal{A}}, \lambda^{\mathcal{A}}\rangle$, where

- $Q^{\mathcal{A}} = Q^{\langle\mathcal{SM},\mathcal{A}\rangle}/\simeq \cup\{q_\top\}$, i.e., the set of states is the set of equivalence classes under $\simeq$ plus an extra state $q_\top$ denoting that an abstract input symbol with no corresponding concrete input symbol has been received,
- $q_0^{\mathcal{A}} = [\langle q_0, r_0\rangle]_\simeq$ is the equivalence class of the initial state of $\mathcal{A}\langle\langle\mathcal{SM}\rangle\rangle$,
- $\delta^{\mathcal{A}}$ and $\lambda^{\mathcal{A}}$ are defined as follows:
  for any $a \in \Sigma_I$ with $abstr_I(r, a) = a^{\mathcal{A}}$ we have
  - $\delta^{\mathcal{A}}([\langle q, r\rangle]_\simeq, a^{\mathcal{A}}) = [\langle \delta(q, a), \delta^{\mathcal{R}}(r, a/\lambda(q, a))\rangle]_\simeq$, and
  - $\lambda^{\mathcal{A}}([\langle q, r\rangle]_\simeq, a^{\mathcal{A}}) = abstr_O(\delta^{\mathcal{R}}(r, a), \lambda(q, a))$,

  for any $a^{\mathcal{A}} \in \Sigma_I^{\mathcal{A}}$ s.t. there is no $a \in \Sigma_I$ with $abstr_I(r, a) = a^{\mathcal{A}}$ we have
  - $\lambda^{\mathcal{A}}([\langle q, r\rangle]_\simeq, a^{\mathcal{A}}) = \top$, and
  - $\delta^{\mathcal{A}}([\langle q, r\rangle]_\simeq, a^{\mathcal{A}}) = q_\top$.

  To complete the definition, we define $\delta^{\mathcal{A}}(q_\top, a^{\mathcal{A}}) = q_\top$ for any $a^{\mathcal{A}} \in \Sigma_I^{\mathcal{A}}$.

The definition of $\simeq$ can be used to show that $\mathcal{M}^{\mathcal{A}}$ is well-defined.

The Mealy machine $\mathcal{M}^{\mathcal{A}}$ may be finite- or infinite-state. If a finite-state Mealy machine $\mathcal{M}^{\mathcal{A}} = \langle \Sigma_I^{\mathcal{A}}, \Sigma_O^{\mathcal{A}}, Q^{\mathcal{A}}, q_0^{\mathcal{A}}, \delta^{\mathcal{A}}, \lambda^{\mathcal{A}}\rangle$ is produced by the *Learner*, then we must finally "reverse" the effect of the abstraction $\mathcal{A}$ to obtain the original SMM $\mathcal{SM}$, such that $\mathcal{A}\langle\langle\mathcal{SM}\rangle\rangle$ is equivalent to $\mathcal{M}^{\mathcal{A}}$. In general, there can of course be many SMMs with this property. In order that the SMMs be determined uniquely up to equivalence (which is anyway the best we can hope for), it is necessary that each abstract output symbol correspond to a uniquely determined concrete output symbol generated by the SMM. We formulate this as follows.

Definition 6. An $\langle I, O\rangle$-abstraction $\mathcal{A} = \langle \Sigma_I^{\mathcal{A}}, \Sigma_O^{\mathcal{A}}, R, r_0, abstr_I, abstr_O, \delta^{\mathcal{R}}\rangle$ is *unambiguous* if for all abstract output symbols $b^{\mathcal{A}}$ and all $r \in R$ there is at most one output symbol b such that $b^{\mathcal{A}} = abstr_O(\delta^{\mathcal{R}}(r, a), b)$ for some input symbol $a \in \Sigma_I$. □

Intuitively, this means that we can deduce which output symbol is produced by $\mathcal{SM}$ by seeing only its abstraction.

If $\mathcal{A}$ is unambiguous, and $\mathcal{M}^{\mathcal{A}} = \langle \Sigma_I^{\mathcal{A}}, \Sigma_O^{\mathcal{A}}, Q^{\mathcal{A}}, q_0^{\mathcal{A}}, \delta^{\mathcal{A}}, \lambda^{\mathcal{A}}\rangle$ is a finite-state mealy machine, define $\mathcal{A}^{-1}\langle\langle\mathcal{M}^{\mathcal{A}}\rangle\rangle$ as be the Mealy machine $\langle \Sigma_I, \Sigma_O, Q^{\mathcal{A}} \times R, \langle q_0^{\mathcal{A}}, r_0\rangle, \delta, \lambda\rangle$, where δ and λ are defined by

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

We can now prove that under the conditions we have introduced, the SUT can be inferred from $\mathcal{M}^{\mathcal{A}}$, of course up to equivalence.

Proposition 1. *If $\mathcal{A}$ is unambiguous and adequate for $\mathcal{SM}$, and if $\mathcal{A}\langle\langle\mathcal{SM}\rangle\rangle$ is equivalent to $\mathcal{M}^{\mathcal{A}}$ (i.e., $\lambda^{\mathcal{A}\langle\langle\mathcal{SM}\rangle\rangle}(\langle q_0, r_0\rangle, u) = \{\lambda_{\mathcal{M}^{\mathcal{A}}}(u)\}$ for any $u \in (\Sigma_I^{\mathcal{A}})^*$), then $\mathcal{SM}$ is equivalent to $\mathcal{A}^{-1}\langle\langle\mathcal{M}^{\mathcal{A}}\rangle\rangle$.* □

The proposition can be proven by establishing that $\mathcal{A}^{-1}\langle\langle\mathcal{M}^{\mathcal{A}}\rangle\rangle$ satisfies the conditions of the propsition, i.e., that $\mathcal{A}\langle\langle\mathcal{A}^{-1}\langle\langle\mathcal{M}^{\mathcal{A}}\rangle\rangle\rangle\rangle$ is equivalent to $\mathcal{M}^{\mathcal{A}}$, and by establishing (e.g., by induction on the length of u) that the ouput generated in response to an input sequence u, by any $\mathcal{SM}$ such that $\mathcal{A}\langle\langle\mathcal{SM}\rangle\rangle$ is equivalent to $\mathcal{M}^{\mathcal{A}}$, is uniquely determined.

6 Illustrating Example

In this section, we sketch how a model of the booking service could be obtained by combining automata learning, and the abstraction that was developed in the course of the two previous sections.

Having supplied the abstraction described in the previous section, we can now employ the L^* algorithm to learn a finite-state Mealy machine, which interacts using the sets $\Sigma_I^{\mathcal{A}}$ and $\Sigma_O^{\mathcal{A}}$ of symbols. Assume that the result is as described by the finite-state Mealy machine in Figure 5. In this figure, we have omitted all transitions that return *error*, and concentrate on those that make the session progress.

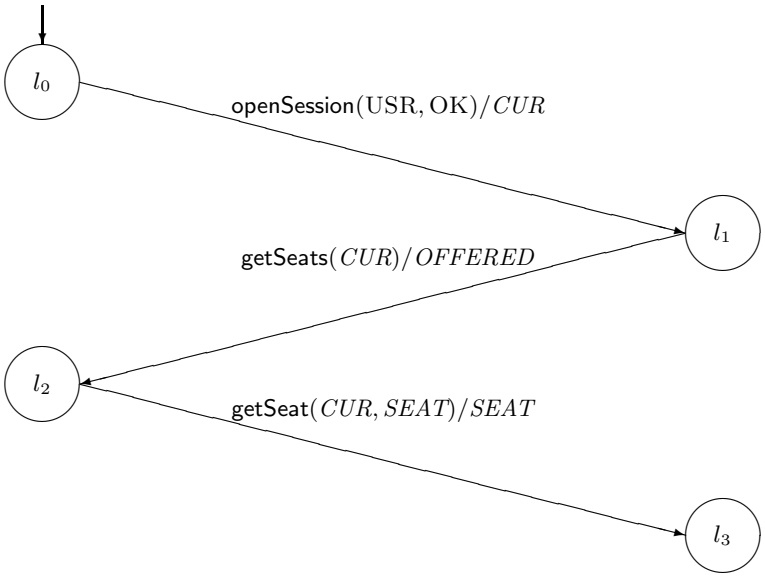


Fig. 5. Learned Abstract Mealy machine (self-loops suppressed)

Starting from the finite-state Mealy machine in Figure 5, we can apply the construction described at the end of the preceding section to generate a possible SMM that models the service. It is shown in Figure 6.

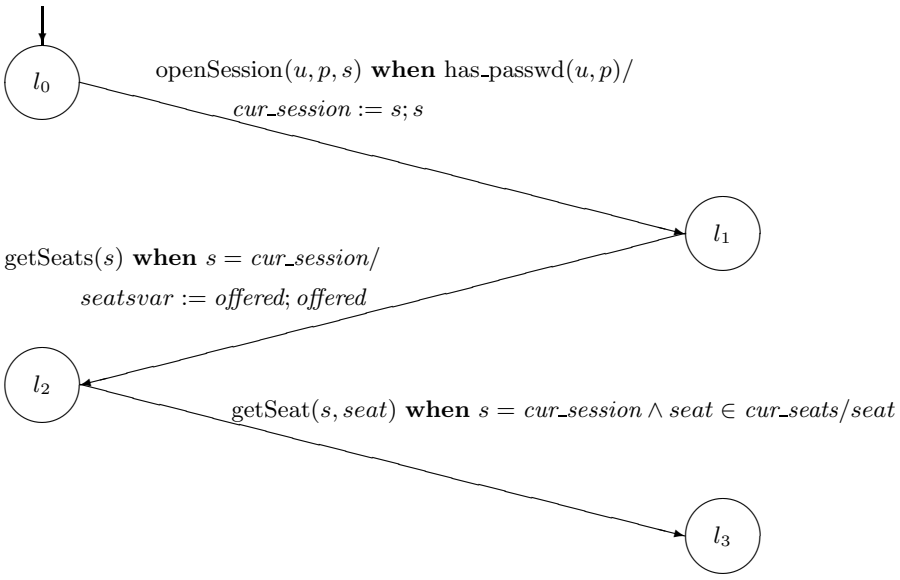


Fig. 6. Constructed Symbolic Mealy machine (self-loops suppressed)

7 Systematic Construction of Abstractions

The construction of a suitable abstraction is crucial for successful inference of an SMM $\mathcal{SM}$. In this subsection, we discuss how a successful abstraction can be constructued more systematically. A necessary prerequisite for constructing an abstraction is obviously that the sets I and O of input and output actions of $\mathcal{SM}$, together with their arities, are known *a priori*.

Furthermore, it appears necessary to have some *a priori* knowledge about how $\mathcal{SM}$ stores and manipulates data that it receives and emits. On the other hand, the control aspects of $\mathcal{SM}$ can be inferred by the *Learner* using automata learning, provided that the abstraction is "good enough". In the following, we give some sufficient criteria for "good enough" abstractions, under which learning and $\mathcal{SM}$, according to the technique described in Section 5 can be successful. At the end of this section, we discuss how such knowledge can sometimes be obtained by testing and experimentation on $\mathcal{SM}$.

In the running example in the previous subsection, we see that the abstraction mapping for input symbols uses expressions that become guards in the resulting

SMM, and that the abstraction mapping for output symbols uses expressions that occur in output expressions of the SMM. When $\mathcal{SM}$ is only available as a black box, such an abstraction can be produced if the following information is available.

- An "overestimate" of the information that is stored in state variables of $\mathcal{SM}$. More precisely, such an overestimate can be represented by a set R of states of the abstraction and an update function $\delta^{\mathcal{R}}$ such that after any sequence of pairs of input and output symbols, the information in the "data state", represented by the current valuation σ of state variables, of $\mathcal{SM}$ can be obtained from the current state r of the abstraction. One way to formalize is by finding a mapping h from the set of valuations of the state variables X to the set r of states of the *Mapper*, which has the properties that
  - $h(\sigma_0) = r_0$, and
  - for any symbolic transition in $\longrightarrow$ of form

    $(l) \xrightarrow{\alpha(\overline{p})\ \textbf{when}\ g\ /\ x_1,\ldots,x_k := e_1,\ldots,e_k\ ;\ \beta(e^{out}{}_1,\ldots,e^{out}{}_m)} (l')$

    and valuation σ and data values $\overline{d}$ such that $\sigma(g[\overline{d}/\overline{p}])$ evaluates to true, then if $h(\sigma) = r$, then it holds that for σ' defined by
    - $\sigma'(x_i) = \sigma(e_i[\overline{d}/\overline{p}])$ for $1 \leq i \leq k$, and
    - $\sigma'(x) = \sigma(x)$ if x is not among $x_1,\ldots,x_k$,

    we have $h(\sigma') = \delta^{\mathcal{R}}(r, \alpha(\overline{d})/\beta(\sigma'(e_1^{out}[\overline{d}/\overline{p}]),\ldots,\sigma'(e_m^{out}[\overline{d}/\overline{p}]))$.
- The abstraction should distinguish between the different symbolic transitions from a location. More precisely, this means that if from a location and from some state $\langle l, \sigma\rangle$ with $h(\sigma) = r$, there are two different symbolic transitions taken for input $\alpha(\overline{d})$ and $\alpha'(\overline{d}')$, then $\mathit{abstr}_I(r, \alpha(\overline{d})) \neq \mathit{abstr}_I(r, \alpha'(\overline{d}'))$. This can be achieved by letting each possible guard correspond to a different abstract input symbol. For the case that the abstraction is not fine enough to distinguish between symbolic transitions that cause different output, a technique for refining the abstraction on-the-fly, during the learning process, has been developed by Howar, Steffen, and Merten [19].
- The abstraction should be unambiguous. This can be achieved if different output expressions are mapped to different abstract output symbols. For instance, one could let abstract output symbols "be" the output expressions that can occur in symbolic transitions, assuming that an output expression is uniquely obtainable from the actual output symbol produced.

Under the above assumptions, we can construct an abstraction which maps combinations of parameterized input actions and guards in a possibe SMM to abstract input symbols, and maps combinations of expressions in output symbols of a possible SMM to abstract output symbols, as in the running example. The updates to state variables will simply consist in assigning some input parameters to state variables: the problem here is to decide which input parameters will influence the future behavior of $\mathcal{SM}$, and must be remembered in state variables. In our experiments, we have made this decision based on observing the response of $\mathcal{SM}$ to selected input strings, i.e., by posing membership queries, and saving

those parameter values that are used to produce future output. For parameter values on which the only performed operation is a test for equality, such as the *id* parameter of the running example, we have made these ideas more precise in our earlier work [6], as follows:

Consider an input string u, which contains a parameter value d. We observe the output of $\mathcal{M}$ in response to u and to selected continuations of u, and decide to store d in a state variable if there is some continuation v of u such that d is used to produce the response to v. More precisely, this happens if there is a fresh (i.e., previously unused) data value d' such that the response $\lambda(\delta(q_0, u), v)$ to v and the response $\lambda(\delta(q_0, u), v[d'/d])$ to $v[d'/d]$ (i.e., v where all occurrences of d have been replaced by d') satisfy $\lambda(\delta(q_0, u), v)[d'/d] \neq \lambda(\delta(q_0, u), v[d'/d])$, i.e., $\mathcal{SM}$ does not treat d in the same way as a fresh (previously unused) value d'. This happens, e.g., if $\lambda(\delta(q_0, u), v[d'/d])$ contains the data value d implying that d must have been remembered before seeing the subsequent input $v[d'/d]$, and that d should be stored in a state variable.

8 Conclusions and Future Work

We have considered the problem of extending automata learning to incorporate data parameters, including their influence on control behavior. We concentrated on presenting an approach that adapts ideas using abstraction that have been successfully applied in formal verification. This approach has been used on some nontrivial examples [1, 2], and techniques for revising abstractions by need have been developed [19]. However, it is clear that much work remains in order to make automata learning with data easily applicable to a wide class of systems. Issues that need to be addressed include to remove (some of) the need for manual construction of abstractions: this could be addressed by developing more canonical models for automata with data. Another issue is that the determinism of the Mealy machine model is limiting the modeling power: ways should be found to effectively learn nondeterministic models.

References

1. F. Aarts, B. Jonsson, and J. Uijen. Generating models of infinite-state communication protocols using regular inference with abstraction. In *Proc. ICTSS, 22nd IFIP WG 6.1 International Conference on Testing Software and Systems, Natal, Brazil, November 8-10, 2010*, volume 6435 of *Lecture Notes in Computer Science*, pages 188–204. Springer, 2010.
2. F. Aarts, J. Schmaltz, and F. Vaandrager. Inference and abstraction of the biometric passport. In *Proc. ISoLA, 4th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation 18-20 October 2010 - Amirandes, Heraclion, Crete*, volume 6415 of *Lecture Notes in Computer Science*, pages 673–686. Springer, 2010.
3. G. Ammons, R. Bodik, and J. Larus. Mining specifications. 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. T. Berg, B. Jonsson, and H. Raffelt. Regular inference for state machines with parameters. In L. Baresi and R. Heckel, editors, *FASE*, volume 3922 of *Lecture Notes in Computer Science*, pages 107–121. Springer, 2006.
6. T. Berg, B. Jonsson, and H. Raffelt. Regular inference for state machines using domains with equality tests. In J. L. Fiadeiro and P. Inverardi, editors, *FASE*, volume 4961 of *Lecture Notes in Computer Science*, pages 317–331. Springer, 2008.
7. M. Broy, B. Jonsson, J.-P. Katoen, M. Leucker, and A. Pretschner, editors. *Model-Based Testing of Reactive Systems*, volume 3472 of *Lecture Notes in Computer Science*. Springer Verlag, 2004.
8. Y. Brun and M. Ernst. Finding latent code errors via machine learning over program executions. In *ICSE'04: 26th Int. Conf. on Software Enginering*, May 2004.
9. 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.
10. J. Cobleigh, D. Giannakopoulou, and C. Pasareanu. Learning assumptions for compositional verification. In *Proc. TACAS '03, 9^{th} Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, volume 2619 of *Lecture Notes in Computer Science*, pages 331–346. Springer Verlag, 2003.
11. P. Dupont. Incremental regular inference. In L. Miclet and C. de la Higuera, editors, *ICGI*, volume 1147 of *Lecture Notes in Computer Science*, pages 222–237. Springer, 1996.
12. M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The daikon system for dynamic detection of likely invariants. *Sci. Comput. Program.*, 69(1-3):35–45, 2007.
13. E. M. Gold. Language identification in the limit. *Information and Control*, 10(5):447–474, 1967.
14. O. Grinchtein. *Learning of Timed Systems*. PhD thesis, Dept. of IT, Uppsala University, Sweden, 2008.
15. O. Grinchtein, B. Jonsson, and M. Leucker. Learning of event-recording automata. In *Proceedings of the Joint Conferences FORMATS and FTRTFT*, volume 3253 of *LNCS*, pages 379–396, Sept. 2004.
16. 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.
17. R. Groz, K. Li, A. Petrenko, and M. Shahbaz. Modular system verification by inference, testing and reachability analysis. In *TestCom/FATES*, volume 5047 of *Lecture Notes in Computer Science*, pages 216–233, 2008.
18. 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.
19. F. Howar, B. Steffen, and M. Merten. Automata learning with automated alphabet abstraction refinement. In *VMCAI, Verification, Model Checking, and Abstract Interpretation - 12th International Conference, VMCAI 2011, Austin, TX,*, volume 6538 of *Lecture Notes in Computer Science*, pages 263–277. Springer, 2011.

20. A. Huima. Implementing conformiq qtronic. In A. Petrenko, M. Veanes, J. Tretmans, and W. Grieskamp, editors, *Proc. TestCom/FATES, Tallinn, Estonia, June, 2007*, volume 4581 of *Lecture Notes in Computer Science*, pages 1–12, 2007.
21. H. Hungar, O. Niese, and B. Steffen. Domain-specific optimization in automata learning. In *Proc.* 15^{th} *Int. Conf. on Computer Aided Verification*, 2003.
22. V. Issarny, B. Steffen, B. Jonsson, G. S. Blair, P. Grace, M. Z. Kwiatkowska, R. Calinescu, P. Inverardi, M. Tivoli, A. Bertolino, and A. Sabetta. Connect challenges: Towards emergent connectors for eternal networked systems. In *ICECCS*, pages 154–161, 2009.
23. M. Kearns and U. Vazirani. *An Introduction to Computational Learning Theory*. MIT Press, 1994.
24. K. Li, R. Groz, and M. Shahbaz. Integration testing of distributed components based on learning parameterized I/O models. In E. Najm, J.-F. Pradat-Peyre, and V. Donzeau-Gouge, editors, *FORTE*, volume 4229 of *Lecture Notes in Computer Science*, pages 436–450, 2006.
25. D. Lo and S. Maoz. Scenario-based and value-based specification mining: better together. In *ASE 2010, 25th IEEE/ACM International Conference on Automated Software Engineering, Antwerp, Belgium*, pages 387–396. ACM, 2010.
26. 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.
27. D. Lorenzoli, L. Mariani, and M. Pezzè. Automatic generation of software behavioral models. In *Proc. ICSE'08: 30th Int. Conf. on Software Enginering*, pages 501–510, 2008.
28. L. Mariani and M. Pezzè. Dynamic detection of COTS components incompatibility. *IEEE Software*, 24(5):76–85, September/October 2007.
29. 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.
30. R. Rivest and R. Schapire. Inference of finite automata using homing sequences. *Information and Computation*, 103:299–347, 1993.
31. M. Shahbaz, K. Li, and R. Groz. Learning and integration of parameterized components through testing. In A. Petrenko, M. Veanes, J. Tretmans, and W. Grieskamp, editors, *TestCom/FATES*, volume 4581 of *Lecture Notes in Computer Science*, pages 319–334. Springer, 2007.
32. G. Shu and D. Lee. Testing security properties of protocol implementations - a machine learning based approach. In *Proc. ICDCS'07, 27th IEEE Int. Conf. on Distributed Computing Systems, Toronto, Ontario*. IEEE Computer Society, 2007.
33. B. Trakhtenbrot and J. Barzdin. *Finite automata: behaviour and synthesis*. North-Holland, 1973.