Verification of Data-Intensive Embedded Systems

Massimo Narizzano and Armando Tacchella

KEYWORDS

Safety Evaluation, Dependable Systems, Embedded Systems, Formal Verification.

ABSTRACT

Verification of embedded software relying on black-box hardware is challenging whenever precise specifications of the underlying systems are incomplete or not available. Learning structured hardware models is a powerful enabler of verification in these cases, but it can be inefficient when the system to be learned is data-intensive rather than control-intensive. We contribute a methodology to attack this problem based on a specific class of automata which are well suited to model systems wherein data paths are known to be decoupled from control paths. We show the effectiveness of our approach by combining learning and verification to assess the correctness of embedded programs relying on FIFO register circuitry to control an elevator system.

INTRODUCTION

It is a known fact that our reliance on the functioning of information-and-communication systems is growing rapidly. Today, for the most part, they pervade our lives in the form of embedded systems such as mobile phones, vehicle control units, and TV sets. It is expected that in the future all sorts of objects of daily use might be equipped with some computing capability, including personal medical devices, home appliances, and even clothes [AM00]. Absence of errors, as well as safety and security guarantees, are thus crucial in embedded systems while their reliable operation is already of large social importance. In this direction, formal methods are one of the key technologies expected to improve the quality of our embedded system designs [HS06].

The main hurdle on the path towards adoption of formal methos is that formal specifications are notoriously hard to come by. While several reasons contribute to this state of affairs, the consequence is that many systems in use today lack adequate specifications or make use of under-specified components [HS14]. In practice, this situation is all but infrequent in embedded systems where third-party hardware components are used as parts, but only their interface and some informal description about their behavior is available. To overcome the problem posed by such *black-box* components, several authors considered automata learning techniques — see, e.g., [PVY99], [GPY06], [Sha08] — to obtain precise models through controlled experimentation. In spite of many success stories, and the availability of effective tools like LEARNLIB [RSBM09], learning components which cannot be modeled as having finite-size input alphabets is still challenging [HIS+12]. This is a problem when dealing with data-intensive, rather than control-intensive (sub)systems, because Angluin-based [Ang87] methods are not well suited for such systems.

In this paper we consider the problem of verifying embedded software relying on black-box hardware components. Such components are assumed to be data-intensive, i.e., to be useful, the characterization of their behavior must take into account data exchanged between the components and their embedding context. The practical usage scenario is that of an embedded program developed on top a third-party computing platform, where a precise specification of the facilities offered by the platform is not available. With respect to other works in the literature [HIS+12], [Aar14] we make a simplifying, but realistic assumption, i.e., that the data path and the control path are *separable*. By this we mean that the specific values exchanged between the embedding context and the system will have no effect on the control path of the system. On the other hand, the actual values exchanged are important to assess the correctness of the component. For instance, in a serial data bus, the actual content of transmitted packets does not alter the transmission protocol, but it is expected that packets will be received in the same order they were transmitted. The separability assumption still allows us to deal with interesting components — e.g., different kind of registers, memories, and data buses — while avoiding the theoretical and practical intricacies connected to more expressive models.

Several researchers addressed the problem of finding models for black-box hardware circuits. Most of these works aim to find a specification in the form of invariants — see, e.g., the GoldMine tool [She11] — or, more generally, in the form of temporal properties — see, e.g., the IODINE tool [HNCC05]. There are also some previous efforts devoted to learning hardware circuits as finite state machines [MKK14]. In [AR12], authors proposed to abstract the I/O functionality of continuous-time dynamical systems as Finite State Machines and to infer them using Angluin's [Ang87] algorithm. Our approach differs both from [MKK14], in that we do not wish to disregard entirely the data component, and from [AR12], in that we are interested in high-level behavioral simulations rather than accurate gate-level representations. There are works in the literature which consider automata learning as an enabler for formal verification of black-box systems. The pioneering work in this domain is *Black-box checking*

Massimo Narizzano and Armando Tacchella are with Università degli Studi di Genova, DIBRIS (Department of Informatics, Bioengineering, Robotics and Systems Engineering), Viale Causa 13, 16145 Genova. E-mail: massimo.narizzano@unige.it, armando.tacchella@unige.it. Both authors contributed equally to the paper. The corresponding author is Armando Tacchella.

Communications of the ECMS, Volume 37, Issue 1,
Proceedings, ©ECMS Enrico Vicario, Romeo Bandinelli,
Virginia Fani, Michele Mastroianni (Editors) 2023
ISBN: 978-3-937436-80-7/978-3-937436-79-1 (CD) ISSN 2522-2414

(BBC) [PVY99] which challenges the problem of verifying black-box systems through model checking. *Adaptive model checking* (AMC) [GPY06] is an extension of BBC where it is assumed that some model of the system to be verified exists, but such model might be inaccurate or partially obsolete. The main idea behind AMC is that initializing the learning algorithm with existing information improves on the performances of the whole verification process when compared to the BBC approach. While both BBC and AMC present some commonalities with our approach, the main difference is that in both BBC and AMC the system to be verified is the same of the system to be learned and, to some extent, verification and learning are intertwined. In our case, learning is performed only on the black-box hardware, whereas verification involves both the program relying on such hardware and the models thereof.

Summing up, our main contribution in this paper is to show that learning (models of) components whose data and control paths are separable is not hindering the ability to perform model checking on the whole system including such components. In more details, we consider the following steps:

- Learning the model of a FIFO register by interacting with a VHDL simulator using our tool AIDE [KT] (Automata IDentification Engine).
- Encoding of models obtained by AIDE into the language of the model checker SPIN [Hol97]; albeit the example we show is specific to the FIFO register, the econding can be applied to every kind of model learned by AIDE.
- Verifying an elevator control system built around the FIFO register using SPIN; the model of the control system is based on previous contributions by Nagafuji and Yamaguchi [NY14] and Attie et al. [ALPC06].

Our experiments show that learning models from black-box hardware parts is an effective path to increase reliance in the system as a whole. At least in our experience, the scalability challenge is still mainly on the side of model checking, i.e., AIDE can learn models larger than those verifiable with SPIN.

The remainder of this paper is organized as follows. In the "Background" section we introduce basic definitions and terminology about automata learning and model checking. In the "Learning and Verification" section we describe the elevator control system case study, including the properties that we wish to verify, and the encoding of learned automata for SPIN. In section "Experimental Analysis" we present experimental data related to the learning phase with AIDE, as well as the verification phase with SPIN. We conclude the paper in with some final remarks and a tentative agenda for future research.

BACKGROUND

A. Learning models of black-box components

Automata learning — also known as automata-based identification or grammatical inference — is a set of techniques that enables the inference of formal models of systems considering examples of their execution. Automata learning can be divided into two wide categories, i.e., passive and active learning. In passive learning, there is no control over the observations received to learn the model, whereas in active learning, the target system can be experimented with, and experimental results are collected to learn a model. In this paper, we focus on the latter kind of techniques, thereby assuming that the system under learning (SUL) is always available for controlled experimentation. Since we are interested in systems in which there is a clear separation between user-provided inputs and system-generated output, we consider Mealy machines as reference models for black-box systems. Active learning of Mealy machines was first developed by Niese [Nie03] and it was further extended by Shahbaz's [Sha08] L_M^+ algorithm — for more details, see [SHM11]. However, since modeling real-life systems often requires a finite number of interaction primitives (methods, operations, commands, protocol messages), but actual interactions often carry additional data values (parameters, resource identifiers, authentication credentials), standard Mealy machines might be not expressive enough. *Register Mealy Machines* (RMMs) are an extension that equips the structural skeleton of Mealy machines with a finite set of registers. The increase in expressiveness of RMMs makes learning such models intrinsically more complex. In [Aar14] an approach based on counterexample guided abstraction refinement (CEGAR) is proposed to construct models of black-box RMMs. This approach is implemented in a tool called Tomte[1] which, together with the learners provided by learnLib [RSBM09], enables identification of RMMs. Another approach for inference of RMMs is presented in [HIS+12], where a dedicated learning algorithm is proposed. At the time of this writing, an implementation of this approach is made available in the learnLib public repository.

While RMMs could fit our purposes, our case study requires learning models of black-box systems wherein the control component is independent from the data component. This is a substantial simplification over RMMs, one that allows for simpler learning algorithms and a more efficient identification process with respect to RMMs. To reap these benefits, we have introduced *Parametrized Mealy Machines* (PMMs), a restricted class of RMMs, together with their inference algorithm. In the following, we briefly describe PMMs to the extent required to understand the case study and the experiments in this paper. For lack of space, we do not describe the inference algorithm for PMMs which turns out to be a relatively straightforward extension of Shahbaz's [Sha08] L_M^+ algorithm to infer Mealy automata. The algorithm is implemented in our tool AIDE[2] which features learners for several classes of deterministic and non-deterministic models of computation — see [KT14] for details. PMMs are defined assuming an unbounded domain D of data values, a finite set of input symbols Σ_I, and a finite set of output

[1] `http://tomte.cs.ru.nl/`.
[2] `https://aide.codeplex.com/`.

symbols Σ_O where each input or output symbol is parameterized and takes a single formal parameter from D.[3] The set Σ_I (Σ_O) is called the input (output) alphabet of the machine. Let further $X = \{x_1, \ldots, x_m\}$ be a finite set of *registers*. An *assignment* is a partial mapping $\rho : X \to X \cup \Sigma_I$. A *Parametrized Mealy Machine* (PMM) is a tuple $(Q, q_0, \Sigma_I, \Sigma_O, D, X, \tau)$ where Q is a finite set of *locations*; $q_0 \in Q$ is the *initial location*; Σ_I and Σ_O are the finite sets of *parametrized input* and *output symbols*, respectively; D is the *data domain* of input and output parameters; X is the set of *registers*; and τ is a finite set of *transitions* in the form $\langle q, q', (i, d_i), (o, d_o), \rho\rangle$ where q and q' are the source and destination locations of the transition, $i \in \Sigma_I$ is the input symbol, $o \in \Sigma_O$ is the output symbol, $d_i \in D$ is the user-provided input data, $d_o \in D$ is the generated output data, and ρ is an assignment.

To characterize the semantics of PMMs, we first define a *valuation* as a partial mapping $\nu : X \to D$ which determines the values of *active* registers. A *state* is a pair (q, ν) where q is allocation and ν is a valuation. The initial state of the machine is always $(q, \emptyset)$, i.e., it has an empty valuation and no register is active. One *step* of a PMM takes it from state (q, ν) to state (q', ν') by input (i, d) and emits the output (o, d') if there is a transition $\langle q, q', (i, d), (o, d'), \rho\rangle$ such that ν' is the updated valuation, where $\nu'(x_j) = \nu(x_k)$ whenever $\rho(x_j) = x_k$ and $\nu'(x_j) = d$ whenever $\rho(x_j) = i$. In each step, (i) an input i with its parameter d is given to the machine, (ii) the machine may assign the value of the input parameter d to one of its registers x_j, provided that $\rho(x_j) = i$, (iii) registers may be copied, if there is some j, k such that $\rho(x_j) = x_k$, (iv) an output action o with its parameter d' is generated, and finally (v) the current location of machine changes from q to q'. Notice that in (ii) the PMM may change the value of a register, and in (iii) active registers may change and/or their values can be copied. The generated output parameter d' may come from a register or it can be some constant in D. Similarly to RMMs, the *execution* of machine is defined as a finite alternating sequence of states and steps $u_0, s_0, u_1, ..., u_n$ such that u_i is a concrete state and s_i is one step for all $i < n$.

B. Model checking system properties

Model checking — see, e.g., [BK08] — is a prominent formal verification technique for assessing functional properties of information and communication systems. The prerequisites of model checking are (i) a model of the system under consideration and (ii) a list of properties that the system must fulfill expressed in some formal logic. While not essential in theory, the availability of a system that can automate the check is taken for granted in practical applications. The task of such system is to perform an exploration of the state space of the system, until either a violation of the stated property is found, or no more new states can be explored.

[3]Notice that we consider only input and output symbols of arity one. This can be extended for arbitrary, but fixed a priori, number of parameters.

While there are various tools that support model checking for a variety of modelling and property-specification languages, they can be divided into two broad categories, namely explicit-state and symbolic-state model checkers. The former category encompasses tools that maintain the set of explored states using an explicit data structure, i.e., one in which the main elements stored are descriptions of the explored states. The latter category encompasses tools that represent the set of explored states as a logical formula on state variables, such that the formula is satisfied only when the variables are evaluated to explored states. The details of model checking algorithms are beyond the scope of this paper — see,e.g., [CGP99] for further details. Here it is sufficient to say that the crucial problem is that, while many model checking algorithms are polynomial in the size of the state space, the state space size is huge for all but the simplest models. At present, this is the main limit for the applicability of model checking techniques, which makes scalability the main parameter of evaluation in our experimental analysis.

In our experiments, we use the explicit-state model checker SPIN [Hol97] to evaluate correctness of properties. The reason of our choice is that SPIN is a mature and well-maintained tool which has been successfully deployed to verify a wide variety of industrial-size applications, from operating systems software and communications protocols to railway signaling systems [Hol97]. The modeling language of SPIN is PROMELA (PROcess MEta LAnguage), a formalism to describe communicating finite-state machines. The basic building blocks of PROMELA are *Process*, *Data Objects* and *Message Channels*. A *Process* defines the behaviour of a (sub)system, and it is defined by the keyword `Proctype` followed by the process name, the list of input parameters, and the body of the process which consists of data declarations and statements. *Data objects* are declared in a C-language style. Finally, *Message Channels* admit two operations, *send* and *receive*, where each channel has associated a *message* type, and only messages of that type can be sent and received. The channel declaration allows for the specification of a capacity. When the capacity is zero, the channel implements a *rendezvous* communication, i.e., the sender cannot proceed unless the receiver reads a message, and simmetrically, the reader cannot proceed unless the sender sends a message. When the capacity is at least one, and unless the buffer is either full or empty, the reader and the sender can process messages without the need of synchronizing. This is called a *buffered* channel in PROMELA.

Correctness properties expressing requirements about the behavior of a stystem are specified using Linear Temporal Logic (LTL) in SPIN. The language of LTL can be defined as follows. Given a set *Prop* of *propositional letters*, the set $\{\neg, \vee, \wedge, \leftrightarrow\}$ of *propositional connectives*, the set $\{\top, \bot\}$ of *propositional constants* and the set $\{\mathcal{X}, \mathcal{U}\}$ of *modal connectives*, the set *Form* of *formulas* is defined as the smallest set such that $\top \in$ *Form*, $\bot \in$ *Form*; if $p \in$ *Prop* then $p \in$ *Form*;

if $\alpha \in \mathit{Form}$ then $\neg\alpha \in \mathit{Form}$; if $\alpha \in \mathit{Form}$ then $\mathcal{X}\alpha \in \mathit{Form}$; if $\alpha, \beta \in \mathit{Form}$ then $(\alpha \odot \beta) \in \mathit{Form}$ where $\odot \in \{\vee, \wedge, \leftrightarrow, \mathcal{U}\}$. A formula is interpreted over *computations*. A computation $\pi : \mathbf{N} \to \mathbf{2}^{\mathit{Prop}}$ is a function which assigns truth values to the elements of *Prop* at each time instant (natural number). For a computation π and a point $i \in \mathbf{N}$, we have that: $\pi, i \not\models \bot$ and $\pi, i \models \top$; $\pi, i \models p$ for $p \in \mathit{Prop}$ iff $p \in \pi(i)$; $\pi, i \models \neg\alpha$ iff $\pi, i \not\models \alpha$; $\pi, i \models \mathcal{X}\alpha$ iff $\pi, i+1 \models \alpha$; $\pi, i \models (\alpha \odot \beta)$ is interpreted in the usual way for Boolean connectives, whereas for the modal connective $\mathcal{U}$ ("until") the semantics is that $\pi, i \models \alpha\,\mathcal{U}\,\beta$ iff for some $j \geq i$, we have $\pi, j \models \beta$ and for all k, $i \leq k < j$ we have $\pi, k \models \alpha$. We say that π *satisfies* a formula φ, denoted $\pi \models \varphi$, iff $\pi, 0 \models \varphi$. We see the formula $\mathcal{F}\alpha$ ("eventually" α) as an abbreviation of $\top\,\mathcal{U}\,\alpha$, and the formula $\mathcal{G}\alpha$ ("globally" α) as an abbreviation of $\neg\mathcal{F}\neg\alpha$. Intuitively, the task of SPIN is to check whether a given requirement φ is such that $\pi \models \varphi$ for every possible computation of the system model described in PROMELA. If this is not the case, then SPIN should exhibit the computation π^* (the "counterexample") such that $\pi^* \not\models \varphi$.

LEARNING AND VERIFICATION

A. Elevator system case study

The behavior of the system under consideration is schematized in Figure 1. The system is composed by N elevators moving through M different floors. On each floor, users have access to a number pad from which requests to go to specific floors can be made. Each request is queued and then it is served by the first available elevator. From an implementation point of view, we assume that the control system is implemented as a program whose source code is available to us, whereas the FIFO register used to queue users' requests is available as a part of an off-the-shelf emebedded computing platform whereon the control system runs. For this reason, a verification of the whole system is not possible unless a precise model of the FIFO queue is made available by the vendor of the computing platform. Practice tells us that it is unlikely that vendors supply such models, therefore we assume that a model of the FIFO queue must be inferred by experimenting with the system. In the following, we briefly describe the PROMELA model of the control system and the requirements it should satisfy as LTL formulas. In the next subsection we describe the learned model of the black-box FIFO queue together with its encoding in PROMELA.

The model of the system presented in Figure 1, is composed by three submodels, namely number pad, user requests queue and elevator. In more details:

- A number pad is a PROMELA process which randomly generates a request — thereby simulating user input to the system — and adds it to the user requests queue (see below). As a convention, each number pad process is named as $\mathit{Pad}[f]$, where f is the floor associated to the number pad. For each floor, there is only one internal variable storing the request made by the user which can be accessed with the syntax *f.request*.
- The user requests queue maintains data about user requests and corresponds to the black-box FIFO register available in the embedded computing platform. The process model is inferred as described in the next subsection. Here we just mention that the only two operations supported by the user request queue are *push* and *pop* primitives with the usual semantics.
- Each elevator is also a PROMELA process named as $\mathit{Elevator}[e]$, where e is the unique identification number of the elevator. An elevator has three main state variables, namely *floor* representing the current floor of the elevator, *state* indicating if the elevator is moving up (UP) , moving down (DOWN) or stopping (STOP); finally, *request* stores the floors that must served and has the value *NONE* if no request has to be served. The internal state of an elevator e can be accessed using the syntax *e.<variable>* where *<variable>* is one of *floor*, *state* and *request*.

An elevator system like the one described above should fulfill a number of safety and liveness requirements. Considering the literature [ALPC06], [NY14], we were able to find a number of typical constraints that we describe next. The motors, due to physical constraints, cannot be switched from going down (up) to going up (down) without stopping first. This requirement can be translated in LTL as

$$\begin{array}{l}\mathcal{G}((e.\mathit{state} = \mathit{UP}) \rightarrow \\ ((e.\mathit{state} = \mathit{UP})\,\mathcal{U}\,(e.\mathit{state} = \mathit{STOP})))\end{array} \quad (1)$$

where e is a generic elevator, $\alpha \rightarrow \beta$ is an abbreviation for $\neg\alpha \vee \beta$ and $A = B$ is the usual Boolean equality predicate — which is predefined in PROMELA. The same property must be instantiated for the DOWN state, and both properties must be checked for each elevator of the system. Each user request should always be satisfied — a typical *liveness* property. In order to make the check easier, we split the property in two parts. Firstly, we require that a user request is always accepted by at least an elevator. This corresponds to the LTL formula

$$\mathcal{G}((f.\mathit{request} = r) \rightarrow \mathcal{F}(\textstyle\bigvee_{i=1}^{N} e_i.\mathit{request} = r)) \quad (2)$$

where $\bigvee_{i=1}^{N} \alpha_i$ stands for $\alpha_1 \vee \ldots \vee \alpha_N$, and N is the number of elevators. Secondly, we require that a request accepted by an elevator is always satisfied with the following constraint:

$$\begin{array}{l}\mathcal{G}((e.\mathit{request} = \langle l, d\rangle) \rightarrow \\ ((e.\mathit{request} = \langle l, d\rangle)\,\mathcal{U} \\ \quad (e.\mathit{floor} = l \wedge e.\mathit{state} = \mathit{STOP})))\end{array} \quad (3)$$

where $\langle l, d\rangle$ is a generic request with load floor l and delivery floor d. Both properties (2) and (3) should be checked for each possible request. Notice that property (3) states that if an elevator e accepts a request $\langle l, d\rangle$ then it has to stop at floor l before serving another request, but it does not constrain the elevator to go to floor d afterwards. Finally, an obvious requirement is that no elevator tries to go beyond the top floor or

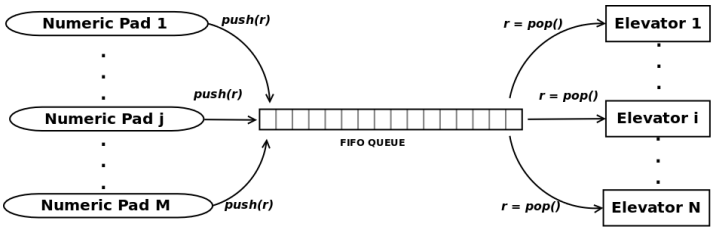


Fig. 1. Elevators System

below the ground floor. The LTL formulas for these two properties are:

$$\mathcal{G}((e.floor = (M-1)) \rightarrow \\ ((e.floor = (M-1))\,\mathcal{U}\,(e.state = \mathcal{STOP}))) \quad (4)$$

and

$$\mathcal{G}((e.floor = 0) \rightarrow \\ ((e.floor = 0)\,\mathcal{U}\,(e.state = \mathcal{STOP}))) \quad (5)$$

where e is a generic elevator, M is the number of floors and 0 is the ground floor number. Also these properties should be checked for each elevator.

B. Encoding of user request queue

A model for the user request queue learned by AIDE with the PMM inference algorithm is shown in Figure 2 (left). In this case we have assumed that the queue has 3 elements at most, which correspond to three PMM registers R0, R1 and R2. The system has only two interaction primitives (input symbols), namely PUSH and POP. The identified system has a total of four states and **0** is the initial state. Every transition is labeled as "$i/o/r$" where i is the concrete input symbol, o is the output symbol and r are the register operations. For instance from state **0** to state **1** the action "PUSH, d / NONE / R0 = d" means that data item d is pushed on the queue, no output is given and the data item is stored in R0. Notice that subsequent PUSH operations use registers in increasing order, and corresponding POP operations "shift" the registers to maintain queue ordering.

In Figure 2 (right) we show the encoding of the PMM in Figure 2 (left) into PROMELA. The encoding procedure is standard and works for any model inferred by AIDE. In particular every PMM is translated into a PROMELA process with two input channels, namely *inCH* and *outCH*, both with capacity 0. In the learned FIFO model, *inCH* is used to comunicate PUSH and POP operations from some external process, and *outCH* is used to return the output of the request, i.e., the first request to be served in case of a POP request. The translation also caters for some local variables, namely "S", "d" and an array "R". Variable "S" is used to store the current state of the PMM, variable "d" is used as a temporary storage for incoming data, and array "R" corresponds to the registers. In the learned FIFO model, "S" takes values in $\{0, 1, 2, 3\}$, corresponding to the states of the PMM in Figure 2 (left) and "R" is an array of three elements

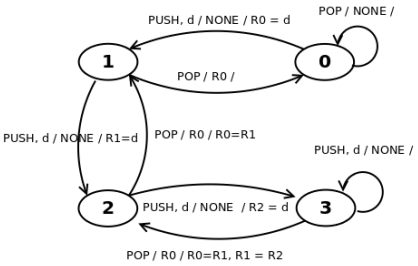
Size of queue	OQ	EQ	Time (s)
5	1768	2	46
10	11406	4	150
15	43496	6	466
20	106650	7	1257

TABLE I: Running time of AIDE to learn a model of the user request queue.

— indexed from 0 to 2 — corresponding to the three registers of the PMM. The main body of the process corresponding to a PMM is just a loop which updates the state according to the current state and the input channel value, thereby implementing the PMM computation semantics. For instance, the transition from state 2 to state 3 in Figure 2 (left) is coded into lines 5-8 in Figure 2 (right): when a $PUSH, d$ is received as input, the guard at line 5 becomes true, the result of the operation — NONE in this case — is given as output (line 6), then the value of the input is stored into the first empty register (line 7), and the state is updated (line 8). It is easy to see how the example given in Figure 2 can be generalized to the same FIFO model with a different number of places in the queue, and it is also straighforward to see how the construction is apt to simulate any PMM learned by AIDE.

EXPERIMENTAL ANALYSIS

All the experiments reported in this section ran on an Intel i7 3.4GHz PC equipped with 32GB of RAM and running Ubuntu 14.04. The inference of the FIFO queue models is performed by AIDE using a VHDL simulator loaded with an industrial-grade hardware system design. The system originally caters for a FIFO queue with 10 positions, but it is easy to modify the design in order to increase/decrease the size of the queue and thus evaluate the scalability of learning. The running time spent by AIDE to learn FIFO queues of various sizes is presented in Table I. Here we report the size of the queue, the number of output queries (OQ) and equivalence queries (EQ) performed by AIDE, as well as the total runtime (in CPU seconds). Output queries correspond to experiments in which AIDE asks the VHDL simulator to provide output on a specific input. The answer is used by AIDE to construct a conjecture about the structure of the SUL. Equivalence queries correspond to sets of experiments in which AIDE tries to understand whether its current conjecture could be a model of the SUL or not. As it could be expected, by increasing the number of positions, the number of




```
1:  proctype PMM(chan inCH, outCH) {
2:    int d,S;
3:    int R[3];
4:    do
5:      :: (S < 3) && (inCH ? PUSH , d) →
6:           outCH ! NONE;
7:           R[S] = d;
8:           S = S + 1;
9:      :: (S > 0) && (inCH ? POP , d) →
10:          outCH ! R[0];
11:          R[0] = R[1];
12:          R[1] = R[2];
13:          S = S - 1;
14:     :: (S == 3) && (inCH ? PUSH , d) →
15:          outCH ! NONE;
16:     :: (S == 0) && (inCH ? POP , d) →
17:          outCH ! NONE;
18:   od
```

Fig. 2. FIFO queue model as learned by AIDE (left); translation into a PROMELA Process (right)

output queries and the number of steps to obtain the right conjecture increases. Indeed, the PMM inference algorithm built in AIDE is able to learn FIFO registers of up to 20 places in about 20 minutes of CPU time. As we will see in the following, verification turns out to be unfeasible already for much smaller queue sizes.

In the verification experiments we consider two stages. In the first stage, we compile the PROMELA code without any optimization technique, while in the second stage we use the flags `-DCOLLAPSE`, `-DMA=n`, where `n` is suggested by SPIN after the first stage. Since property (2) is a liveness property we also enforce (*weak*) fairness conditions to verify it by adding `-DNFAIR=n` as a flag, where `n` is the number of processes fired in this case. We consider different instances of the elevator system — including the model inferred by AIDE — varying the number P of places in the queue with $P \in \{3, 5, 10\}$, the number E of elevators with $E \in \{1, 2, 3\}$ and the number F of floors with $F \in \{2, 3, 4, 5\}$, for a total of 27 different configurations. The results of verification for properties (1-5) on all the configurations are presented in Table 3, where each line of the table represents a different configuration of the elevator system. The columns in the table are divided into two parts: the first three columns ("System") represent the system configuration, while the second group of columns shows the verification results of each property, where Pi stands for property (i) in Section -B. For each property, we report the result (column "R") which is either "S" for a successfull verification (the property holds), "E" for an unsuccessfull verification (the property does not hold), and "M" stands for memory out; the column "T" reports the CPU time used by SPIN to verify the properties. Observing Table 3, we can see that most configurations with $P = 3$ can be verified by SPIN, whereas only about half of those with $P = 5$ are completed, and only the simplest one with one elevator and three floors can be handled for $P = 10$. Under this perspective, while the time spent for learning the FIFO model is not negligible with respect to the time spent for verification, we observe that (i) the learning time of AIDE for a user request queue with a given number of places is amortized over several configurations, and (ii) SPIN exhausts the main memory before completing the property check for all but the simplest configurations.

As a side remark, we notice that the only property that cannot be satisfied is (2), i.e., it is not guaranteed that a user request will be always served by an elevator. Inspecting the counterexample provided by SPIN we can see that this is the case because when the queue is full, further requests will be dropped, thus invalidating the property. As for the second stage, we did not report detailed results as they are not very different from those of the first stage. In particular, even allowing SPIN 2 hours of CPU time for each run, the number of system configurations that can be verified increases only by two, namely the ones that were not verified in stage one with 3 places in the FIFO.

CONCLUSIONS

Our experimental results, albeit restricted to learning a FIFO register for an elevator control systems, witness that PMMs are an effective model of hardware circuits wherein data and control paths are separable. Indeed, scalability in the verification of the whole design was limited by SPIN results before we could consider the largest FIFO register learned by AIDE as a component. Given these results, we think of extending our research along different directions. The first one is aimed to experiment with other components which can still be modeled as PMMs and try to improve on the verification part. In this direction, we could as well consider more complex models, like RMMs, and see if similar results can be obtained, or learning becomes less efficient than verification. A second direction is aimed to consider different model checkers, such for example the MCRL2 toolset [CGK+13], and see if they can increase the scalability of our approach. Finally, it would be interesting to try if a dynamic combination between learning and verification — possibly based on abstraction in the spirit of [Aar14] — may turn out to be more effective for systems like the one herewith considered.

References

[Aar14] Fides Aarts. *Tomte: Bridging the Gap between Active Learning and Real-World Systems*. PhD thesis, Radboud University Nijmegen, 2014.

[ALPC06] Paul C Attie, David H Lorenz, Aleksandra Portnova, and Hana Chockler. Behavioral compatibility without state explosion: Design and verification of a component-based elevator control system. In

System			P1		P2		P3		P4		P5	
#P	#E	#F	R	T(s)	R	T(s)	R	T(s)	R	T(s)	R	T(s)
3	1	3	S	1	E	1	S	1	S	1	S	0
3	1	4	S	1	E	1	S	1	S	1	S	1
3	1	5	S	1	E	0	S	1	S	1	S	1
3	2	3	S	2	E	1	S	2	S	1	S	1
3	2	4	S	6	E	1	S	11	S	3	S	3
3	2	5	S	22	E	1	S	54	S	11	S	11
3	3	3	S	179	E	2	S	263	S	81	S	81
3	3	4	M	61	E	4	M	58	M	62	M	61
3	3	5	M	65	E	2	M	60	M	64	M	64
5	1	3	S	1	E	1	S	1	S	1	S	1
5	1	4	S	1	E	1	S	2	S	1	S	1
5	1	5	S	4	E	1	S	11	S	2	S	3
5	2	3	S	11	E	1	S	16	S	6	S	6
5	2	4	S	110	E	1	S	214	S	51	S	52
5	2	5	M	57	E	1	M	52	M	57	M	56
5	3	3	M	54	E	3	M	55	M	58	M	58
5	3	4	M	58	E	3	M	57	M	63	M	62
5	3	5	M	62	E	4	M	56	M	67	M	66
10	1	3	S	38	E	1	S	62	S	19	S	20
10	1	4	M	44	E	0	M	44	M	48	M	47
10	1	5	M	45	E	1	M	45	M	49	M	49
10	2	3	M	47	E	1	M	47	M	47	M	46
10	2	4	M	47	E	1	M	48	M	47	M	47
10	2	5	M	50	E	1	M	46	M	50	M	49
10	3	3	M	49	E	6	M	46	M	49	M	48
10	3	4	M	51	E	5	M	48	M	52	M	51
10	3	5	M	54	E	12	M	51	M	55	M	54

Fig. 3. Result of verification (first stage).

Component-Based Software Engineering, pages 33–49. Springer, 2006.

[AM00] Gregory D Abowd and Elizabeth D Mynatt. Charting past, present, and future research in ubiquitous computing. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 7(1):29–58, 2000.

[Ang87] D. Angluin. Learning regular sets from queries and counterexamples. *Information and computation*, 75(2):87–106, 1987.

[AR12] Karthik V Aadithya and Jaijeet Roychowdhury. DAE2FSM: Automatic generation of accurate discrete-time logical abstractions for continuous-time circuit dynamics. In *Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE*, pages 311–316. IEEE, 2012.

[BK08] Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT press Cambridge, 2008.

[CGK+13] Sjoerd Cranen, Jan Friso Groote, Jeroen JA Keiren, Frank PM Stappers, Erik P de Vink, Wieger Wesselink, and Tim AC Willemse. An overview of the mcrl2 toolset and its recent advances. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 199–213. Springer, 2013.

[CGP99] Edmund M Clarke, Orna Grumberg, and Doron Peled. *Model checking*. MIT press, 1999.

[GPY06] A. Groce, D. Peled, and M. Yannakakis. Adaptive model checking. *Logic Journal of IGPL*, 14(5):729–744, 2006.

[HIS+12] Falk Howar, Malte Isberner, Bernhard Steffen, Oliver Bauer, and Bengt Jonsson. Inferring semantic interfaces of data structures. In *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change*, pages 554–571. Springer, 2012.

[HNCC05] Sudheendra Hangal, Sridhar Narayanan, Naveen Chandra, and Sandeep Chakravorty. IODINE: a tool to automatically infer dynamic invariants for hardware designs. In *Design Automation Conference, 2005. Proceedings. 42nd*, pages 775–778. IEEE, 2005.

[Hol97] Gerard J Holzmann. The model checker Spin. *IEEE Transactions on software engineering*, 23(5):279–295, 1997.

[HS06] Thomas A Henzinger and Joseph Sifakis. The embedded systems design challenge. In *FM 2006: Formal Methods*, pages 1–15. Springer, 2006.

[HS14] Falk Howar and Bernhard Steffen. Learning Models for Verification and Testing — Special Track at ISoLA 2014 Track Introduction. In *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change*, pages 199–201. Springer, 2014.

[KT] Ali Khalili and Armando Tacchella. AIDE: Automata-Identification Engine. `http://aide.codeplex.com`.

[KT14] Ali Khalili and Armando Tacchella. Learning nondeterministic Mealy machines. In *Proceedings of the 12^{th} International Conference on Grammatical Inference (ICGI)*, pages 109–123, 2014.

[MKK14] Lucie Matuova, Jan Katil, and Zdenek Kotasek. Automatic construction of on-line checking circuits based on finite automata. In *Digital System Design (DSD), 2014 17th Euromicro Conference on*, pages 326–332. IEEE, 2014.

[Nie03] Oliver Niese. *An integrated approach to testing complex systems*. PhD thesis, Universität Dortmund, Dortmund, Germany, December 2003.

[NY14] Kazuya Nagafuji and Shingo Yamaguchi. Éclair: An elevator group controller model checking system based on s-ring and spin. In *Consumer Electronics (GCCE), 2014 IEEE 3rd Global Conference on*, pages 178–181. IEEE, 2014.

[PVY99] Doron Peled, Moshe Y Vardi, and Mihalis Yannakakis. Black box checking. In *Formal Methods for Protocol Engineering and Distributed Systems*, pages 225–240. Springer, 1999.

[RSBM09] H. Raffelt, B. Steffen, T. Berg, and T. Margaria. LearnLib: a framework for extrapolating behavioral models. *International Journal on Software Tools for Technology Transfer (STTT)*, 11(5):393–407, 2009.

[Sha08] M. Shahbaz. *Reverse Engineering Enhanced State Models of Black Box Software Components to Support Integration Testing*. PhD thesis, Institut Polytechnique de Grenoble, Grenoble, France, 2008.

[She11] David Sheridan. Goldmine: An integration of data mining and static analysis for automatic generation of hardware assertions. Master's thesis, University of Illinois at Urbana-Champaign, 2011.

[SHM11] Bernhard Steffen, Falk Howar, and Maik Merten. Introduction to active automata learning from a practical perspective. In *Formal Methods for Eternal Networked Software Systems*, pages 256–296. Springer, 2011.