A Systematic Approach for Interfacing Component-Based Software with an Active Automata Learning Tool

Dennis Hendriks[1,2(✉)] and Kousar Aslam[3]

[1] ESI (TNO), Eindhoven, The Netherlands
dennis.hendriks@tno.nl
[2] Radboud University, Nijmegen, The Netherlands
dennis.hendriks@ru.nl
[3] Vrije Universiteit, Amsterdam, The Netherlands
k.aslam@vu.nl

Abstract. Applying Model-Driven Engineering can improve development efficiency. But gaining such benefits for legacy software requires models, and creating them manually is both laborious and error prone. Active automata learning has the potential to make it cost-effective, but practitioners face practical challenges applying it to software components of industrial cyber-physical systems. To overcome these challenges, we present a framework to learn the behavior of component-based software with a client/server architecture, focusing on interfacing isolated component code with an active learning tool. An essential part of the framework is an interfacing protocol that provides a structured way of handling the (a)synchronous communications between the component and learning tool. Our main contribution is the systematic derivation of such interfacing protocols for component-based software, which we demonstrate on the software architecture of ASML, a leading company in developing lithography machines. Through several practical case studies we show that our semi-automatic approach enables setting up a learning environment to learn component behaviors within hours. The protocol's responsibilities and the way it handles different communication types apply to component-based software in general. Our framework could thus be adapted for companies with similar software architectures.

Keywords: Active automata learning · Component-based systems · Industrial application

D. Hendriks—This research is carried out as part of the Transposition project under the responsibility of ESI (TNO) in co-operation with ASML. The research activities are supported by the Netherlands Ministry of Economic Affairs and TKI-HTSM.
K. Aslam—This research was supported by the Eindhoven University of Technology and ASML Netherlands B.V., carried out as part of the IMPULS II project.

T. Margaria and B. Steffen (Eds.): ISoLA 2022, LNCS 13702, pp. 216–236, 2022.
https://doi.org/10.1007/978-3-031-19756-7_13

1 Introduction

Cyber-physical systems often employ a component-based software architecture, dividing the system into components that can be developed, tested and deployed independently [17]. Model-Driven Engineering (MDE) places models at the center of attention, allowing for early analysis of a component's software behavior and for implementations to be automatically generated [18]. But gaining such benefits for legacy software requires models, and manual modeling for legacy components is often laborious and error prone due to a lack of understanding of their current behavior, for instance caused by insufficient documentation and the original developers having long since left the company.

To facilitate a cost-effective transition to MDE, model learning can automatically infer first-order models to bootstrap a subsequent manual modeling effort. Passive state machine learning for instance infers models based on execution logs [9,10], but the resulting models are often incomplete due to logs covering only parts of the component's behavior. Active automata learning (AAL) on the other hand repeatedly queries the component to ultimately infer a model capturing the component's complete behavior. AAL was introduced in Dana Angluin's seminal work on the L* algorithm [2]. A comprehensive body of work extended upon this to, e.g., learn different types of models, improve scalability, and show its practical value [12].

However, practitioners face practical challenges applying AAL to software components of real-world industrial cyber-physical systems. In order for an AAL tool to send queries to a component and gauge its responses, the component must be isolated from its environment and subsequently connected to the learning tool. Existing case studies typically explicitly or implicitly explain their learning setup [5,6,16]. But, establishing such a learning setup is laborious. It can therefore pay off to use a generic setup that can be (re)configured for reuse. By analyzing existing interface descriptions the new configuration can even be automatically generated. This has been shown to be effective for web services [15].

But what is lacking is a systematic approach to connect software components operating under the client/server paradigm to a learning tool. Therefore, similar to what is already available for web services, we contribute a general, reusable and configurable framework, to quickly produce an AAL setup, specifically for component-based software with a client/server architecture. Through multiple case studies we show that our semi-automatic approach enables setting up a learning environment to learn (sub-)component behaviors within hours.

A particular challenge when interfacing with such components, is how to deal with their various types of (a)synchronous communications, especially when considering their dual roles as servers to their clients and as clients to their servers. For instance, a reply from a server is an input to the component, but it is only possible after a request from the component itself. Components may thus not be input-enabled, a practical requirement of various AAL algorithms.

If a component is not input-enabled, a learning purpose [1] can be placed between the learner and the component. A learning purpose is essentially a protocol model, which rejects inputs not allowed by the protocol, and forwards any

other inputs to the component, to learn the subset of a component's behavior satisfying the protocol. Therefore, an essential part of our framework is an *interfacing protocol* model, a learning purpose that provides a practical but structured way of handling the communications between the AAL tool and the component whose behavior is to be learned. This way, the System Under Learning (SUL), the isolated component code combined with the protocol, is input-enabled, provides a single output per input, and represents a finite Mealy machine, even if the isolated component code does not satisfy these properties.

Our main contribution is the systematic derivation of the interfacing protocol. As an example, we derive such a protocol for the software architecture of ASML, a leading company in developing lithography systems. However, the responsibilities of the interfacing protocol and the way we handle the different types of communication patterns apply to component-based software in general. Our work could therefore be used to similarly derive interfacing protocols and set up learning frameworks at other companies with similar software architectures.

The remainder of this paper is organized as follows. In Sect. 2 we briefly introduce component-based software architectures and AAL. Section 3 describes our general AAL framework. Section 4 contains our main contribution, as it introduces interfacing protocols and their responsibilities, and describes the systematic derivation of such a protocol for ASML's software architecture. We then apply our approach in practice to infer behavioral models of several software components in Sect. 5, before concluding in Sect. 6.

2 Background

Component-based software architectures are often employed by cyber-physical systems to divide the system into *components* that can be developed, tested and deployed independently [17]. A component can act as a *server* offering its functionality via *interfaces* to other components, its *clients*. Components may interact with multiple other components, its *environment*, acting as either client or server to the various components. Interactions typically involve calling *functions* from interfaces of other components, e.g., as remote procedure calls. Concurrent components can communicate *synchronously* and *asynchronously* [8]. With synchronous communication a client remains idle while awaiting the server's *response*, while asynchronous communication allows a client to perform other interactions while a server is processing the client's *request*.

Active automata learning (AAL) involves repeatedly querying a component to ultimately infer a model that captures the component's complete behavior. Given our context of component-based software, queries involve function calls. As for instance function calls from clients are naturally inputs, and their corresponding return values are then outputs, we infer Mealy machine models. AAL then involves a *learner* using *membership queries* (MQs), sending sequences of *inputs* (matching, e.g., function calls from clients) to the *System Under Learning* (SUL) and observing the *outputs* (e.g., function return values), using them to construct a *hypothesis* model for the SUL's unknown behavior. An *equivalence*

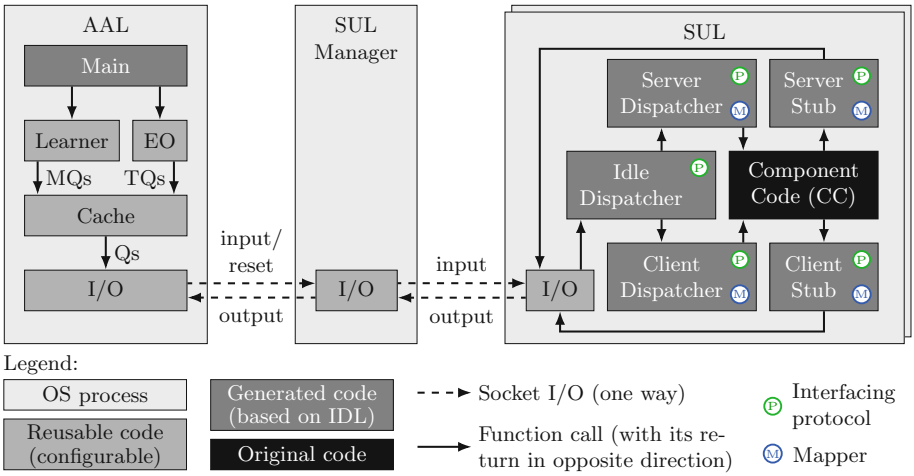


Fig. 1. General framework to perform AAL on software components operating under the client/server paradigm.

oracle (EO) then either confirms that the hypothesis matches the SUL's behavior or it produces a counterexample. The EO is typically implemented using conformance testing techniques, using *test queries* (TQs) to find differences in behavior between the hypothesis and the SUL. Iteratively, the learner uses the counterexample and further membership queries to refine its hypothesis, and the EO checks this refined hypothesis, until the EO considers a hypothesis correct. See also the left part of Fig. 1.

3 Active Automata Learning Framework

Complex cyber-physical systems may consist of millions of lines of code, spread out over many hundreds of components. Our goal is to infer the externally visible behavior of individual (sub-)components, e.g., to allow replacing the legacy implementation of their business logic. This requires that a component whose behavior is to be learned is isolated from its environment and is subsequently connected to the AAL tool. We assume single-threaded components. Figure 1 shows our general framework to perform AAL on software components operating under the client/server paradigm. The outermost boxes represent processes.

At the left is the *AAL* process. It consists of a *Main* function that configures a *Learner* and *EO*, and subsequently iteratively invokes them as described in Sect. 2. The *Learner* and *EO* produce *MQs* and *TQs*, respectively, which are handled identically by a *Cache*. The *Cache* caches and directly answers previously asked queries. For new queries, the *I/O* module sends the *input* symbols to a *SUL Manager* process via sockets, and similarly receives *output* symbols.

The *I/O* module of the *AAL* process sends after each *MQ*/*TQ* a *reset* symbol to the *SUL Manager*, informing it of query completion. The *SUL Manager* uses

a new *SUL* instance for every query to ensure that the *SUL* executes inputs from the same initial state. It manages a pool of *SUL* instances and concurrently processes *MQs*/*TQs* and spawns new *SUL* instances for better performance. The *SUL Manager* thus forwards inputs from the *AAL* process to a *SUL* instance, and outputs in the reverse direction. Using multiple processes allows a *SUL* to be implemented in a different programming language than the *AAL* process and eases spawning of new *SUL* instances and killing obsolete ones.

At the core of the *SUL* is the *Component Code* (*CC*) whose behavior is to be inferred. As part of introducing MDE, this code is to be replaced by code generated from models. The wrapper code that handles inter-process communications to other components via middleware, dealing with serialization and such, is not considered part of *CC*, as it will remain in place also for the newly generated code. Instead, only the code that implements the component's functionality is used. Dispatchers and stubs replace the wrapper code, similar to how code is isolated in the field of automated software testing.

When the *I/O* module receives an input, it forwards the input to the *Idle Dispatcher*, if the *SUL* is idle. A component can be both a server to its clients and a client to its servers. The *Idle Dispatcher* forwards inputs to the *Server Dispatcher* if the *CC* (and thus the SUL) acts as server for requests from clients, or to the *Client Dispatcher* if it acts as client to responses from servers, e.g., due to earlier asynchronous requests to servers. These *Client*/*Server Dispatchers* act as *mappers* [15] translating input symbols provided as strings to function calls on the *CC*. Once a function returns, its return value is mapped back to an output (string), which via the *Idle Dispatcher* and *I/O* module is provided back to the *SUL Manager* and *AAL* processes. When the *CC* communicates as a client to one of its servers, the call is intercepted in the *Client Stub*. This is also a mapper, forwarding the output symbol corresponding to the call to the *I/O* module. Upon receiving the server reply as *input* from the *I/O* module, the *Client Stub* translates this server reply to a return value to be returned back to the *CC*. Finally, the *Server Stub* similarly intercepts and handles calls from the *CC* (acting as server) to the client. In our work, the mappers do not apply any abstractions to the input alphabet.

The final aspect of our framework, the *interfacing protocol*, as implemented in the dispatchers and stubs, is further explained in Sect. 4. The various shades of gray used to color the boxes of Fig. 1 are explained in Sect. 5.

4 Interfacing Protocol

In this section, we discuss interfacing protocols and their systematic derivation, using the software architecture of ASML as an example. After briefly explaining the company's software architecture, we describe its communication patterns, and how they map to AAL inputs and outputs, before introducing the interfacing protocol and its responsibilities, and systematically deriving such a protocol from the patterns and the input/output mapping.

4.1 Software Architecture and Communication Patterns

The components interact through remote function calls. A call from a client is, by the middleware, placed in the server's message queue. A server is idle while awaiting incoming messages in its `main` function's message processing loop. As long as the queue is non-empty, it picks up messages from the queue and processes them one by one, non-preemptively.

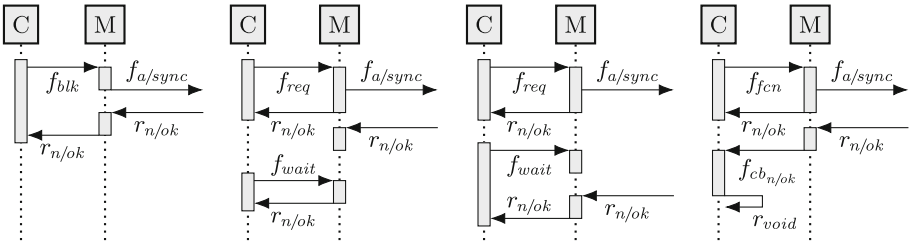
Components communicate with each other using various communication patterns. Clients can invoke functions of their servers in three ways: as *blocking* calls, *request/wait* calls, and *function completion notification (FCN)* calls. Servers can *handle* these request either *synchronously* or *asynchronously*. The middleware transparently hides these details, as clients are not aware of how servers handle their calls, nor are servers aware of how clients invoke the calls. We ignore *library* calls, which for the purpose of this work are identical to synchronously-handled blocking calls. The last two patterns are 'fire-and-forget' type of communications: *triggers* and *events*, which give no assurances of (successful) function execution, e.g., a call may fail without notifying the caller. Next, we describe the 7 patterns (3 client requests, 2 server handlers, 2 fire/forget) in more detail using the Message Sequence Charts (MSCs) from Fig. 2.

Here f denotes any *function*, t any *trigger* and e any *event*. Collectively, these three are called *methods*. A method is assumed to include the identity of the server that provides it, distinguishing methods from interfaces provided by multiple servers. The company uses a proprietary Interface Description Language (IDL) to define interfaces and their methods. A generator generates, from IDL files, implementation functions to call and handle all defined methods, for various programming languages. We distinguish IDL functions (*i-functions*) from generated functions (*g-functions*) where relevant. Any non-void g-function returns a value of type `ASML_RESULT`, an integer result indicating success (0) or failure (non-zero). The company's software architecture rule book states that callers must not use individual error codes, but only OK (0) vs not-OK (non-zero), except for logging, to prevent tight coupling between functions.

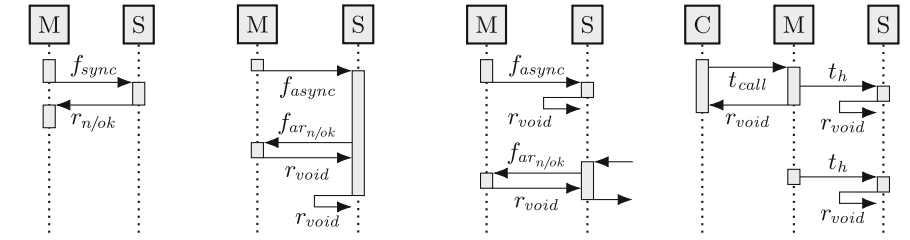
(i) Blocking Call (Figure 2a): A client may invoke an i-function f synchronously as a blocking call (with g-function f_{blk}). While awaiting the server's response (return value r_{ok} or r_{nok}), the client is then *blocked* and can not do any internal processing, perform calls, or process messages from its message queue.

(ii) Request/Wait Call (2b / 2c): A client may asynchronously *request* (f_{req}) a server to start executing an i-function f. The client is then free to do other things before explicitly *waiting* (f_{wait}) for the server's response (the second r_{ok} or r_{nok}). If the server has already finished executing the i-function, the middleware has stored the server's response and immediately returns this to the client (**2b**). Otherwise, the client is blocked until the server finishes and its response is provided back to the client via the middleware (**2c**). With request/wait calls the client is in control of when it is ready to receive the server's response.

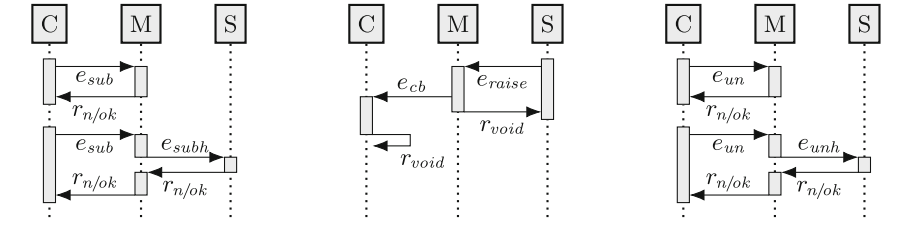
(iii) Function Completion Notification (FCN) Call (2d): A client may asynchronously request (f_{fcn}) a server to start executing an i-function f, providing it a callback address (f_{cb}). The client is then free to do other things. Once



(a) Blocking call (b) Request/wait call (server finishes first) (c) Request/wait call (client waits first) (d) FCN call



(e) Synchronous handler (f) Asynchronous handler (direct) (g) Asynchronous handler (delayed) (h) Trigger



(i) Event subscription (j) Event raise (k) Event unsubscription

Lifelines

C	Client
M	Middleware
S	Server

Methods

f	IDL function (i-function)
t	IDL trigger
e	IDL event

Messages for g-functions

f_{blk}	Blocking call for f
f_{req}	Request call for f
f_{wait}	Wait call for f
f_{fcn}	FCN call for f
f_{cb}	FCN callback for f
$f_{cb_{n/ok}}$	Either $f_{cb}(r_{ok})$ or $f_{cb}(r_{nok})$
f_{sync}	Synchronous handler for f
f_{async}	Asynchronous handler for f
$f_{a/sync}$	Either f_{sync} or f_{async}
f_{ar}	Asynchronous result call for f
$f_{ar_{n/ok}}$	Either $f_{ar}(r_{ok})$ or $f_{ar}(r_{nok})$
t_{call}	Call for t
t_h	Handler for t
e_{sub}	Subscribe call for e
e_{subh}	Subscribe handler for e
e_{raise}	Raise call for e
e_{cb}	Callback for e
e_{un}	Unsubscribe call for e
e_{unh}	Unsubscribe handler for e

Messages for return values

r_{void}	Return value for void g-function
r_{ok}	Return value for successful execution
r_{nok}	Return value for unsuccessful execution
$r_{n/ok}$	Either r_{ok} or r_{nok}

Fig. 2. Message Sequence Charts (MSCs) for all (partial) communication patterns.

the server finishes executing f, it provides its response (r_{ok} or r_{nok}) to the middleware, which places both the callback address and the server's response in the client's message queue. Once the client is idle it will process its message queue, eventually processing the server's response using callback g-function f_{cb}, i.e., f_{cb} is called with the queued server response as argument. The client is thus *notified* of the server *completing* the execution of *function* f, as the client requested. With FCN calls the server is in control of when it provides the response.

(iv) Synchronous Handler (2e): A server may *synchronously handle* all blocking calls, request/wait calls and FCN calls for an i-function f. Once the handler (f_{sync}) finishes, it immediately returns its response (r_{ok} or r_{nok}).

(v) Asynchronous Handler (2f / 2g): A server may also *asynchronously handle* all blocking calls, request/wait calls and FCN calls for an i-function f. The asynchronous handler (f_{async}) starts handling the request. In that handler (**2f**), or at any later time in any other g-function (**2g**), it sends its response to the client by calling an *asynchronous result* g-function, i.e., $f_{ar}(r_{ok})$ or $f_{ar}(r_{nok})$.

A client call pattern (**i** − **iii**) and server handler pattern (**iv** − **v**) are to be combined to form a complete pattern, with a client, middleware and a server.

(vi) Trigger (2h): A client may *trigger* a server (t_{call}), for a trigger t. The server handles (t_h) the trigger without responding back to the client (r_{void}). A server may also be triggered directly by the middleware, e.g., periodically.

(vii) Event (2i − **2k):** A client may *subscribe* (e_{sub}) to a specific *event* e of one of its servers (**2i**), providing a callback address (e_{cb}). The middleware stores the subscription. A server may optionally have a *subscription handler* (e_{subh}) to be notified of subscriptions. A server may *raise* (e_{raise}) an event (**2j**), which leads to callback g-functions (e_{cb}) being invoked (akin to FCN callbacks) on all clients subscribed to that server for the specific event. Clients may at any time *unsubscribe* (e_{un}) from events (**2k**), again optionally notifying the server (e_{unh}).

4.2 Mapping Communication Patterns to Inputs/Outputs

For each of the 7 (partial) communication patterns the various calls to g-functions and their replies can be mapped to inputs and outputs for AAL. Table 1 shows this mapping. It is constructed by considering the role of the *CC* (and thus of the SUL), as client and/or server, for each pattern from Fig. 2. As our goal is to infer a component's functional behavior, it is isolated from its environment, 'cutting off' (ignoring) the middleware. Only the incoming and outgoing messages from clients and servers are considered (C and S lifelines in the MSCs). Messages from a client to the SUL (acting as server) are inputs. Reverse communications are outputs. Conversely, messages from the SUL (acting as client) to a server are outputs. Reverse communications are inputs. The role of the SUL, as client or server, thus inverts whether its incoming and outgoing messages are inputs or outputs.

For instance, in Fig. 2a the SUL can only act as client. The outgoing f_{blk} message to a server is then an output, and the incoming r_{ok} or r_{nok} message from a server is an input.

Table 1. Communication pattern messages from Fig. 2 mapped to AAL inputs and outputs, for the SUL acting as client or server.

Communication pattern	Role of SUL	Message	Input/Output
(i) Blocking call	Client	f_{blk}	Output
		r_{ok} / r_{nok}	Input
(ii) Request/wait call	Client	f_{req}	Output
		r_{ok} / r_{nok}	Input
		f_{wait}	Output
		r_{ok} / r_{nok}	Input
(iii) FCN call	Client	f_{fcn}	Output
		r_{ok} / r_{nok}	Input
		$f_{cb_{ok}}$ / $f_{cb_{nok}}$	Input
		r_{void}	Output
(iv) Synchronous handler	Server	f_{sync}	Input
		r_{ok} / r_{nok}	Output
(v) Asynchronous handler	Server	f_{async}	Input
		r_{void}	Output
		$f_{ar_{ok}}$ / $f_{ar_{nok}}$	Output
		r_{void}	Input
(vi) Trigger	Client	t_{call}	Output
		r_{void}	Input
	Server	t_h	Input
		r_{void}	Output
(vii) Event	Client	e_{sub}	Output
		r_{ok} / r_{nok}	Input
		e_{cb}	Input
		r_{void}	Output
		e_{un}	Output
		r_{ok} / r_{nok}	Input
	Server	e_{subh}	Input
		r_{ok} / r_{nok}	Output
		e_{raise}	Output
		r_{void}	Input
		e_{unh}	Input
		r_{ok} / r_{nok}	Output

4.3 The Interfacing Protocol and Its Responsibilities

When applying AAL in practice, often several preconditions must hold, e.g., when using LearnLib [14] to learn Mealy machines, the SUL must be input-enabled.

If the *CC* does not satisfy such conditions, the interfacing protocol ensures that the SUL does satisfy them. The protocol is part of the SUL's dispatchers and stubs, see Fig. 1. All communications between the learner and the *CC* go through the protocol. Here, we discuss the protocol's three responsibilities. The next section explains *how* it satisfies them.

(a) Input-Enabled: For some learning tools/algorithms, the SUL must be input-enabled for the learner to query every input on the SUL for every state. This condition does not always hold, e.g., for FCN callbacks (Fig. 2d). Along with an FCN call (f_{fcn}) the SUL provides a callback address (f_{cb}). In the real system, the middleware places that callback in the client's message queue upon receiving the server's reply. For AAL, the client dispatcher invokes it directly. An FCN callback to the SUL is an input, which is thus only possible after an FCN call by the SUL. Without the call, the callback address is unknown and it can not be invoked on the *CC*. The SUL is then not input-enabled. The interfacing protocol detects such invalid inputs, for which it can not rely on the *CC*. Instead, the protocol itself replies to the learner, making the SUL as a whole input-enabled. As only impossible inputs are rejected, the complete *CC* behavior can still be learned for all valid inputs.

(b) Single Input, Single Output: Inferring Mealy machines with AAL requires that each input produces a single output. The interfacing protocol is designed to always alternate inputs and outputs. By matching g-function calls and their returns, this is a natural fit. It therefore does not prevent learning the full externally-observable behavior of *CC*.

The protocol is an Extended Finite State Machine (EFSM), while we infer Mealy machines. Each protocol input and following output matches a single Mealy machine transition.

(c) Finite Learning Result: Certain component behavior can not be captured as a finite Mealy machine, e.g., for multiple concurrent executions of an asynchronous handler (Fig. 2g). With a the handler's start (f_{async}), b its end (r_{void}), c a later successful asynchronous result call ($f_{ar_{ok}}$), and d its end (r_{void}), this may lead to sequences '...ab...cd...', '...$abab$....$cdcd$...', etc., and in general '...$(ab)^n$...$(cd)^n$...'. Unlike a pushdown automaton, a Mealy machine can not capture this in a finite manner. It would contain infinitely many paths, one for each value of $n \geq 0$. AAL would have to discover each path to infer a complete model, which would never terminate. The interfacing protocol can restrict such concurrent executions ($n \leq m$) to ensure that AAL terminates, at the expense of not learning the full behavior. Not only are higher concurrency variants then absent ($n > m$), any differences in behavior resulting from them would also be absent, e.g., the component's behavior could be different for $n = m + 1$ than for any $n \leq m$, but this would not be in the inferred model. However, the resulting models can still be valuable in practice, as our goal is automatically infer first-order models to bootstrap a subsequent manual modeling effort.

The interfacing protocol is a small wrapper around *CC*, addressing these three responsibilities. Neither ensuring input-enabledness, nor 'single input, single output', prevents learning the full externally-observable behavior of *CC*. Using interaction limit $m = \infty$, the full behavior of a *CC* can thus be learned, assuming *CC* has a finite Mealy machine representation. Otherwise, by restricting m, a subset of the behavior of *CC* (an under-approximation) can be learned. Assuming an AAL algorithm is used that ensures that learned models are minimal, our approach in no way impacts that guarantee.

4.4 Systematic Derivation of the Interfacing Protocol

The interfacing protocol provides a structured way of handling the various communication patterns, ensuring that the SUL is input-enabled, provides a single output per input, and represents a finite Mealy machine, even if the isolated *CC* does not satisfy these properties. We derive such an interfacing protocol in a systematic manner, still using the same example software architecture.

Figure 3 shows the interfacing protocol as an EFSM. The protocol starts in state *Idle* as the *CC* is initially idle, awaiting a call. The *Idle* state is an input state from the *SUL*'s perspective, shown in dark gray in the figure. Upon receiving a *SUL* input, the g-function matching that input is called on the *CC*. The protocol then transitions to the *Busy* state, which is an output state from the *SUL*'s perspective, denoted light gray in the figure. Here the *CC* continues executing the g-function. Upon its return, the protocol transitions back to the *Idle* state. Alternatively, while *Busy*, it may communicate to one of its servers and go to the *Blocked* state. There it is blocked until the call to the server returns, going back to the *Busy* state to continue the still-in-progress g-function execution. For simplicity, we do not consider calls of a component to its own provided interface. Next, we consider all 7 communication patterns from Fig. 2, with their associated inputs/outputs from Table 1, one at a time.

(iv) Synchronous Handler (Figure 2e): An idle SUL, acting as a server, can synchronously handle a call from one of its clients. Upon receiving f_{sync} for some i-function f as an input in the *Idle* state the protocol invokes the corresponding handler (g-function) on the *CC*. It also transitions to the *Busy* state, as the *CC* is then busy executing. Variable v_1 is updated to indicate the in-progress handler is not a void function (update $[v_1 := \bot]$). When the handler returns, depending on its return value (zero for successful execution, non-zero otherwise) the protocol produces an output (r_{ok} for zero, r_{nok} otherwise) and transitions back to the *Idle* state.

(i) Blocking Call (Figure 2a): The *CC*, while it is executing (state *Busy*), may execute a blocking call to one of its servers. If the *Client Stub* receives a blocking call for an i-function f, it maps that to output f_{blk} and transitions to *Blocked*. A blocking call is a non-void g-function ($[v_2 := \bot]$). In *Blocked* the SUL is blocked while waiting until it receives r_{ok} or r_{nok} as input. It then returns from the blocking call back to the *CC*, with return value 0 (for r_{ok}) or 1 (for r_{nok}), and transitions back to *Busy*.

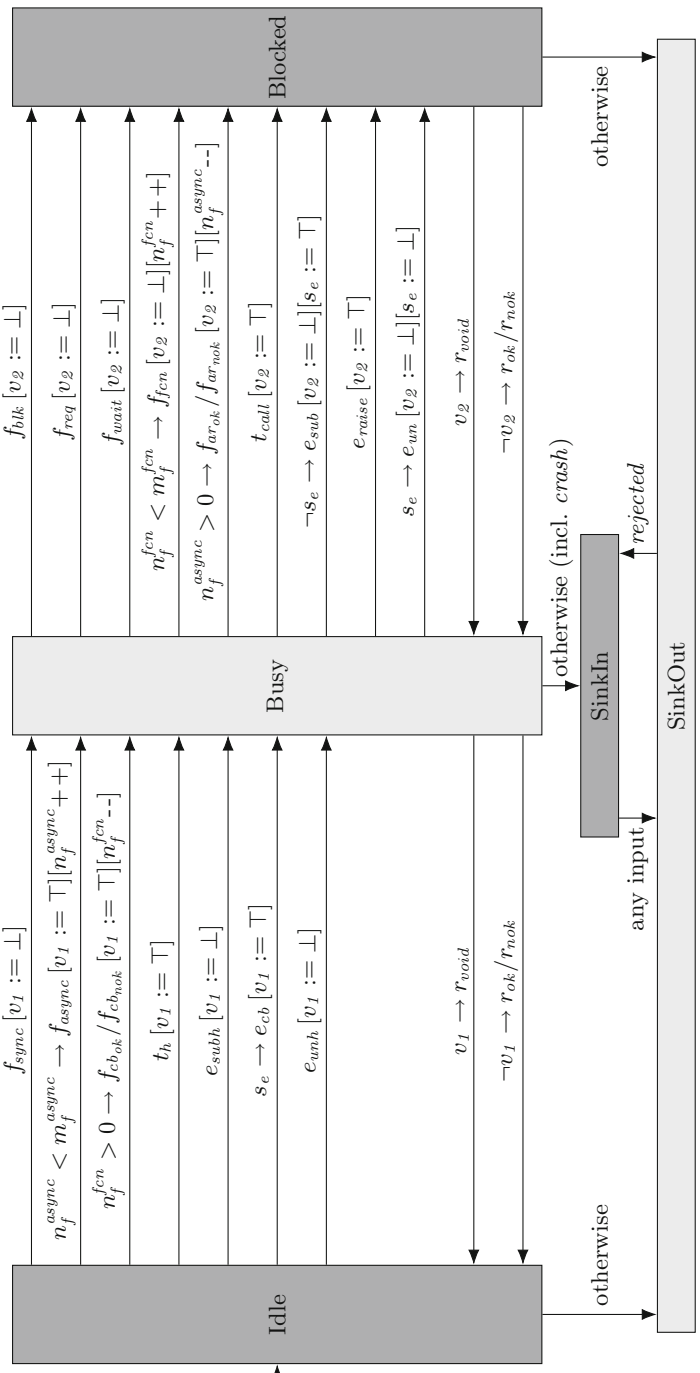


Fig. 3. Interfacing protocol EFSM for ASML's software architecture and communication patterns.

(ii) Request/Wait Call (Figures 2b / 2c): Similar to blocking calls, the *CC* may perform request/wait calls (f_{req} and f_{wait}), going from *Busy* to *Blocked* and back (r_{ok} or r_{nok}).

(iii) FCN Call (Figure 2d): The *CC* may also invoke an FCN call (f_{fcn}), going from *Busy* to *Blocked* and back (r_{ok} or r_{nok}). n_f^{fcn} is incremented by one ($[n_f^{fcn}{+}{+}]$) to indicate the FCN callback corresponding to this FCN call has not yet been handled. At a later time, when the SUL is *Idle*, it may handle the FCN callback, i.e., $f_{cb_{ok}}$ in case of success or $f_{cb_{nok}}$ upon failure of the FCN call. An FCN callback is a void g-function ($[v_1 := \top]$), and n_f^{fcn} is then decreased by one ($[n_f^{fcn}{-}{-}]$). For any i-function f, the protocol restricts the number of concurrently outstanding FCN calls (n_f^{fcn}) to at most m_f^{fcn}. Its callback ($f_{cb_{ok}}$ or $f_{cb_{nok}}$) is only possible if there is an outstanding FCN call (guard '$n_f^{fcn} > 0 \rightarrow$'). For simplicity, we ignore the use of FCN call timeouts.

(v) Asynchronous Handler (Figures 2f / 2g): Similar to synchronous handlers, the SUL may asynchronously handle calls from its clients (f_{async}). Such handlers are void g-functions ($[v_1 := \top]$). The *CC* may, during that handler or at any later time that it is *Busy*, invoke an asynchronous result g-function for this asynchronous handler ($f_{ar_{ok}}$ or $f_{ar_{nok}}$), which returns r_{void}. Variable $0 \leq n_f^{async} \leq m_f^{async}$ keep track of and restricts the number of concurrently outstanding asynchronous handler calls for i-function f.

(vi) Trigger (Figure 2h): While *Busy*, the SUL can trigger a server (t_{call}), returning void ($[v_2 := \top]$). While *Idle*, the SUL can handle a trigger from a client (t_h), also returning void ($[v_1 := \top]$).

(vii) Event (Figures 2i–2k): While *Busy*, a SUL may subscribe to an event of a server (e_{sub}), after which it is subscribed ($[s_e := \top]$). It can only do so if not yet subscribed to that event of that server ($\neg s_e \rightarrow$). Similarly, it may unsubscribe (e_{un}) if already subscribed ($s_e \rightarrow$) and is then no longer subscribed ($[s_e := \bot]$). While *Idle* and subscribed ($s_e \rightarrow$), it may process event callbacks (e_{cb}). Reversely, acting as a server to its clients, it may execute event (un)subscription handlers (e_{subh} and e_{unh}) while *Idle*, and raise events (e_{raise}) while *Busy*. For simplicity, we ignore the rare use of re-subscriptions.

States *Idle*, *Busy* and *Blocked*, and the transitions between them, support all 7 communication patterns, i.e., allow the interaction patterns modeled as MSCs in Fig. 2. Next, we explain how the protocol satisfies its responsibilities.

(a) Input-Enabled: Some inputs are impossible in certain input states. For instance, a t_h input is possible in state *Idle*, but not in state *Blocked*. Also, $f_{cb_{ok}}$ is only allowed in *Idle* if $n_f^{fcn} > 0$ holds. For all impossible inputs in input states, the interfacing protocol transitions to a sink state, where it keeps producing *rejected* outputs. That is, for invalid inputs it goes to the *SinkOut* output state. There it produces output *rejected*, goes to input sink state *SinkIn*, where it accepts any input, goes to *SinkOut*, produces *rejected* as output, etc. This turns a non-input-enabled *CC* into an input-enabled *SUL*, while preserving all its original externally-observable communication behavior.

(b) Single Input, Single Output: Each of the five protocol states is either an input state (dark gray) or output state (light gray). Input states have only outgoing transitions for inputs, and output states only for outputs. Transitions for inputs go to output states, while output transitions lead to input states. It intuitively matches the duality of g-function call starts and their returns. If the *CC* crashes in state *Busy*, the protocol produces a single *crash* output symbol and goes to *SinkIn*. This way the protocol ensures that each input is followed by a single output, and that they alternate. It also remains input-enabled, and still supports all communication patterns to allow inferring the full *CC* behavior.

(c) Finite Learning Result: To ensure that the SUL represents a finite Mealy machine, certain interactions can be limited. For instance, m_f^{async} limits the number of concurrently outstanding asynchronous handlers for i-function f. Starting with $m_f^{async} = \infty$, intermediate hypotheses may reveal it is necessary to restrict m_f^{async}. This ensures a finite SUL and learning result at the expense of potentially missing some component behavior. The protocol restricts both inputs (e.g., f_{async}) and outputs (e.g., $f_{ar_{ok}}$), redirecting them to sink states. The protocol in Fig. 3 limits only outstanding FCN calls and asynchronous handlers. Theoretically, similar issues could arise for other communication patterns. These can similarly be restricted, but this has been omitted in this paper to keep the protocol simpler, and because they rarely need to be restricted in practice. In particular, request/wait calls are not restricted as they involve only outputs, not inputs, and the company's software architecture rule book, to which all its software must adhere, already allows at most one concurrently outstanding request per i-function f.

The complete behavior of a *CC* can be learned, if it is finitely representable as a Mealy machine, by setting all interaction limits to ∞. Otherwise, by setting interaction limits, a subset of the *CC* behavior can be learned.

The validity of the interfacing protocol follows from its systematic derivation, providing correctness by construction. We do not provide a formal proof of the correctness of our approach, leaving this as future work.

4.5 Interfacing Protocol Optimization

In the interfacing protocol (Fig. 3) the *CC* may, while *Busy*, raise an event (e_{raise}). It is then *Blocked* until the event raise g-function returns (r_{void}). The part from Fig. 3 related to raising event is shown in Fig. 4a. While in state *Blocked* only one input (r_{void}) is allowed, the learner will try out all inputs, only to find out all of them get rejected, except for r_{void}. This holds any time an event is raised by the *CC*.

(a) Non-optimized (b) Optimized

Fig. 4. Optimization for raising event in the interfacing protocol.

This can be optimized as shown in Fig. 4b. Here the output (e_{raise}) and subsequent input (r_{void}) are combined into a single transition. To preserve the *single input, single output* property of the interface protocol, the e_{raise}, r_{void} transition from output state *Busy* is considered an 'extra output'. Upon executing such a self-loop, the protocol stores the extra outputs until the next 'real' output. It then prefixes the 'real' output with the extra outputs, in the order they were produced. The mapper, being part of the protocol, maps each of them to a string and combines them to form a single output. For instance, for two consecutive Mealy transitions i/e_{raise} and r_{void}/o, with i some input and o some output, the optimized result would be a single Mealy transition $i/e_{raise}, r_{void}, o$.

All void g-function outputs from *Busy* to *Blocked* allow for this optimization, i.e., $f_{ar_{ok}}$, $f_{ar_{nok}}$, t_{call}, and e_{raise}.

5 Application

We apply our approach to infer the behavior of two ASML software components: a high-level wafer exposure controller (we name it WEC), and a metrology driver (MD). However, these case studies are not a main contribution of our work, but rather examples to show the feasibility of applying our framework in practice, and to discuss the practicalities that it involves. Therefore, and for reasons of confidentiality, we do not describe them in more detail.

For each component, the AAL framework from Fig. 1 needs to be instantiated. The following steps were involved:

1. Framework Generation: For the component of interest, its interfaces must be identified, and their relevant code (g-functions) collected to use as *CC*. The company's proprietary generator takes IDL files with interface methods and automatically generates g-functions. We extended it to generate the *Main* function, dispatchers and stubs, including the interfacing protocol and mappers. The three *I/O* modules are hand-written and reusable. The *SUL*'s *I/O* module includes its main function, which establishes a socket connection with the *SUL Manager* and waits for inputs to dispatch. While any AAL tool can be used, we opt for the mature AAL tool LearnLib [14], making our *AAL* process Java-based. The *SUL Manager* and *SUL*, including the *CC*, are C-based.

2. Initialization: We manually add initialization code to the *SUL*'s new main function, reusing code from the component's original main function.

3. Function Parameters: WEC and MD are control components. They pass function call parameter values along, rather than them being used for control decisions (e.g., `if` statement conditions). We therefore mostly ignore function call parameters, rather than learning register automata. Our generator generates default values for function call arguments (0 for *int*, `NULL` for pointers, etc.). This may be insufficient, e.g., when the *CC* tries to pass along a field of a `NULL`-valued argument for a `struct`-typed parameter. It was an iterative process to manually adapt these values, where the *CC* and existing test code served as inspiration.

4. Interaction Limits: Based on expert knowledge of WEC and MD, we set both interfacing protocol interaction limits: $m_f^{async} = 1$ and $m_f^{fcn} = 1$. Using this low value reduces the size of the *SUL*'s unknown behavior model, for better AAL performance.

5. SUL Compilation: We adapt the component's build scripts, excluding any irrelevant code, e.g., its main function and serialization wrappers, and including the new dispatchers, stubs, and *I/O* module, before compiling the SUL.

6. SUL Manager Compilation: We configure the SUL pool size to 100, and compile the *SUL Manager* using its generated build script.

7. Learner and EO: Our generated *Main* Java class is by default configured to use TTT [13] as *Learner* and Wp [7] as *EO*. To guarantee learning a complete model, Wp requires n, the number of states of the *SUL*'s unknown behavior model. As we don't know n, we guess it, and iteratively increase the value if needed. Caching is also enabled by default.

8. Input Alphabet: The generated *Main* class by default configures the complete input alphabet as derived from the IDL files. It can be reduced to only learn a part of the component's behavior. Considering all provided (to clients) and required (to servers) interfaces, WEC has 591 inputs. It implements 25 distinct workflows. We select five of them, of various complexities, and learn them, including their prerequisite other workflows. For component MD, we keep the complete alphabet with 25 inputs.

9. AAL Process Compilation: We compile the *AAL* process executable.

10. Perform AAL: Finally, we execute the *AAL* process executable to learn models, repeating earlier steps in case of required changes, e.g., after adapting function call arguments or protocol interaction limits.

Each experiment was executed for 24 hours. For WEC, a dedicated system with 24 CPU cores (Intel Xeon Gold 6126) and 64 GB memory was used. For MD, a readily-available virtualized platform with shared resources was used.

We consider the learning/testing rounds up to and including the last learning round that produced the largest hypothesis, and omit subsequent testing that did not find any more counterexamples. Table 2 shows for each (sub-)component (C) the number of inputs (I), the Wp EO n value (Wp-n), the number of Mealy machine states in the model we learned (M-n), the number of equivalence queries (EQs), the number of membership queries (MQs) and membership symbols (MSs), the number of test queries (TQs) and test symbols (TSs), both to the cache (/C) and to the SUL (/S), and the total time in seconds (T).

WEC-1 and WEC-2 are small workflows, without prerequisites. Their largest hypotheses are produced within a few seconds, and no new behavior was found during the many remaining hours of AAL execution. Manual inspection of the component code leads us to conclude they have been learned completely.

WEC-3, WEC-4 and WEC-5 have other workflows as prerequisites. Their largest hypotheses are produced within a few hours. However, they do not accept

Table 2. Case study metrics, per (sub-)component.

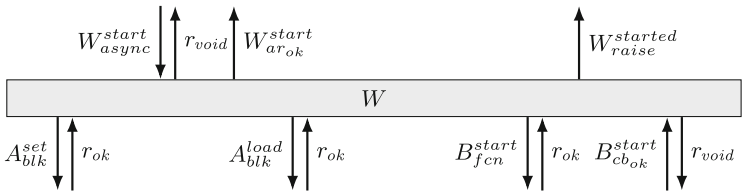
C	WEC-1	WEC-2	WEC-3	WEC-4	WEC-5	MD
I	2	6	25	26	30	25
Wp-n	46	17	66	39	66	916
M-n	50	23	71	55	71	917
EQs	5	4	13	8	13	544
MQs/C	538	967	8,876	7,685	10,644	98,635
MQs/S	52	129	1,462	1,281	1,779	38,300
MSs/C	12,554	12,015	143,335	112,681	171,851	2,122,205
MSs/S	1,015	1,490	23,020	18,266	28,072	854,765
TQs/C	1.59×10^{7}	1,435	1.67×10^{9}	6.57×10^{9}	4.13×10^{9}	9.80×10^{7}
TQs/S	43	3	5,543	3,048	22,689	196,686
TSs/C	4.91×10^{8}	14,074	3.08×10^{10}	1.06×10^{11}	7.65×10^{10}	2.00×10^{9}
TSs/S	1,399	36	88,398	41,046	372,376	4,262,369
T	23	5	2,171	7,569	5,870	62,604

traces that we manually constructed based on their source code. The traces have 86, 100 and 101 inputs, respectively. Learning thus did not yet find all their behavior. This is to be expected though, given that it is hard for black-box testing to find the exact (combination of) prerequisite sequences to test out of all possible test sequences. And even more so considering that we test breadth-first by incrementally increasing the n value of the Wp EO.

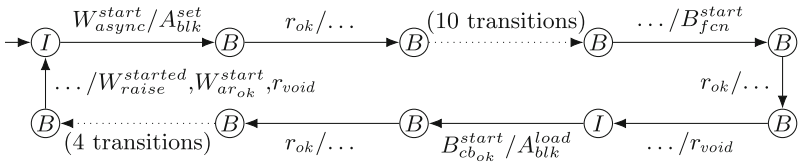
For component MD in total 544 hypotheses are constructed in about 17.4 hours. The last hypothesis accepts our manually constructed trace. Acceptance of this trace, and no further counterexample being found for the remaining 6.6 hours, gives us some confidence that we might have found the complete behavior, although we do not have any evidence that we indeed found all behavior.

The learned models can be used for various purposes [3]. Here, our goal is to facilitate a cost-effective transition to MDE, concretely to Verum's commercial ASD Suite[1], which is based on their patented ASD technology [4], and is used by ASML as MDE tooling. To exemplify this, Fig. 5a shows WEC-2 (abbreviated to W) and part of its anonymized context. Figure 5b shows a part of the anonymized learned Mealy machine of WEC-2. The sink state and its incoming and outgoing transitions are purposely omitted. Figure 5c shows the result of manual conversion of the Mealy machine to a partial ASD design model. The conversion is straightforward: Mealy machine states and transitions correspond one-on-one to states and transitions in the ASD model, where inputs become triggers and outputs become actions. For simplicity, we ignore function parameters. ASD requires for each control component both an interface model and a design model. An interface model can be automatically obtained from the AAL result [3], and then similarly converted to an ASD interface model. From the

[1] See https://verum.com/asd.



(a) W and some of its communications with its context.



(b) Mealy machine representing a partial learning result. Mealy machine states are labelled with interfacing protocol states: I=Idle, Y=Busy, B=Blocked.

W (W.dm)

W | Data Variables | Service References | Implemented Service | Used Services

SBS | States | State Variables | State Diagram

s1 | s1 (initial state)

	Interface	Event	Guard	Actions	State Variable Updates	Target State
1	**s1 (initial state)**					
4	W	start_async		A:A.set+		s2
10	**s2 (synchronous return state)**					
14	A:A	OK		...		s3
19	**s3 (synchronous return state)**					
...						
109	**s13 (synchronous return state)**					
117	...	...		B:B.start_fcn+		s14
118	**s14 (synchronous return state)**					
123	B:B	OK		...		s15
127	**s15 (synchronous return state)**					
135	...	...		W.VoidReply		s16
136	**s16**					
143	B:B	start_cb_ok		A:A.load+		s17
145	**s17 (synchronous return state)**					
149	A:A	OK		...		s18
154	**s18 (synchronous return state)**					
...						
190	**s22 (synchronous return state)**					
198	...	...		W.started; W.start_ar_ok B:B.VoidReply		s1

(c) Screenshot of a partial ASD design model in ASD Suite. Irrelevant rows are hidden.

Fig. 5. Partial example of converting learning results to ASD, for WEC-2 (abbreviated 'W'). Naming scheme: W^{start}_{async} is an asynchronous handler for function *start* of W.

ASD models, new component code can be automatically generated using the ASD Suite. This can then replace the existing *CC*. All that remains is to update the glue code as needed, and to include the ASD runtime for compilation. If a complete Mealy machine of the *CC* was learned, the newly generated code is then a drop-in replacement, its externally-visible communication behavior being identical to that of *CC*.

6 Conclusions and Future Work

In this paper, we describe a general AAL framework to learn the external communication behavior of software components with a client-server architecture, filling a practical gap when applying AAL to such components. Our framework includes an interfacing protocol, which ensures that the SUL satisfies various practical preconditions of AAL, even if the isolated component code does not satisfy them. It is future work to infer pushdown automata to prevent having to use interaction limits to ensure finite learning results.

Our main contribution is the systematic way in which we derive the protocol, handling the different types of (a)synchronous communications. We derive, as an example, such a protocol specifically for ASML's software architecture. However, we rely on generic concepts, e.g., function calls and returns, requests and replies, synchronous vs asynchronous calls, and MSCs, that apply to communication patterns of component-based software in general. We therefore expect that our work can be used to similarly derive such protocols and set up learning frameworks at companies with similar software architectures.

The approach ensures correct-by-construction interfacing protocols, but proving this is considered future work. We do show the feasibility of our approach by applying it to infer the behavior of several ASML (sub-)components. We believe that company engineers should be able to similarly apply our framework, given only a document with detailed instructions, which is future work.

Using generators we automate most of the work to set up an AAL environment. Still, this takes up to a few hours per (sub-)component. It is especially time-consuming to provide sensible function call arguments, to ensure that the SUL does not crash and thus exhibits relevant behavior. It is future work to automate this using white-box techniques, and to infer register automata for components with argument-dependent behavior.

Furthermore, scalability remains a major challenge. Even after hundreds of billions of test symbols, the complete behavior was not learned for some subcomponents. There are various techniques that can improve active learning performance, including checkpointing, incremental equivalence queries [11], white-box approaches [12] and incorporating available traces [19]. Integrating them into our framework is also future work.

Still, for some (sub-)components we learned the complete behavior well within a day. This can significantly reduce the time to obtain a model of their behavior, compared to modeling them in a completely manual way. It is future work to automate the conversion to ASD, and further investigate the qualitative and quantitative advantages of our approach compared to manual modeling.

Acknowledgments. The authors would like to thank ASML for making this work possible and supporting it, and Mladen Skelin for his contributions to this work, in particular the implementation.

References

1. Aarts, F., Heidarian, F., Vaandrager, F.: A theory of history dependent abstractions for learning interface automata. In: Koutny, M., Ulidowski, I. (eds.) CONCUR 2012. LNCS, vol. 7454, pp. 240–255. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-32940-1_18
2. Angluin, D.: Learning regular sets from queries and counterexamples. Inf. Comput. **75**(2), 87–106 (1987). https://doi.org/10.1016/0890-5401(87)90052-6
3. Aslam, K., Cleophas, L., Schiffelers, R., van den Brand, M.: Interface protocol inference to aid understanding legacy software components. Softw. Syst. Model. **19**(6), 1519–1540 (2020). https://doi.org/10.1007/s10270-020-00809-2
4. Broadfoot, G.H., Broadfoot, P.J.: Academia and industry meet: some experiences of formal methods in practice. In: Tenth Asia-Pacific Software Engineering Conference, pp. 49–58. IEEE (2003). https://doi.org/10.1109/APSEC.2003.1254357
5. Cho, C.Y., Babić, D., Shin, E.C.R., Song, D.: Inference and analysis of formal models of botnet command and control protocols. In: Proceedings of the 17th ACM conference on Computer and communications security, pp. 426–439 (2010). https://doi.org/10.1145/1866307.1866355
6. al Duhaiby, O., Mooij, A., van Wezep, H., Groote, J.F.: Pitfalls in applying model learning to industrial legacy software. In: Margaria, T., Steffen, B. (eds.) ISoLA 2018. LNCS, vol. 11247, pp. 121–138. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-03427-6_13
7. Fujiwara, S., Bochmann, G.V., Khendek, F., Amalou, M., Ghedamsi, A.: Test selection based on finite state models. IEEE Trans. Softw. Eng. **17**(6), 591–603 (1991). https://doi.org/10.1109/32.87284
8. Gomaa, H.: Real-time software design for embedded systems. Cambridge University Press, USA, 1st edn. (2016). https://doi.org/10.1017/CBO9781139644532
9. de la Higuera, C.: Grammatical inference: learning automata and grammars. Cambridge University Press (2010). https://doi.org/10.1017/CBO9781139194655
10. Hooimeijer, B., Geilen, M., Groote, J.F., Hendriks, D., Schiffelers, R.: Constructive Model Inference: model learning for component-based software architectures. In: Proceedings of the 17th International Conference on Software Technologies (ICSOFT), pp. 146–158. SciTePress (2022). https://doi.org/10.5220/0011145700003266
11. Howar, F.: Active learning of interface programs, Ph. D. thesis, Technische Universität Dortmund (2012). https://doi.org/10.17877/DE290R-4817
12. Howar, F., Steffen, B.: Active automata learning in practice. In: Bennaceur, A., Hähnle, R., Meinke, K. (eds.) Machine Learning for Dynamic Software Analysis: Potentials and Limits. LNCS, vol. 11026, pp. 123–148. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96562-8_5
13. Isberner, M., Howar, F., Steffen, B.: The TTT algorithm: a redundancy-free approach to active automata learning. In: Bonakdarpour, B., Smolka, S.A. (eds.) RV 2014. LNCS, vol. 8734, pp. 307–322. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11164-3_26

14. Isberner, M., Howar, F., Steffen, B.: The open-source LearnLib. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9206, pp. 487–495. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21690-4_32
15. Merten, M., Isberner, M., Howar, F., Steffen, B., Margaria, T.: Automated learning setups in automata learning. In: Margaria, T., Steffen, B. (eds.) ISoLA 2012. LNCS, vol. 7609, pp. 591–607. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-34026-0_44
16. de Ruiter, J., Poll, E.: Protocol state fuzzing of tls implementations. In: 24th USENIX Security Symposium (USENIX Security 15), pp. 193–206. USENIX Association (2015). https://doi.org/10.5555/2831143.2831156
17. Szyperski, C., Gruntz, D., Murer, S.: Component software: beyond object-oriented programming. Pearson Education, 2nd edn. (2002)
18. Whittle, J., Hutchinson, J., Rouncefield, M.: The state of practice in model-driven engineering. IEEE Softw. **31**(3), 79–85 (2014). https://doi.org/10.1109/MS.2013.65
19. Yang, N., et al.: Improving model inference in industry by combining active and passive learning. In: 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 253–263 (2019). https://doi.org/10.1109/SANER.2019.8668007