# Regular Inference for State Machines Using Domains with Equality Tests

Therese Berg[1], Bengt Jonsson[1], and Harald Raffelt[2]

[1] Department of Computer Systems, Uppsala University, Sweden
{thereseb,bengt}@it.uu.se
[2] Chair of Programming Systems and Compiler Construction, University of Dortmund, Germany
harald.raffelt@cs.uni-dortmund.de

**Abstract.** Existing algorithms for regular inference (aka automata learning) allows to infer a finite state machine by observing the output that the machine produces in response to a selected sequence of input strings. We generalize regular inference techniques to infer a class of state machines with an infinite state space. We consider Mealy machines extended with state variables that can assume values from a potentially unbounded domain. These values can be passed as parameters in input and output symbols, and can be used in tests for equality between state variables and/or message parameters. This is to our knowledge the first extension of regular inference to infinite-state systems. We intend to use these techniques to generate models of communication protocols from observations of their input-output behavior. Such protocols often have parameters that represent node adresses, connection identifiers, etc. that have a large domain, and on which test for equality is the only meaningful operation. Our extension consists of two phases. In the first phase we apply an existing inference technique for finite-state Mealy machines to generate a model for the case that the values are taken from a small data domain. In the second phase we transform this finite-state Mealy machine into an infinite-state Mealy machine by folding it into a compact symbolic form.

## 1 Introduction

Model-based techniques for verification and validation of reactive systems, such as model checking and model-based test generation [1] have witnessed drastic advances in the last decades. They depend on the availability of a formal model, specifying the intended behavior of a system or component, which ideally should be developed during specification and design. However, in practice often no such model is available, or becomes outdated as the system evolves over time, implying that a large effort in many model-based verification and test generation projects is spent on manually constructing a model from an implementation. It is therefore important to develop techniques for automating the task of generating models of existing implementations. A potential approach is to use program analysis to construct models from source code, as in software verification

J. Fiadeiro and P. Inverardi (Eds.): FASE 2008, LNCS 4961, pp. 317–331, 2008.
© Springer-Verlag Berlin Heidelberg 2008

(e.g., [2, 3]). However, many system components, including peripheral hardware components, library modules, or third-party components 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, 8, 9, 10]). This class of techniques has recently started to get attention in the testing and verification community, e.g., for regression testing of telecommunication systems [11, 12], and for combining conformance testing and model checking [13, 14]. They describe how to construct a finite-state machine (or a regular language) from the answers to a finite sequence 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 system under test (SUT). Angluin [4] and others introduce *equivalence queries*, queries to whether a hypothesized automaton is a correct model of the SUT, they can be seen as idealizing some procedure for extensively verifying (e.g., by conformance testing) whether the learning procedure is completed. The reply to an equivalence query is either *yes* or a counterexample, an input string on which the constructed automaton and the SUT respond with different output.

We intend to use regular inference to construct models of communication protocol entities. Such entities typically communicate by messages that consist of a protocol data unit (PDU) type with a number of parameters, each of which ranges over a sometimes large domain. Standard regular inference can only construct models with a moderately large finite alphabet. In previous work [15], we presented an optimization of regular inference to cope with models where the domain over which parameters range is large but finite. But, in order to fully support the generation of models with data parameters, we must consider a general theory for inference of infinite-state state machines with input and output symbols from potentially infinite domains.

In this paper, we present the first extension of regular inference to infinite-state state machines. We consider Mealy machines where input and output symbols are constructed from a finite number of message types that can have parameters from a potentially infinite domain. These parameters can be stored in state variables of the machine for later use. The only allowed operation on parameter values is a test for equality. The motivation is to handle parameters that, e.g., are identifiers of connections, objects, etc. This class of systems is similar to, and slightly more expressive than, the class of "data-independent" systems, which was the subject of some of the first works on model checking of infinite-state systems [16, 17].

In standard regular inference states and transitions are infered, and counterexamples to a hypothesized automaton are only used to add more states to the automaton. In this paper, we also infer state variables and operations on them, and counterexamples to a hypothesized model are used to extend the model with either more states or more state variables.

In our approach, we first observe the behavior of the protocol when the parameters of input messages are from a small domain. Using the regular inference algorithm by Niese [18] (which adapts Angluin's algorithm to Mealy machines), we generate a finite-state Mealy machine, which describes the behavior of the component on this small domain. We thereafter fold this finite-state Mealy machine into a smaller *symbolic* model.

*Organization.* The paper is organized as follows. In the next section, we review the Mealy machine model and in Section 3 we introduce the model for state machines using domains with equality tests. In Section 4 we review the inference algorithm for Mealy machines by Niese [18], and the adaptation required for our setting. In Section 5 we present our algorithm to map Mealy machines to Symbolic Mealy machines. Correctness and complexity of our algorithm is discussed in Section 6, and conclusions and future work are presented in Section 7.

*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 [19], for regression testing to create a specification and a test suite [11, 12], to perform model checking without access to code or to formal models [14, 13], for program analysis [20], and for formal specification and verification [19]. Li, Groz, and Shahbaz [21, 22] extend regular inference to Mealy machines with a finite subset of input and output symbols from the possible infinite set of symbols. This work resembles the intermediate model, in our earlier work [15], used to construct a symbolic model. In this work we handle infinite sets of parameter values, and can also generate infinite-state models with both control states and state variables that range over potentially infinite domains.

## 2 Mealy Machines

A *Mealy machine* is a tuple $\mathcal{M} = \langle \Sigma_I, \Sigma_O, Q, q_0, \delta, \lambda \rangle$ where $\Sigma_I$ is a finite nonempty set of *input symbols*, $\Sigma_O$ 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 $\Sigma_I^*$ and $\Sigma_O^*$ are (input and output, respectively) *strings* or *words*. Given $u, v \in \Sigma_I^*$, $u$ is said to be a *prefix* of $v$ if $v = uw$ for some $w \in \Sigma_I^*$.

An intuitive interpretation of a Mealy machine is as follows. At any point in time, the machine is in one state $q \in Q$. It is possible to give inputs to the machine, by supplying an input symbol $a$. The machine responds by producing an output string $\lambda(q, a)$ and transforming itself to the new state $\delta(q, a)$.

We extend the transition and output functions from input symbols to sequences of input symbols, by defining:

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

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 $\Sigma_I$,

i.e., $\delta$ and $\lambda$ are total. They are also *deterministic*, meaning that for each state $q$ and input $a$ exactly one next state $\delta(q, a)$ and output string $\lambda(q, a)$ is possible.

Let $q$ and $q'$ be two states of the same Mealy machine, or two states in different machines. The states $q$ and $q'$ are *equivalent* if $\lambda(q, u) = \lambda(q', u)$ for each input string $u \in \Sigma_I^*$. That is, for each input string the machine starting in $q$ will produce the same output string as the machine starting in $q'$. A Mealy machine $\mathcal{M}$ is *minimized* if there are no pair of states $q$ and $q'$ of $\mathcal{M}$, where $q \neq q'$, that are equivalent. There are well-known algorithms for efficiently minimizing a given Mealy machine [23]. Given a Mealy machine $\mathcal{M}$ with input alphabet $\Sigma_I$ 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 $\Sigma_I$ are *equivalent* if $\lambda_{\mathcal{M}} = \lambda_{\mathcal{M}'}$.

## 3 Symbolic Mealy Machines

In this section, we introduce Symbolic Mealy machines. They extend ordinary Mealy machines in that input and output symbols are messages with parameters, e.g., as in a typical communication protocol. We will specialize to the case when these parameters are from a large (in practice "infinite") domain $\mathcal{D}$, on which the only permitted operation is test for equality. In general, we could have several such domains, but let us here assume that all parameters are from one domain.

Let $I$ and $O$ be finite sets of *actions*, each of which has a nonnegative arity. Let $\mathcal{D}$ be a (finite or infinite) domain of data values. Let $\Sigma_I^{\mathcal{D}}$ be the set of *input symbols* of form $\alpha(d_1, \ldots, d_n)$, where $\alpha \in I$ is an action of arity $n$, and $d_1, \ldots, d_n \in \mathcal{D}$ are parameters from $\mathcal{D}$. The set of *output symbols* $\Sigma_O^{\mathcal{D}}$ is defined analogously.

We assume a set of *location variables*, ranged over by $v, v_1, v_2, \ldots$, and a set of *formal parameters*, ranged over by $p, p_1, p_2, \ldots$. A *symbolic value* is either a location variable or a formal parameter. We use $y$ or $z$ to range over symbolic values. A *parameterized input action* is a term of form $\alpha(p_1, \ldots, p_n)$, where $\alpha$ is an input action of arity $n$, and $p_1, \ldots, p_n$ are formal parameters. A *parameterized output action* is a term of form $\beta(z_1, \ldots, z_k)$, where $\beta \in O$ is an output action of arity $k$ and each $z_i$ is a symbolic value. We write $\overline{v}$ for $v_1, \ldots, v_m$, $\overline{y}$ for $y_1, \ldots, y_m$, $\overline{p}$ for $p_1, \ldots, p_n$, and $\overline{z}$ for $z_1, \ldots, z_k$. A *guard* over the location variables $\overline{v}$ and formal parameters $\overline{p}$ is a conjunction of equalities and inequalities between the symbolic values in $\overline{v}, \overline{p}$. A *guarded action* is of the form

$$\alpha(\overline{p}); g/\overline{v} := \overline{y}; \beta(\overline{z})$$

where $\alpha(\overline{p})$ is a parameterized input action, $g$ is a guard over location variables and the formal parameters $\overline{p}$, where $\overline{v} := \overline{y}$ is the assignment $\langle v_1, \ldots, v_m \rangle := \langle y_1, \ldots, y_m \rangle$, meaning that $v_i := y_i$, in which each $y_i$ is a location variable or a formal parameter, and $\beta(\overline{z})$ is a parameterized output action over symbolic values. An example of a guarded action is $\alpha(p_1, p_2); true/v_1 := p_1; \beta(p_2)$, where the first input parameter value is stored in a location variable, and the second input parameter value is output.

Let a *valuation function* $\rho$ be a mapping from a (possibly empty) set $Dom(\rho)$ of location variables to data values in $\mathcal{D}$. We let $\rho_0$ denote the valuation function with an empty domain. We extend valuation functions to operate on vectors of location variables by defining $\rho(v_1, \ldots, v_m)$ as $\rho(v_1), \ldots, \rho(v_m)$. For an input symbol $\alpha(\overline{d})$, a valuation function $\rho$, and guard $g$ such that $v_i \in Dom(\rho)$ for all $v_i$ occurring in $g$, we let $(\alpha(\overline{d}), \rho) \models g$ denote that $g[\overline{d}/\overline{p}, \rho(\overline{v})/\overline{v}]$ is true, i.e., that $g$ is true when the formal parameters $\overline{p}$ assume the data values $\overline{d}$ and the location variables $\overline{v}$ assume the data values $\rho(\overline{v})$.

**Definition 1 (Symbolic Mealy machine).** *A* Symbolic Mealy machine *is a tuple* $\mathcal{SM} = (I, O, L, l_0, \longrightarrow)$, *where*

- $I$ *is a finite set of* input actions*,*
- $O$ *is a finite set of* output actions*,*
- $L$ *is a finite set of* locations*, each of which has a nonnegative* arity *representing the number of location variables in that location,*
- $l_0 \in L$ *is the* initial location *with arity* 0*, and*
- $\longrightarrow$ *is a finite set of* transitions*. Each transition is a tuple* $\langle l, \alpha(\overline{p}), g, \overline{v} := \overline{y}, \beta(\overline{z}), l' \rangle$, *where* $l, l' \in L$ *are locations and* $\alpha(\overline{p}); g/\overline{v} := \overline{y}; \beta(\overline{z})$ *is a guarded action respecting constrains given by arities of* $l$ *and* $l'$*.* □

We write $l \xrightarrow{\alpha(\overline{p}); g/\overline{v}:=\overline{y}; \beta(\overline{z})} l'$ to denote $\langle l, \alpha(\overline{p}), g, \overline{v} := \overline{y}, \beta(\overline{z}), l' \rangle \in \longrightarrow$. We require that Symbolic Mealy machines are completely specified and deterministic, i.e., for each reachable $\langle l, \rho \rangle$ and symbol $\alpha(\overline{d})$, there is exactly one transition $\langle l, \alpha(\overline{p}), g, \overline{v} := \overline{y}, \beta(\overline{z}), l' \rangle$ from $l$ such that $(\alpha(\overline{d}), \rho) \models g$.

*Example.* An example of a Symbolic Mealy machine is shown in Figure 1. The Symbolic Mealy machine has one input action $\alpha$ with arity two, two output actions $\beta$ and $\beta'$ both with arity one, one location $l_0$ with arity zero and a second location $l_1$ with arity one.

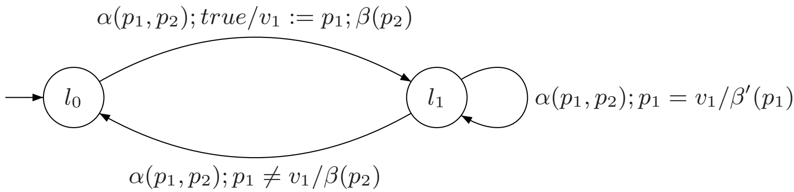


**Fig. 1.** Example of a Symbolic Mealy machine

Let $\mathcal{D}$ be a (finite or infinite) domain of data values. A Symbolic Mealy machine $\mathcal{SM} = (I, O, L, l_0, \longrightarrow)$ acting on the domain $\mathcal{D}$ is the (possibly infinite-state) Mealy machine $\mathcal{SM}_{\mathcal{D}} = \langle \Sigma_I^{\mathcal{D}}, \Sigma_O^{\mathcal{D}}, Q, \langle l_0, \rho_0 \rangle, \delta, \lambda \rangle$, where

- $\Sigma_I^{\mathcal{D}}$ is the set of input symbols,
- $\Sigma_O^{\mathcal{D}}$ is the set of output symbols,

- $Q$ is the set of pairs $\langle l, \rho \rangle$, where $l \in L$ and $\rho$ is a valuation function,
- $\langle l_0, \rho_0 \rangle$ is the initial state, and
- $\delta$ and $\lambda$ are defined as follows. Whenever $l \overset{\alpha(\overline{p});g/\overline{v}:=\overline{y};\beta(\overline{z})}{\longrightarrow} l'$ and $(\alpha(\overline{d}), \rho) \models g$ then
  - $\delta(\langle l, \rho \rangle, \alpha(\overline{d})) = \langle l', \rho' \rangle$, where $\rho'(v_i)$ is
    - $\rho(v_j)$ if $y_i$ is a location variable $v_j$,
    - $d_j$ if $y_i$ is a formal parameter $p_j$, and
  - $\lambda(\langle l, \rho \rangle, \alpha(\overline{d})) = \beta(d'_1, \ldots, d'_k)$, where $d'_i$ is
    - $\rho(v_j)$ if $z_i$ is a location variable $v_j$,
    - $d_j$ if $z_i$ is a formal parameter $p_j$.

Note that $\delta$ is well-defined since $\mathcal{SM}$ is completely specified and deterministic.

If $\mathcal{D}$ is finite, then $\mathcal{SM}_\mathcal{D}$ is a finite-state Mealy machine with at most $\sum_{l \in L} |\mathcal{D}|^{arity(l)}$ states, where $arity(l)$ is the arity of location $l$.

## 4 Inference of Symbolic Mealy Machines

In this section, we present our algorithm for inference of Symbolic Mealy machines. It is formulated in the same setting as Angluin's algorithm [4], in which a so called *Learner*, who initially knows nothing about $\mathcal{SM}$, is trying to infer $\mathcal{SM}$ by asking queries to a so called *Oracle*. The queries are of two kinds.

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

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

In our algorithm for the *Learner*, we build a hypothesis $\mathcal{H}$ in two phases: In the first phase we supply input from a small finite domain $\mathcal{E}$, and infer a hypothesis $\mathcal{M}$ for the Mealy machine $\mathcal{SM}_\mathcal{E}$ using an adaptation of Angluin's algorithm due to Niese [18]. There is an optimal smallest size of $\mathcal{E}$ which is still large enough to be able to exercise all tests for equalities between parameter values in $\mathcal{SM}$, but we do not *a priori* know this size. Therefore, we start with a small domain $\mathcal{E}$ that can be gradually extended when feedback from equivalence queries makes it necessary.

To characterize the "optimal smallest size", let $\mathcal{M}$ be $\langle \Sigma_I^\mathcal{E}, \Sigma_O^\mathcal{E}, Q, q_0, \delta, \lambda \rangle$, where $I, O$ are the input and output action alphabet, respectively. We say that

a data value $d \in \mathcal{E}$ is *fresh* in a state $q \in Q$ if there exists an input string $u \in (\Sigma_I^{\mathcal{E}})^*$ which leads to $\delta(q_0, u) = q$ such that $d$ does not occur as a data value in $u$. We say that $\mathcal{M}$ is *fresh* if for each state $q \in Q$, there are $n$ data values in $\mathcal{E}$ which are fresh in $q$, where $n$ is the maximal arity of input actions in $I$. A sufficient condition on $\mathcal{E}$ is given by the following Lemma 1.

**Lemma 1.** *Let $\mathcal{SM} = (I, O, L, l_0, \longrightarrow)$ be a Symbolic Mealy machine, $\mathcal{E}$ be a domain of data values, $m$ be the maximal arity of the locations in $L$, and $n$ be the maximal arity of the input actions in $I$. Then if the size of $\mathcal{E}$ is bigger than $m + n$, the Mealy machine $\mathcal{SM}_{\mathcal{E}}$ is fresh.* □

Returning to our inference algorithm, if the hypothesis $\mathcal{M}$ generated by Niese's algorithm is not fresh, we do not make any equivalence query, but instead enlarge $\mathcal{E}$ by one, and continue Niese's algorithm with the new enlarged $\mathcal{E}$. If $\mathcal{M}$ is fresh, we transform $\mathcal{M}$ into a Symbolic Mealy machine $\mathcal{H}$ such that $\mathcal{H}_{\mathcal{E}}$ is equivalent to $\mathcal{M}$: this transformation is presented in Section 5. Thereafter $\mathcal{H}$ is supplied in an equivalence query, to find out whether $\mathcal{H}$ is equivalent to $\mathcal{SM}$. If the result is successful, the algorithm terminates. Otherwise, the counterexample returned needs to be analyzed, since it may contain values outside of $\mathcal{E}$. Let the counterexample be input string $u \in \Sigma_I^{\mathcal{D}}$, with data values from domain $\mathcal{D}$.

In the case that $|\mathcal{D}| \leq |\mathcal{E}|$, we apply any injective mapping from $\mathcal{D}$ to $\mathcal{E}$, on the data values in the counterexample, and use the mapped counterexample in Niese's algorithm.

In the case that $|\mathcal{D}| > |\mathcal{E}|$, we try to find a mapping from $\mathcal{D}$ to a subset $\mathcal{D}' \subseteq \mathcal{D}$ which is as small as possible, but such that the mapped counterexample is still a counterexample to $\mathcal{H}$. To find such a mapping, one can either use an exhaustive search or a heuristic search guided by $\mathcal{H}$. The search involves asking more membership queries. We then extend $\mathcal{E}$ to the same size as $\mathcal{D}'$, and continue Niese's algorithm with the mapped counterexample.

## 5 Transforming Mealy Machines to Symbolic Mealy Machines

In this section, we present the transformation from a Mealy machine $\mathcal{M}$ to a Symbolic Mealy machine $\mathcal{SM}$. Throughout this section, we will use the Symbolic Mealy machine in Figure 1 as a running example to illustrate the steps in our algorithm. We assume that we used the domain $\mathcal{E} = \{1, 2, 3\}$ of size 3, which is the smallest domain to make $\mathcal{SM}_{\mathcal{E}}$ fresh, and that we obtained the hypothesis Mealy machine shown in Figure 2.

In the second phase, we transform the Mealy machine $\mathcal{M}$ into a Symbolic Mealy machine $\mathcal{SM}$, which must "simulate" $\mathcal{M}$ in the sense that $\mathcal{SM}_{\mathcal{E}}$ is equivalent to $\mathcal{M}$. The transformation algorithm has four steps.

- In the first step, the algorithm figures out for each state of n $\mathcal{M}$ which data values must be remembered by a corresponding Symbolic Mealy machine in order to produce its future behavior. These data values are the basis for

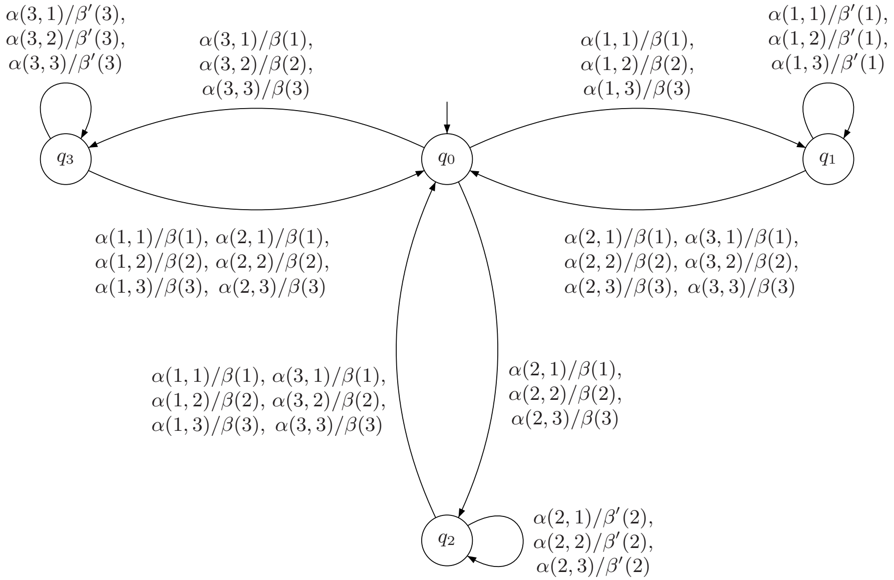


**Fig. 2.** A Mealy machine constructed with the inference algorithm applied to the example in Figure 1. All transitions that have the same start and target state are depicted with one edge.

constructing the location variables required in the corresponding location in the Symbolic Mealy machine. In the running example of Figure 2, the algorithm figures that, e.g., in state $q_1$ the data value 1 must be remembered.
- In the second step, we use the data values inferred in the first step to transform transitions of $\mathcal{M}$ into a so called symbolic normal form, which is especially designed to capture exactly the equalities and inequalities between formal parameters and location variables. In the running example, the transition on $\alpha(1,2)/\beta'(1)$ from, e.g., location $q_1$, will be transformed into $\alpha(v_1, p_1)/v_1 := v_1; \beta'(v_1)$.
- In the third step, we merge states of $\mathcal{M}$ into locations of $\mathcal{SM}$, if the symbolic forms of their future behavior are the same, using an adaptation of a standard partition-refinement algorithm. In the running example, states $q_1$, $q_2$, and $q_3$ will be merged into one location.
- In the fourth and final step, we transform transitions from symbolic normal form to the standard form used in Definition 1, and merge transitions when possible. The result of the first part of this step is shown in Figure 4.

*Step One.* The first step in our algorithm is to compute a *state-value function* $V : Q \mapsto 2^{\mathcal{E}}$ from states in $\mathcal{M}$ to sets of data values, which for each state gives the set of data values that $\mathcal{M}$ must remember for its future behavior. We first observe that if the data value $d$ is fresh in a state $q$, then obviously $d$ should not be in $V(q)$. Furthermore, if a data value $d'$ is not remembered by $\mathcal{M}$ in $q$,

then the future behavior from $q$ remains unchanged if we swap the roles of $d$ and $d'$ (note that $d$ and $d'$ will occur in future input symbols). Let $u$ be the input string which takes $\mathcal{M}$ from its initial state to $q$ and does not contain $d$. The input string $u'$, obtained by replacing all occurrences of $d'$ by $d$ in $u$ will, by symmetry, take $\mathcal{M}$ from its initial state to a state $q'$ whose future behavior has swapped the roles of $d$ and $d'$ as compared with $q$. Since $\mathcal{M}$ is minimized, $q$ must be the same state as $q'$. Thus $d'$ is also in fresh in $q$. In summary, $V(q)$ should contain all data values that are not fresh in $q$.

The calculation of the state-value function in the example in Figure 2 yields that $V(q_0)$ is the empty set $\emptyset$, $V(q_1)$ is $\{1\}$, $V(q_2)$ is $\{2\}$, and $V(q_3)$ is $\{3\}$.

*Step Two.* In the second step, we transform transition labels into a symbolic form. Intuitively, the transition $q \xrightarrow{\alpha(\overline{d}^I)/\beta(\overline{d}^O)} q'$ will be transformed into a symbolic transition $q \xrightarrow{\alpha(\overline{p});g/\overline{v}:=\overline{y};\beta(\overline{z})} q'$ where $g$ is as strong as possible. We will use a different representation for such a symbolic transition, which we call *symbolic normal form*, which does not use guards: instead each data value in $\overline{d}^I$ and $\overline{d}^O$ will be replaced by $v_i$ if it occurs as the $i$th data value stored in state $q$, otherwise by an appropriate input parameter symbol $p_j$. Furthermore, different symbolic values are implicitly required to be different. A problem is that we cannot know in which "order" the data values in $V(q)$ will be mapped to location variables, so therefore the transformation will depend on a specific ordering $\overline{d}$ of the data values in $V(q)$ and a specific ordering $\overline{d}'$ of the data values in $V(q')$.

Let us define precisely the symbolic normal form. For each ordering $\overline{d}$ of the data values in $V(q)$, for each vector $\overline{d}^I$ of data values received in an input symbol, and for each data value $d$ in $\overline{d} \cup \overline{d}^I$, define $SV_{\overline{d},\overline{d}^I}(d)$ as

- $v_i$ if there exists $d_i \in \overline{d}$ such that $d = d_i$, or else
- $p_k$ if $j$ is the smallest index such that $d_j \in \overline{d}^I$ is $d_j = d$, and $k$ is the number of unique data values $d_l \in \overline{d}^I$ with index $l \leq j$, such that $d_l$ does not appear in $\overline{d}$,

We extend $SV_{\overline{d},\overline{d}^I}$ to vectors of data values, by defining $SV_{\overline{d},\overline{d}^I}(d_1,\ldots,d_n)$ as $SV_{\overline{d},\overline{d}^I}(d_1),\ldots,SV_{\overline{d},\overline{d}^I}(d_n)$.

For each ordering $\overline{d}$ of the data values in $V(q)$ and each ordering $\overline{d}'$ of the data values in $V(q')$, the *symbolic normal form* of $q \xrightarrow{\alpha(\overline{d}^I)/\beta(\overline{d}^O)} q'$ is defined as

$$(q,\overline{d}) \xrightarrow{\alpha(SV_{\overline{d},\overline{d}^I}(\overline{d}^I))/\overline{v}:=\overline{y};\beta(SV_{\overline{d},\overline{d}^I}(\overline{d}^O))} (q',\overline{d}'),$$

where $\overline{v} := \overline{y}$ is an assignment in which $y_i$ is

- $SV_{\overline{d},\overline{d}^I}(d_j)$ if $d'_i = d_j$, for some $d'_i \in \overline{d}'$ and $d_j \in \overline{d}$, or else
- $SV_{\overline{d},\overline{d}^I}(d^I_j)$ if $d'_i = d^I_j$, for some $d'_i \in \overline{d}'$ and $d^I_j \in \overline{d}^I$.

As an example, the symbolic normal form of $q_0 \stackrel{\alpha(1,2)/\beta(2)}{\longrightarrow} q_1$, a transition in the Mealy machine in Figure 2, calculates to $(q_0, []) \stackrel{\alpha(p_1,p_2)/v_1:=p_1;\beta(p_2)}{\longrightarrow} (q_1, 1)$, where $[]$ is the empty vector of data values.

*Step Three.* In the third step, we merge states of $\mathcal{M}$ if the symbolic forms of their future behaviors are equivalent. As explained in the description of Step two, the symbolic normal form of the behavior from a state $q$ is defined only with respect to a given ordering $\overline{d}$ of the data values in $V(q)$, meaning that for each state $q$ we must fix some ordering of the stored data values. However, since some combinations of orderings allow to merge more states and obtain smaller machines than others, we shall not fix this ordering *a priori*. Instead, we create several copies of each $q$, one for each possible ordering of the data values in $V(q)$, and thereafter perform the merging starting with all these copies. Since at the end, we need only one pair of form $(q, \overline{d})$ for each $q$, we will prune copies of $q$ that will create additional states, as long as at least one copy of each $q$ remains.

Thus, our partitioning algorithm partitions pairs $(q, \overline{d})$ of states and data-value vectors into blocks $\mathcal{B}_1, \ldots, \mathcal{B}_m$, representing potential locations in a Symbolic Mealy machine. Each block $\mathcal{B}_i$ is a set of pairs $(q, \overline{d})$, where $q \in Q$ and $\overline{d}$ is some ordering of the data values $V(q)$. To break the symmetry between different orderings of data values, we pick for each block $\mathcal{B}_i$ an arbitrary pair $(q, \overline{d}) \in \mathcal{B}_i$ which is called *representative* for $\mathcal{B}_i$, to represent how the symbolic transitions from $\mathcal{B}_i$ will look like. The goal of the partitioning is that each block should be consistent, as defined in the following definition.

**Definition 2 (Block consistency for a block $\mathcal{B}$).** *Let $(q, \overline{d}) \in \mathcal{B}$ be the representative for block $\mathcal{B}$. Block $\mathcal{B}$ is* consistent *if whenever $(r, \overline{e}) \in \mathcal{B}$ there is a transition $(r, \overline{e}) \stackrel{\alpha(\overline{z})/\overline{v}:=\overline{y};\beta(\overline{z}')}{\longrightarrow} (r', \overline{e}')$ (on symbolic normal form) iff there is a symbolic transition $(q, \overline{d}) \stackrel{\alpha(\overline{z})/\overline{v}:=\overline{y};\beta(\overline{z}')}{\longrightarrow} (q', \overline{d}')$ (on symbolic normal form) with the same label, such that $(r', \overline{e}')$ and $(q', \overline{d}')$ are in the same block.* □

We find a partitioning of pairs into consistent blocks by fix-point iteration, using a variation of the standard partition-refinement algorithm, as follows.

- Initially, for each $i$ which is the size of $V(q)$ for some $q \in Q$, there is a block $\mathcal{B}_i$ with all pairs $(q, \overline{d})$ such that $\overline{d}$ has exactly $i$ data values.
- Repeat the following step until convergence.
  - Pick a block $\mathcal{B}_i$ and let $(q, \overline{d})$ be the representative for $\mathcal{B}_i$.
  - Split $\mathcal{B}_i$ by letting a pair $(r, \overline{e})$ in $\mathcal{B}_i$ remain in the block if for all $\alpha(\overline{z})$ there is a symbolic transition $(q, \overline{d}) \stackrel{\alpha(\overline{z})/\overline{v}:=\overline{y};\beta(\overline{z}')}{\longrightarrow} (q', \overline{d}')$ from the representative $(q, \overline{d})$ iff there is a symbolic transition $(r, \overline{e}) \stackrel{\alpha(\overline{z})/\overline{v}:=\overline{y};\beta(\overline{z}')}{\longrightarrow} (r', \overline{e}')$ from $(r, \overline{e})$ with the same label, such that $(q', \overline{d}')$ and $(r', \overline{e}')$ are in the same block. Let $\mathcal{B}_i'$ be the set of all pairs $(r, \overline{e})$ that were originally in $\mathcal{B}_i$ but did not pass this test.
  - Delete from $\mathcal{B}_i'$ all pairs $(r, \overline{e})$ for which some other pair $(r, \overline{e}')$ with the same state $r$ remains in $\mathcal{B}_i$. It should be noted that this deletion of copies

is safe due to the strong symmetry between different orderings of data values $\overline{e}$ in pairs of form $(r, \overline{e})$.
- If $\mathcal{B}'_i$ is thereafter non-empty, we let $\mathcal{B}'_i$ be a new block, and choose an arbitrary member in $\mathcal{B}'_i$ as its representative,

– The algorithm terminates when all blocks are consistent. Let $\{\mathcal{B}_1, \ldots, \mathcal{B}_f\}$ denote the final set of blocks.

From the example in Figure 2 we create two consistent blocks, $\mathcal{B}_0$ and $\mathcal{B}_1$, see Figure 3. Block $\mathcal{B}_0$ contains the pair $(q_0, [])$, and block $\mathcal{B}_1$ contains the pairs $(q_1, 1)$, $(q_2, 2)$, and $(q_3, 3)$. Let us choose $(q_0, [])$ as the representative for block $\mathcal{B}_0$, and $(q_1, 1)$ as the representative for block $\mathcal{B}_1$. In the figure, from block $\mathcal{B}_1$ only the labels on the outgoing transitions from one of the pairs is drawn, since all pairs have the same labels on outgoing transitions.

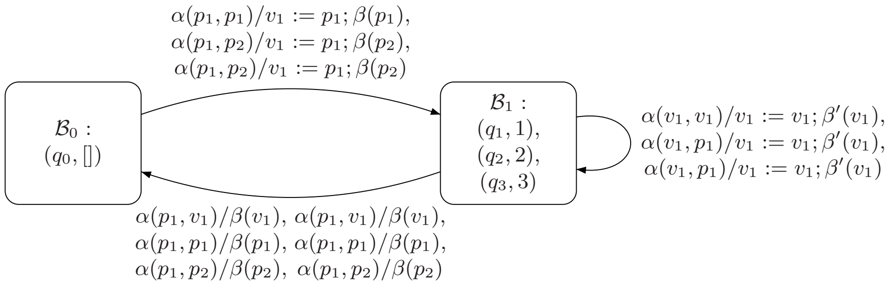


**Fig. 3.** The set of consistent blocks constructed from the example in Figure 2

*Step Four.* In the fourth step, we complete the transformation of the Mealy machine $\mathcal{M}$ by creating a Symbolic Mealy machine $\mathcal{SM}$ from the result of step three. This also involves transforming the symbolic normal form of transitions into the form used in Definition 1, and merging transitions. The locations of $\mathcal{SM}$ correspond to the blocks resulting from the third step. When creating the transitions of $\mathcal{SM}$, we can select an arbitrary member $(q, \overline{d})$ in each block $\mathcal{B}_i$, and use only these selected members and the transitions between them to construct $\mathcal{SM}$. It is possible to select only one member since, by block consistency, all members are equivalent. For simplicity we let the block representative be the selected member.

Each transition between representatives must be transformed from symbolic normal form to the form used in Definition 1. As an example, consider the self-loop from the representative $(q_1, 1)$ in Figure 3 labeled by $\alpha(v_1, p_1)/v_1 := v_1; \beta'(v_1)$. It states that if an input symbol $\alpha(d_1, d_2)$ is received where $d_1$ is equal to the current value of the location variable $v_1$, and $d_2$ is different from the value of any location variable, then the location variable is unchanged, and $\beta'(d_1)$ is generated. It should obtain the label $\alpha(p_1, p_2) : (p_1 = v_1 \wedge p_2 \neq v_1)/v_1 := v_1; \beta'(v_1)$, to conform with Definition 1. To define this transformation precisely,

for a vector $\overline{z}$ of symbolic values, let $T_{\overline{z}}$ be a mapping from symbolic values to symbolic values, defined as

- $T_{\overline{z}}(v_j) = v_j$ for any location variable $v_j$,
- $T_{\overline{z}}(p_j) = p_i$ if $p_j$ is a formal parameter which occurs in $\overline{z}$, and $i$ is the smallest value such that $p_j$ is the $i$th element $z_i$ in $\overline{z}$.

$T_{\overline{z}}$ is extended to vectors of symbolic values, by defining $T_{\overline{z}}(y_1, \ldots, y_n)$ as $T_{\overline{z}}(y_1), \ldots, T_{\overline{z}}(y_n)$. For example $T_{p_1,p_1,v_1,p_2}(p_2, v_1, p_1, p_1) = p_4, v_1, p_1, p_1$.

For a vector $\overline{z}$ of symbolic values and vector $\overline{v}$ of location variables, define the guard $g_{\overline{z},\overline{v}}$ over formal parameters $\overline{p}$ and location variables $\overline{v}$ as the conjunction of the following conjuncts:

- for each formal parameter $p_j$ which is the $i$th element $z_i$ in $\overline{z}$:
  - $p_i \neq v_k$ for all location variables $v_k$ in $\overline{v}$,
  - $p_i = p_k$ whenever $p_j$ is both the $i$th and the $k$th element of $\overline{z}$, and $k$ is the largest index such that $p_j$ is the $k$th element of $\overline{z}$,
  - $p_i \neq p_k$ whenever the $k$th element $z_k$ in $\overline{z}$ is a formal parameter different from $p_j$ and $i < k$,
- for each location variable $v_j$ which is the $i$th element $z_i$ in $\overline{z}$:
  - $p_i = v_j$.

From the set of consistent blocks $\{\mathcal{B}_1, \ldots, \mathcal{B}_f\}$ resulting from the third step, starting from $\mathcal{M} = \langle \Sigma_I^{\mathcal{E}}, \Sigma_O^{\mathcal{E}}, Q, q_0, \delta, \lambda \rangle$, we can now construct a Symbolic Mealy machine $\mathcal{SM} = (I, O, L, l_0, \longrightarrow)$, where

- $I$ is the set of input actions,
- $O$ is the set of output actions,
- $L$ is the set of blocks $\{\mathcal{B}_1, \ldots, \mathcal{B}_f\}$; the arity of each location $\mathcal{B}_i \in L$ is the size of $\overline{d}$, where $(q, \overline{d})$ is the representative of $\mathcal{B}_i$,
- $l_0 \in L$ is the block among $\mathcal{B}_1, \ldots, \mathcal{B}_f$ that contains the pair $(q_0, [])$,
- $\longrightarrow$ contains for each symbolic normal form $(q, \overline{d}) \xrightarrow{\alpha(\overline{z})/\overline{v}':=\overline{y};\beta(\overline{z}')} (q', \overline{d}')$ of a transition between representative $q$ of block $\mathcal{B}$ and representative $q'$ of $\mathcal{B}'$, the transition
$$\mathcal{B} \xrightarrow{\alpha(\overline{p})\ ;\ g_{\overline{z},\overline{v}}\ /\ \overline{v}':=T_{\overline{z}}(\overline{y})\ ;\ \beta(T_{\overline{z}}(\overline{z}'))} \mathcal{B}' ,$$
where $\overline{v}$ is the location variables of $\mathcal{B}$, and $\overline{p}$ is the vector $p_1, \ldots, p_n$ of formal parameters where $n$ is the size of $\overline{z}$.

In general, the resulting Symbolic Mealy machine in general has "too small" transitions, since each guard completely characterizes equalities and inequalities between the formal input parameter $\overline{p}$ and the location variables. To get a final Symbolic Mealy machine, we merge transitions that differ only in their guards, by taking the disjunction of their guards, whenever the new guard is still a conjunction.

In the final step in our work with our example, we create a Symbolic Mealy machine from the set of consistent blocks shown in Figure 3. The initial construction, after transforming transitions from symbolic normal form is the Symbolic

Mealy machine shown in Figure 4. We thereafter merge the two self loops in Figure 4 into $\alpha(p_1, p_2) : p_1 = v_1/v_1 := v_1; \beta'(v_1)$. On the other edges, the parameterized output actions can be replaced by $\beta(p_2)$ without altering the output behavior, then we merge transitions, obtaining the Symbolic Mealy machine in Figure 1.

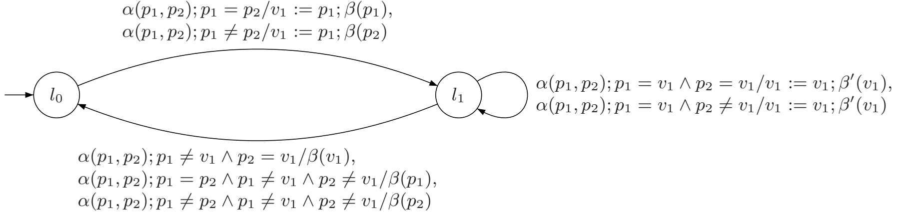


**Fig. 4.** Intermediate Symbolic Mealy machine constructed from the blocks in Figure 3

## 6 Correctness and Complexity

The correctness of our transformation from a Mealy machine $\mathcal{M}$ to a Symbolic Mealy machine $\mathcal{SM}$ follows from the following two theorems.

**Theorem 1.** *Let $\mathcal{E}$ be a finite domain, and let $\mathcal{M} = \langle \Sigma_I^{\mathcal{E}}, \Sigma_O^{\mathcal{E}}, Q, q_0, \delta, \lambda \rangle$ be a finite-state (minimized) Mealy machine. If $\mathcal{SM} = (I, O, L, \longrightarrow, l_0)$ is the result of the transformation described in Section 5, then $\mathcal{M}$ and $\mathcal{SM}_{\mathcal{E}}$ are equivalent (i.e., they produce the same output for all input strings $u \in (\Sigma_I^{\mathcal{E}})^*$).* □

**Theorem 2.** *Let $\mathcal{SM}$ and $\mathcal{SM}'$ be Symbolic Mealy machines with the same set of input actions $I$ and output actions $O$. Let $m$ be the maximal arity of the locations in $\mathcal{SM}$ and $\mathcal{SM}'$, and let $n$ be the maximal arity of the input actions in $I$. If the size of $\mathcal{E}$ is bigger than $m+n$, and if $\mathcal{SM}_{\mathcal{E}}$ and $\mathcal{SM}'_{\mathcal{E}}$ are equivalent, then for any data-value domain $\mathcal{D}$ the Mealy machines $\mathcal{SM}_{\mathcal{D}}$ and $\mathcal{SM}'_{\mathcal{D}}$ are equivalent.* □

The two preceding theorems imply that if $\mathcal{SM}$ is the Mealy machine which we attempt to learn, and $\mathcal{SM}'$ is the machine we construct, then $\mathcal{SM}_{\mathcal{E}}$ is equivalent to $\mathcal{SM}'_{\mathcal{E}}$ for any domain $\mathcal{E}$.

An upper bound on the number of membership queries can be obtained from the corresponding bound on the number of membership queries needed to infer a Mealy machine $\mathcal{M} = \langle \Sigma_I^{\mathcal{E}}, \Sigma_O^{\mathcal{E}}, Q, q_0, \delta, \lambda \rangle$, which is $\mathcal{O}(|\Sigma_I^{\mathcal{E}}| \times |Q| \times \max(|\Sigma_I^{\mathcal{E}}|, |Q|) \times C)$, where

- $|\Sigma_I^{\mathcal{E}}| = \sum_{\alpha \in I} |\mathcal{E}|^{actarity(\alpha)}$, where $actarity(\alpha)$ is the arity of $\alpha$,
- $|Q| = \sum_{l \in L} |\mathcal{E}|^{arity(l)}$,
- $C$ is the length of the longest counterexample returned in equivalence queries.

where $|\mathcal{E}|$ can be chosen as $m + n + 1$ (in the notation of Lemma 1) to make sure that $\mathcal{SM}_{\mathcal{E}}$ is fresh. The maximum number of equivalence queries required by the algorithm is $|Q| + |\mathcal{E}|$. To infer our example in Figure 2 we performed 333 membership queries and 1 equivalence query.

A way to reduce the number of membership queries required to infer the Mealy machine is to use a symmetry filter which deduces answers to membership queries from already answered membership queries [12]. The symmetry filter will filter out membership queries which have the same differences and equalities between parameter values as an already answered membership query.

## 7 Conclusions and Future Work

We have extended regular inference to a class of state machines with infinite state-spaces and infinite alphabets. Our motivation is to develop techniques for inferring models of entities in communication protocols by observing test executions. It would be interesting to try to extend our approach to data domains with more complex operations, such as counters, time-stamps, etc.

Our two-phase approach implies that the intermediate finite-state Mealy machine may get rather large, in comparison with the final Symbolic Mealy machine. This problem might be mitigated by developing an algorithm where the generation of the intermediate machine and the compacting transformation are performed "in parallel". Next on our agenda is to apply the results of this paper in a case study on realistic communication protocols.

*Acknowledgement.* We thank B. Steffen and J. Parrow for helpful discussions.

## References

1. Broy, M., Jonsson, B., Katoen, J.-P., Leucker, M., Pretschner, A. (eds.): Model-Based Testing of Reactive Systems. LNCS, vol. 3472. Springer, Heidelberg (2005)
2. Ball, T., Rajamani, S.: The SLAM project: Debugging system software via static analysis. In: Proc. $29^{th}$ ACM Symp. on Principles of Programming Languages, pp. 1–3 (2002)
3. Henzinger, T., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. In: Proc. $29^{th}$ ACM Symp. on Principles of Programming Languages, pp. 58–70 (2002)
4. Angluin, D.: Learning regular sets from queries and counterexamples. Information and Computation 75, 87–106 (1987)
5. Balcázar, J., Díaz, J., Gavaldá, R.: Algorithms for learning finite automata from queries: A unified view. In: Advances in Algorithms, Languages, and Complexity, pp. 53–72. Kluwer Academic Publishers, Dordrecht (1997)
6. Dupont, P.: Incremental regular inference. In: Miclet, L., de la Higuera, C. (eds.) ICGI 1996. LNCS, vol. 1147, pp. 222–237. Springer, Heidelberg (1996)
7. Gold, E.M.: Language identification in the limit. Information and Control 10, 447–474 (1967)
8. Kearns, M., Vazirani, U.: An Introduction to Computational Learning Theory. MIT Press, Cambridge (1994)

9. Rivest, R., Schapire, R.: Inference of finite automata using homing sequences. Information and Computation 103, 299–347 (1993)
10. Trakhtenbrot, B.A., Barzdin, J.M.: Finite automata: behaviour and synthesis. North-Holland, Amsterdam (1973)
11. Hagerer, A., Hungar, H., Niese, O., Steffen, B.: Model generation by moderated regular extrapolation. In: Kutsche, R.-D., Weber, H. (eds.) FASE 2002. LNCS, vol. 2306, pp. 80–95. Springer, Heidelberg (2002)
12. Hungar, H., Niese, O., Steffen, B.: Domain-specific optimization in automata learning. In: Proc. $15^{th}$ Int. Conf. on Computer Aided Verification (2003)
13. Peled, D., Vardi, M.Y., Yannakakis, M.: Black box checking. In: Wu, J., Chanson, S.T., Gao, Q. (eds.) Formal Methods for Protocol Engineering and Distributed Systems, FORTE/PSTV, Beijing, China, pp. 225–240. Kluwer Academic Publishers, Dordrecht (1999)
14. Groce, A., Peled, D., Yannakakis, M.: Adaptive model checking. In: Katoen, J.-P., Stevens, P. (eds.) TACAS 2002. LNCS, vol. 2280, pp. 357–370. Springer, Heidelberg (2002)
15. Berg, T., Jonsson, B., Raffelt, H.: Regular inference for state machines with parameters. In: Baresi, L., Heckel, R. (eds.) FASE 2006. LNCS, vol. 3922, pp. 107–121. Springer, Heidelberg (2006)
16. Wolper, P.: Expressing interesting properties of programs in propositional temporal logic (extended abstract). In: Proc. $13^{th}$ ACM Symp. on Principles of Programming Languages, pp. 184–193 (1986)
17. Jonsson, B., Parrow, J.: Deciding bisimulation equivalences for a class of non-finite-state programs. Information and Computation 107(2), 272–302 (1993)
18. Niese, O.: An integrated approach to testing complex systems. Technical report, Dortmund University, Doctoral thesis (2003)
19. Cobleigh, J.M., Giannakopoulou, D., Pasareanu, C.S.: Learning assumptions for compositional verification. In: Garavel, H., Hatcliff, J. (eds.) TACAS 2003. LNCS, vol. 2619, pp. 331–346. Springer, Heidelberg (2003)
20. Ammons, G., Bodik, R., Larus, J.: Mining specificatoins. In: Proc. $29^{th}$ ACM Symp. on Principles of Programming Languages, pp. 4–16 (2002)
21. Li, K., Groz, R., Shahbaz, M.: Integration testing of distributed components based on learning parameterized I/O models. In: Najm, E., Pradat-Peyre, J.-F., Donzeau-Gouge, V.V. (eds.) FORTE 2006. LNCS, vol. 4229, pp. 436–450. Springer, Heidelberg (2006)
22. Shahbaz, M., Li, K., Groz, R.: Learning and integration of parameterized components through testing. In: Petrenko, A., Veanes, M., Tretmans, J., Grieskamp, W. (eds.) TestCom/FATES 2007. LNCS, vol. 4581, pp. 319–334. Springer, Heidelberg (2007)
23. Kohavi, Z.: Switching and Finite Automata Theory: Computer Science Series. McGraw-Hill Higher Education (1990)