Constructive Model Inference: Model Learning for Component-based Software Architectures

Bram Hooimeijer[1][a], Marc Geilen[2][b], Jan Friso Groote[3,4][c],
Dennis Hendriks[5,6][d] and Ramon Schiffelers[4,3][e]

[1] *Prodrive Technologies, Eindhoven, The Netherlands*

[2] *Department of Electrical Engineering, Eindhoven University of Technology, Eindhoven, The Netherlands*

[3] *Department of Mathematics and Computer Science, Eindhoven University of Technology, Eindhoven, The Netherlands*

[4] *ASML, Veldhoven, The Netherlands*

[5] *ESI (TNO), Eindhoven, The Netherlands*

[6] *Department of Software Science, Radboud University, Nijmegen, The Netherlands*

Keywords: Model Learning, Component-based Software, Industrial Application.

Abstract: Model learning, learning a state machine from software, can be an effective model-based engineering technique, especially to understand legacy software. However, so far the applicability is limited as models that can be learned are quite small, often insufficient to represent the software behavior of large industrial systems.

We introduce a novel method, called Constructive Model Inference (CMI). It effectively allows us to learn the behavior of large parts of the industrial software at ASML, where we developed the method and it is now being used. The method uses observations in the form of execution logs to infer behavioral models of concurrent component-based (cyber-physical) systems, relying on knowledge of their architecture, deployment and other characteristics, rather than heuristics or counter examples. We provide a trace-theoretical framework, and prove that if the software satisfies certain architectural assumptions, our approach infers the correct results.

We also present a practical approach to deal with situations where the software deviates from the assumptions. In this way we are able to construct accurate and intuitive state machine models. They provide practitioners with valuable insights into the software behavior, and enable all kinds of behavioral analyses.

1 INTRODUCTION

Model-based systems engineering can cope with the increasing complexity of software (Akdur et al., 2018). However, for most software, especially *legacy software*, no models exist and constructing models manually is laborious and error-prone.

A solution is to learn the models automatically from logs of the software. Model inference has been studied in the fields of *model learning* (de la Higuera, 2010) and *process mining* (van der Aalst, 2016). Both encompass a vast body of research, and do not focus specifically on software systems. Still, the techniques have been applied to software components in general (Heule and Verwer, 2013; van der Aalst et al., 2003), and specifically to legacy components (Schuts et al., 2016; Leemans et al., 2018; Bera et al., 2021).

There are known fundamental limitations to the capabilities of model inference: Mark Gold has proven that accurately generalizing a model beyond observations (logs) is impossible based on observations alone (Mark Gold, 1967). Accurate learning from logs requires additional information, such as for instance counter examples, i.e., program runs that can never be executed. In practice, often heuristics are used, based on the input observations (e.g., considering only the last *n* events) or the resulting model (e.g., limiting the number of resulting states).

Alternative to counter examples, active automata learning queries (parts of) the system (Angluin,

[a] https://orcid.org/0000-0003-0152-4909
[b] https://orcid.org/0000-0002-2629-3249
[c] https://orcid.org/0000-0003-2196-6587
[d] https://orcid.org/0000-0002-9886-7918
[e] https://orcid.org/0000-0002-3297-2969

Hooimeijer, B., Geilen, M., Groote, J., Hendriks, D. and Schiffelers, R.
Constructive Model Inference: Model Learning for Component-based Software Architectures.
DOI: 10.5220/0011145700003266
In *Proceedings of the 17th International Conference on Software Technologies (ICSOFT 2022)*, pages 146-158
ISBN: 978-989-758-588-3; ISSN: 2184-2833

1987). It guarantees that the inferred models exactly match the software implementation. But it suffers from scalability issues (Howar and Steffen, 2018; Yang et al., 2019; Aslam et al., 2020), limiting the learned models to a few thousand states at most.

We aim to infer models for large industrial systems. Our approach therefore learns models from software execution logs, which are interpreted using knowledge of the software architecture, its deployment and other characteristics. It does not rely on queries or counter examples. It also has no heuristics that would be hard to configure correctly, especially if they do not directly relate to system properties.

We instead inject our knowledge of the system's component structure, and the services each component provides, which it can implement by invoking services provided by other components. After a component executes a service, it returns a response and is ready to again provide its services. This knowledge of the components and their services is essential to cope with the complexity of the industrial systems we deal with. It allows us to learn multi-level models that are small enough for engineers to interpret, while capturing the complex system behavior of actual software systems at company ASML, consisting of dozens of components, with states spaces that are too large to interpret (i.e., 10^{10} states and beyond). To the best of our knowledge no similar approach exists.

To demonstrate that our learning approach is adequate, we prove that if a component-based software architecture satisfies our assumptions, our learning approach returns the correct result, both in settings with synchronous and asynchronous communication between the concurrent components. While the actual software largely adheres to our structural and behavioral assumptions, there are however parts that do not. We deal with this by analyzing the learned models, e.g., by searching for deadlocks. Part of the approach is a systematic method to add additional knowledge, to exclude from the models any behavior known to not be exhibited by the system.

Inspection of the learned models by experts led to the judgment that the models are very adequate, and provide them the software behavior abstractions that they currently lack (Yang et al., 2021). Learning larger models is limited not by the software size, but by the capability to analyze those models.

This paper is organized as follows. Section 2 presents an overview of the approach. Section 3 recalls basic definitions. The novel Constructive Model Inference (CMI) approach, our main contribution, is described and analyzed, for synchronously and asynchronously composed component-based systems, in Sections 4 and 5, respectively. Section 6 outlines a method to apply the approach, using a case study at ASML as example. Section 7 draws conclusions.

2 CMI OVERVIEW

We present a high-level outline of the method before discussing the detailed steps in the following sections. Figure 1 illustrates the overall approach and the steps involved. The left column shows the assumed system architecture. A *system* (S) consists of a known set of *components* (C_1, C_2, C_3) that collaborate, communicate and use each other's services. A component, in turn, consists of the *services* it offers (F_1, F_2). We refer to different functions in the component's implementation that together implement a service ($F_{1,1}$, $F_{1,2}$, $F_{1,3}$) as *service fragments*. They handle incoming communications, e.g., client requests and server responses. We use this architecture to decompose observations, downwards in the middle column, and compose models, upwards in the right column, to reconstruct the system behavior.

The inference method requires observing system executions (Preparation). The resulting runtime observations in the form of execution logs, consisting of events, are the input to our method. Using knowledge of the system architecture, the observations are first (Step 1) decomposed into observations pertaining to individual components. Assuming that the beginning and the end of each service fragment can be identified, we decompose observations further into observations of individual service fragments (Step 2).

Then, we infer finite state automata models of service fragments from their observations (Step 3), assuming that offered services may be repeatedly requested, and executed from start to end. The service fragment models are combined to form component models (Step 4), where each component repeatedly executes its services, non-preemptively, one at a time. These component models are put in parallel to form the behavior of the system (Step 6).

The learned system model may exhibit behavior that the real system does not, e.g., due to missing dependencies between service fragments. The method provides for optional refinement (Step 5), whereby behavioral constraints derived from the software architecture and expert knowledge can be added to the component models, in a generic and structured way. Injecting such behavior allows turning stateless services into stateful ones, removing non-system behavior from the models.

The composition in Step 6 can be performed in various ways, depending on assumptions about the way the system is composed of components, i.e., ei-

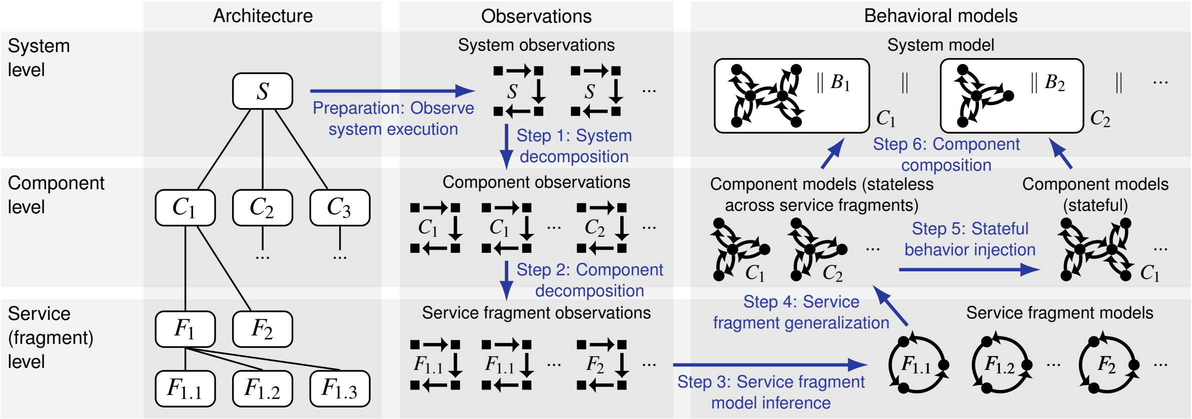
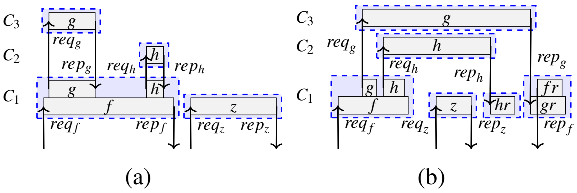

Architecture
Observations
Behavioral models
System level
Component level
Service (fragment) level
S
C1
C2
C3
F1
F2
F1.1
F1.2
F1.3
Preparation: Observe system execution
System observations
Step 1: System decomposition
Component observations
Step 2: Component decomposition
Service fragment observations
Step 3: Service fragment model inference
Service fragment models
Step 4: Service fragment generalization
Component models (stateless across service fragments)
Step 5: Stateful behavior injection
Component models (stateful)
Step 6: Component composition
System model
B1
B2

Figure 1: Constructive Model Inference (CMI): Overview and its six steps, positioned along abstraction levels (rows) and conceptual views (columns).

ther synchronously or asynchronously, with varying buffering and scheduling policies. In the figure, this is visualized as the composition of component models with explicit buffer models (B_1, B_2).

Running Example. Figure 2 shows two examples of observations of the behavior of a component-based system on a horizontal time axis. The system consists of three components, C_1, C_2 and C_3, shown on the vertical axis. In Figure 2a, component C_1 receives an incoming request req_f from the environment, and in response executes its service fragment (function) f, illustrated by the horizontal bar, ultimately leading to a reply rep_f. During the execution of function f, component C_1 executes function g and component C_3 handles it. This involves a remote procedure call (arrow in the figure), with request req_g being sent to component C_3, and after C_3 has executed function g, its reply rep_g being received by C_1. C_1 similarly calls h on component C_2. After C_1 becomes idle again, a second request req_z is received, and handled in service fragment z, leading to a reply rep_z, this time without involving other components.

In the figure, the stacked bars for each component represent call stacks of nested function calls. The bottom bars represent service fragment function executions. Complete call stacks are visualized in the figure by enclosing them in blue dashed rectangles.



(a) (b)

Figure 2: Observations showing client request req_f being handled by component C_1 through calls g and h to its servers, (a) synchronously and (b) asynchronously.

From the start of a service fragment's call stack, until its end where it is idle again, this represents a single observation of its behavior.

Figure 2b shows a different observation of the system, where request req_f is handled asynchronously. Again, services from components C_3 and C_2 are requested, but now the system does not wait for their replies. Instead, it completes service fragment f and proceeds to handle request req_z. When the responses from the other components come in (rep_h and rep_g), it handles these in separate service fragments (hr and gr). Having received both responses, it sends reply rep_f as part of the last service fragment. Here service fragments f, hr and gr together implement a service of C_1, offered via req_f.

Such system behaviors can be observed in the form of an execution log, sequences of events in the order in which they occur in the system. Each start and end of a function execution (bars in Figure 2) has an associated event. We identify an event by its executing component, the related function, and whether it represents the start (↑) or the completion (↓) of its execution. E.g., event $f^{\uparrow}_{C_1}$, abbreviated to $f^{\uparrow}_1$, denotes the start of function f on component C_1. Where appropriate, we identify service fragments by their start events, e.g., $f^{\uparrow}_1$, $z^{\uparrow}_1$, $hr^{\uparrow}_1$, $gr^{\uparrow}_1$, $h^{\uparrow}_2$ and $g^{\uparrow}_3$ for Figure 2.

Our running example has two observations. The first one is the behavior from Figure 2b, i.e., $w_1 = \langle f^{\uparrow}_1, g^{\uparrow}_1, g^{\uparrow}_3, g^{\downarrow}_1, h^{\uparrow}_1, h^{\uparrow}_2, h^{\downarrow}_1, f^{\downarrow}_1, z^{\uparrow}_1, z^{\downarrow}_1, h^{\downarrow}_2, hr^{\uparrow}_1, hr^{\downarrow}_1, g^{\downarrow}_3, gr^{\uparrow}_1, fr^{\uparrow}_1, fr^{\downarrow}_1, gr^{\downarrow}_1\rangle$. The second one is a variation of w_1, where the calls to g and h are reversed, and z handled last, i.e., $w_2 = \langle f^{\uparrow}_1, h^{\uparrow}_1, h^{\uparrow}_2, h^{\downarrow}_1, g^{\uparrow}_1, g^{\uparrow}_3, g^{\downarrow}_1, f^{\downarrow}_1, h^{\downarrow}_2, hr^{\uparrow}_1, hr^{\downarrow}_1, g^{\downarrow}_3, gr^{\uparrow}_1, fr^{\uparrow}_1, fr^{\downarrow}_1, gr^{\downarrow}_1, z^{\uparrow}_1, z^{\downarrow}_1\rangle$. The figure for w_2 is omitted for brevity.

3 PRELIMINARY DEFINITIONS

This section introduces basic definitions that we build upon to place our CMI method in a framework based on finite state automata and regular languages.

3.1 Finite State Automata

Let Σ be a finite set of *symbols*, called an *alphabet*. A word (or string) w over Σ is a finite concatenation of symbols from Σ. The Kleene star closure of Σ, Σ^*, is the set of all finite words over Σ, including the empty word denoted as ε. The Kleene plus closure of Σ is defined as $\Sigma^+ = \Sigma^* \setminus \{\varepsilon\}$.

Given a word w, we denote its length as $|w|$, and its i^{th} symbol as w_i. If words u and v are such that $uv = w$, then u is a prefix of w and v a suffix of w.

We represent inferred models using DFAs:

Definition 3.1 ***(DFA)*** *A deterministic finite automaton (DFA) A is a 5-tuple $A = (Q, \Sigma, \delta, q_0, F)$, with Q a finite set of states, Σ a finite alphabet, $\delta : Q \times \Sigma \hookrightarrow Q$ the partial transition function, $q_0 \in Q$ the initial state and $F \subseteq Q$ a set of accepting states.*

The transition function is extended to words such that $\delta : Q \times \Sigma^* \hookrightarrow Q$, by inductively defining $\delta(q, \varepsilon) = q$ and $\delta(q, wa) = \delta(\delta(q, w), a)$, for $w \in \Sigma^*$, $a \in \Sigma$. DFA $A = (Q, \Sigma, \delta, q_0, F)$ accepts word w iff state $\delta(q_0, w) \in F$. If w is not accepted by A, it is rejected. Set $\mathcal{L}(A) = \{w \in \Sigma^* \mid A \text{ accepts } w\}$ is the language of A.

A DFA is *minimal* iff every two states $p, q \in Q$ $(p \neq q)$ can be distinguished, i.e. there is a $w \in \Sigma^*$ such that $\delta(p, w) \in F$ and $\delta(q, w) \notin F$, or vice versa.

Given a set of words $W \subseteq \Sigma^*$, a *Prefix Tree Automaton PTA(W)* is a tree-structured acyclic DFA with $\mathcal{L}(PTA(W)) = W$, where common prefixes of W share their states and transitions.

Given two languages K, L over the same alphabet, concatenation language KL is $\{uv \mid u \in K, v \in L\}$. The repetition of a language L is recursively defined to be $L^0 = \{\varepsilon\}$, $L^{i+1} = L^i L$. Similarly, the repetition of a word w is $w^0 = \varepsilon$, $w^{i+1} = w^i w$. The Kleene star and plus closures of L are defined as $L^* = \bigcup_{n=0}^{\infty} L^n$ and $L^+ = \bigcup_{n=1}^{\infty} L^n$, respectively.

Given two DFAs A_1, A_2 we define operations on DFAs with notations that reflect the effect on their resulting languages, i.e., $A_1 \cap A_2$, $A_1 \cup A_2$, and $A_1 \setminus A_2$ result in DFAs with languages $\mathcal{L}(A_1) \cap \mathcal{L}(A_2)$, $\mathcal{L}(A_1) \cup \mathcal{L}(A_2)$, and $\mathcal{L}(A_1) \setminus \mathcal{L}(A_2)$, respectively. In addition, we define synchronous composition:

Definition 3.2 ***(Synchronous Composition)*** *Given two DFAs $A_1 = (Q_1, \Sigma_1, \delta_1, q_{0,1}, F_1)$, $A_2 = (Q_2, \Sigma_2, \delta_2, q_{0,2}, F_2)$, their synchronous composition, denoted $A_1 \| A_2$, is the DFA:*

$$A = (Q_1 \times Q_2, \Sigma_1 \cup \Sigma_2, \delta, (q_{0,1}, q_{0,2}), F_1 \times F_2),$$

with $\delta((q_1, q_2), a)$ defined as:

$$\begin{cases} (\delta_1(q_1,a), \delta_2(q_2,a)) & \textit{if } \delta_1(q_1,a), \delta_2(q_2,a) \textit{ are defined,} \\ (\delta_1(q_1,a), q_2) & \textit{if } \delta_1(q_1,a) \textit{ is defined, and } a \notin \Sigma_2, \\ (q_1, \delta_2(q_2,a)) & \textit{if } \delta_2(q_2,a) \textit{ is defined, and } a \notin \Sigma_1, \\ \textit{undefined} & \textit{otherwise.} \end{cases}$$

Two DFAs A_1, A_2 are language equivalent, $A_1 \Leftrightarrow_L A_2$, iff $\mathcal{L}(A_1) = \mathcal{L}(A_2)$. Under language equivalence, each of the operators $\diamond \in \{\|, \cup, \cap\}$ is both commutative and associative, i.e. $A_1 \diamond A_2 \Leftrightarrow_L A_2 \diamond A_1$ and $(A_1 \diamond A_2) \diamond A_3 \Leftrightarrow_L A_1 \diamond (A_2 \diamond A_3)$.

To reason about components of a synchronous composition, we define word projection:

Definition 3.3 ***(Word Projection)*** *Given a word w over alphabet Σ, and a target alphabet Σ', we define the projection $\pi_{\Sigma'}(w) : \Sigma^* \to \Sigma'^*$ inductively as:*

$$\pi_{\Sigma'}(w) = \begin{cases} \varepsilon & \textit{if } w = \varepsilon, \\ \pi_{\Sigma'}(v) & \textit{if } w = va \textit{ with } v \in \Sigma^*, a \notin \Sigma', \\ \pi_{\Sigma'}(v)a & \textit{if } w = va \textit{ with } v \in \Sigma^*, a \in \Sigma'. \end{cases}$$

This definition is lifted to sets of words: $\pi_{\Sigma'}(L) = \{\pi_{\Sigma'}(w) \mid w \in L\}$.

With word projection, we define *synchronization* of languages, which is commutative and associative:

Definition 3.4 ***(Synchronization)*** *Given languages $L_1 \subseteq \Sigma_1^*, L_2 \subseteq \Sigma_2^*$, the* synchronization *of L_1 and L_2 is the language $L_1 \| L_2$ over $\Sigma = \Sigma_1 \cup \Sigma_2$ such that*

$$w \in (L_1 \| L_2) \Leftrightarrow \pi_{\Sigma_1}(w) \in L_1 \wedge \pi_{\Sigma_2}(w) \in L_2.$$

From the definitions we derive:

Proposition 3.5 *Given DFAs A_1, A_2, their synchronous composition is homomorphic with the synchronization of their languages: $\mathcal{L}(A_1 \| A_2) = \mathcal{L}(A_1) \| \mathcal{L}(A_2)$.*

Proof. For proofs, see (Hooimeijer, 2020). □

Corollary 3.6 *Given DFAs A, A_1, A_2 such that $A = A_1 \| A_2$, over alphabets Σ, Σ_1, Σ_2, respectively, then $w \in \mathcal{L}(A) \Leftrightarrow \pi_{\Sigma_1}(w) \in \mathcal{L}(A_1) \wedge \pi_{\Sigma_2}(w) \in \mathcal{L}(A_2)$.*

3.2 Formalizing Concurrent Behavior

Automata model behavior. To represent concurrent behavior, Mazurkiewicz Trace theory is introduced briefly (Mazurkiewicz, 1995). Intuitively, symbols in the alphabets of multiple automata synchronize in the synchronous composition (are dependent), while, e.g., internal non-communicating symbols and communications involving different components interleave (are independent), and can thus be reordered or *commuted*.

Formally, let *dependency* $D \subseteq \Sigma_D \times \Sigma_D$ be a symmetric reflexive relation over *dependency alphabet*

Σ_D. Relation $I_D = (\Sigma_D \times \Sigma_D) \setminus D$ is the *independency* induced by D. Mazurkiewicz *trace equivalence* for D is defined as the least congruence $\equiv_D$ in the monoid Σ_D^* such that for all $a,b \in \Sigma_D$: $(a,b) \in I_D \Rightarrow ab \equiv_D ba$, i.e., the smallest equivalence relation that, in addition to the above, is preserved under concatenation: $u_1 \equiv_D u_2 \wedge v_1 \equiv_D v_2 \Rightarrow u_1v_1 \equiv_D u_2v_2$.

Equivalence classes over $\equiv_D$ are called *traces*. A trace $[w]_D$ for a word w is the set of words equivalent to w under D. This definition is lifted to languages: $[\mathcal{L}]_D = \{[w]_D \mid w \in \mathcal{L}\}$. We drop subscript D if it is clear from the context. Language iteration is extended to traces by defining concatenation of $[u]_D$, $[v]_D \in [\Sigma_D^*]_D$ as $[u]_D[v]_D = [uv]_D$, with $[u]_D$ a prefix of $[uv]_D$ and $[v]_D$ a suffix of $[uv]_D$.

Given a set T of traces, $\mathrm{lin}\, T$ is the linearization of T, i.e., the set $\{w \in \Sigma_D^* \mid [w]_D \in T\}$. For any string language L, if $L = \mathrm{lin}[L]_D$ then L is *consistent* with D, as opposed to when $L \subset \mathrm{lin}[L]_D$. If the language $\mathcal{L}(A)$ of automaton A is consistent with D, A has trace language $\mathcal{T}(A) = [\mathcal{L}(A)]_D$.

Take e.g. $\Sigma_D = \{a,b,c\}$ and $D = \{a,b\}^2 \cup \{a,c\}^2 = \{(a,a),(a,b),(a,c),(b,a),(b,b),(c,a),(c,c)\}$. As b and c can then occur independently, $I_D = \{(b,c),(c,b)\}$. Word $abbca$ is part of trace $[abbca]_D = \{abbca, abcba, acbba\}$, which confirms that commuting b and c results in the same trace.

The commutation of symbols is captured by binary relation $\sim_D$, with $u \sim_D v$ iff there are $x,y \in \Sigma_D^*$ and $(a,b) \in I_D$ such that $u = xaby$ and $v = xbay$. Clearly, $\equiv_D$ is the reflexive transitive closure of $\sim_D$, i.e. $u \equiv_D v$ iff there exists a sequence $(w_0,\ldots,w_n)$ such that $w_0 = u$, $w_n = v$ and $w \sim_D w_{i+1}$ for $0 \leq i < n$.

We rely on an additional result from Mazurkiewicz (Mazurkiewicz, 1995):

Proposition 3.7 *Given dependency D and words $u,v \in \Sigma_D^*$, we have $u \equiv_D v \Rightarrow \pi_\Sigma(u) \equiv_D \pi_\Sigma(v)$ for any alphabet Σ.*

3.3 Asynchronous Compositions

In addition to synchronous composition, we introduce asynchronous composition (Akroun and Salaün, 2018; Brand and Zafiropulo, 1983). This makes use of explicit buffers to pass messages from a sender to a receiver. For an asynchronous composition of DFAs $A_1,\ldots,A_n$, we assume component A_i, $1 \leq i \leq n$, has alphabet Σ_i, partitioned in sending-, receiving- and internal-symbols, $\Sigma_i^!$, $\Sigma_i^?$ and Σ_i^τ, respectively. Each message has a unique sender and receiver, $\Sigma_i^! \cap \Sigma_j^! = \emptyset$, $\Sigma_i^? \cap \Sigma_j^? = \emptyset$, $i \neq j$. The receiver is assumed to exist, and to be different from the sender, $a \in \Sigma_i^! \Rightarrow \exists_{j \neq i} : a \in \Sigma_j^?$. Finally, we assume internal actions are unique to a component, $\Sigma_i^\tau \cap \Sigma_j = \emptyset, i \neq j$. We denote an alphabet under these assumptions as $\Sigma_i^{!,?,\tau}$.

The components communicate via buffers, denoted B_i, which represent, e.g., a FIFO buffer or a bag buffer. FIFO buffers are modeled as a list of symbols over $\Sigma_i^?$, with ε the empty buffer, where messages are added to the tail of the list and consumed from the head of the list. We define the asynchronous composition using FIFO buffers as follows:

Definition 3.8 *Consider n DFAs $A_1,\ldots,A_n$, with $A_i = (Q_i, \Sigma_i^{!,?,\tau}, \delta_i, q_{0,i}, F_i)$, $1 \leq i \leq n$. The asynchronous composition A of $A_1,\ldots,A_n$ using FIFO buffers $B_i,\ldots,B_n$, denoted $A = \|_{i=1}^n (A_i \,\|\, B_i)$, is given as the (typically infinite) state machine:*

$$A = (Q, \Sigma, \delta, (q_{0,1}, \varepsilon, \ldots, q_{0,n}, \varepsilon), F_1 \times \cdots \times F_n)$$

with $Q = Q_1 \times (\Sigma_1^?)^* \times \cdots \times Q_n \times (\Sigma_n^?)^*$, $\Sigma = \bigcup_i \{a! \mid a \in \Sigma_i^!\} \cup \bigcup_i \{a? \mid a \in \Sigma_i^?\} \cup \bigcup_i \Sigma_i^\tau$, and $\delta \subseteq Q \times \Sigma \times Q$ such that for $q = (q_1, b_1, \ldots, q_n, b_n)$ and $q' = (q_1', b_1', \ldots, q_n', b_n')$ we have:

- *(send) $(q, a!, q') \in \delta$ if $\exists_{i,j}$: (i) $a \in \Sigma_i^! \cap \Sigma_j^?$, (ii) $(q_i, a, q_i') \in \delta_i$, (iii) $b_j' = b_j a$, (iv) $\forall_{k \neq i}\, q_k' = q_k$, (v) $\forall_{k \neq j}\, b_k' = b_k$.*
- *(receive) $(q, a?, q') \in \delta$ if $\exists_i$: (i) $a \in \Sigma_i^?$, (ii) $(q_i, a, q_i') \in \delta_i$, (iii) $b_i = ab_i'$, (iv) $\forall_{k \neq i}\, q_k' = q_k$, (v) $\forall_{k \neq i}\, b_k' = b_k$.*
- *(internal) $(q, a, q') \in \delta$ if $\exists_i$: (i) $a \in \Sigma_i^\tau$, (ii) $(q_i, a, q_i') \in \delta_i$, (iii) $\forall_{k \neq i}\, q_k' = q_k$, (iv) $\forall_k\, b_k' = b_k$.*

Bag buffers are defined as a multiset over $\Sigma_i^?$. To use bags instead of FIFOs, we change: ε to $\emptyset$ for A, list type to multiset type for Q, send rule clause (iii) to $b_j' = b_j \cup \{a\}$, and receive rule clause (iii) to $a \in b_i \wedge b_i' = b_i - \{a\}$.

With unbounded buffers, asynchronous compositions can have infinite state spaces. When buffers are bounded, the buffer models can be represented by a (finite) DFA (Muscholl, 2010), and hence the composition as well. We bound buffer B_i to k places, denoted B_i^k, by adding requirement $|b_j| < k$ (such that $|b_j'| \leq k$) to the send rule.

We do not discuss the construction of a DFAs for B_i^k, as it follows from the definition above. Then, the synchronous composition of such constructed DFAs $A_i \,\|\, B_i^k$ is equivalent to asynchronous composition $A_i \,\|\, B_i^k$ as in Definition 3.8, if for synchronous composition we differentiate $a!$ and $a?$ to prevent synchronization of communications to and from buffers, respectively.

The question whether there is a buffer capacity bound such that every accepted word of the unbounded composition is also accepted by the bounded composition is called *boundedness* and is generally undecidable (Genest et al., 2007). However, if the asynchronous composition is 'deadlock free', i.e. a final state can be reached from every reachable state (Kuske and Muscholl, 2019), then it is decidable for a given k whether the asynchronous composition is bounded to k (Genest et al., 2007).

4 CONSTRUCTIVE MODEL INFERENCE (SYNCHRONOUS COMPOSITION)

In this section, we detail our method, considering a system with a synchronous composition of components. We later lift the restriction in Section 5, where we discuss asynchronous compositions.

Recall the CMI method introduced in Section 2, and its overview in Figure 1. In the previous sections we introduced the definitions and the results with which we can now detail the CMI method.

We assume system execution observations are available (*Preparation* step). They are decomposed following the system architecture (Steps 1 and 2). Models are then inferred at the most detailed level, for service fragments (Step 3). Again following the architecture, inferred models are composed to obtain models at various levels of abstraction (Steps 4 – 6).

Formally, in this section, we assume the system under study is a DFA $A = (Q, \Sigma, \delta, q_0, F)$, Σ consists of observable events, A is synchronously composed of n component DFAs, $A = A_1 \parallel \ldots \parallel A_n$, and that observations $W \subseteq \mathcal{L}(A)$ are available to infer an approximation A' of A. We use subscripts for component and service fragment instances, and prime symbols for inferred instances.

4.1 Step 1: System Decomposition

Informal Description: We assume a fixed and known deployment of services on components, such that we can project the observations onto each component.

Formalization: We assume component alphabets Σ_i are known a-priori. W is projected to $\pi_{\Sigma_i}(W)$ for each component, $1 \leq i \leq n$. Recall that we denote events in Σ as f_i^s, with f a function, C_i a component, and $s \in \{\uparrow, \downarrow\}$ denoting the start or completion of a function execution, e.g., $f_1^\uparrow$. Hence, alphabet Σ_i for component C_i contains the events with subscript i.

Example: Consider again the running example from Section 2, with $W = \{w_1, w_2\}$. By projection on, e.g., component C_3, we obtain for both words the projected word $\langle g_3^\uparrow, g_3^\downarrow \rangle$, i.e., $\pi_{\Sigma_3}(W) = \{\langle g_3^\uparrow, g_3^\downarrow \rangle\}$.

4.2 Step 2: Component Decomposition

Informal Description: We assume that, 1) components are sequential (e.g., corresponding to a single operating system thread), 2) client requests (and server responses) can only be handled once the component is idle, and prior requests are finished, i.e., service fragments are executed non-preemptively, and 3) symbols that start or end a service fragment can be distinguished. These assumptions enable us to decompose component observations into service fragment observations.

Formalization: A *task* or *task word* captures a possible execution behavior of a service fragment:

Definition 4.1 ***(Task)*** *Let Σ be a partitioned alphabet $\Sigma^{s,o,e} = \Sigma^s \cup \Sigma^o \cup \Sigma^e$, with service fragment execution start events Σ^s, its corresponding end events Σ^e, and other events Σ^o. Word $w \in \Sigma^*$ is a* task *on component C_i iff $w = f_i^\uparrow v f_i^\downarrow$ with $f_i^\uparrow \in \Sigma^s$, $v \in \Sigma^{o*}$, and $f_i^\downarrow \in \Sigma^e$.*

We also define *task sequence* and *task set*:

Definition 4.2 ***(Task Sequence/Set)*** *Given alphabet $\Sigma^{s,o,e}$, a word $w \in \Sigma^*$ is a* task sequence *iff $w = w_1 \ldots w_n$ with each w_i, $1 \leq i \leq n$, a task. The set $T(w) = \{w_i \,|\, 1 \leq i \leq n\}$ is the* task set *corresponding to w. Similarly $T(W) = \bigcup_{w \in W} T(w)$ for $W \subseteq \Sigma^*$. Identifying a service fragment by its start event $f \in \Sigma^s$, its task set is $T_f(w) = \{v \,|\, v \in T(w) \wedge w_1 = f\}$. $T_f(W) \subseteq T(W)$ is similarly defined.*

For each component C_i, given its component observations $\pi_{\Sigma_i}(W)$, we obtain for each service fragment $f \in \Sigma^s$ its task set $T_f(\pi_{\Sigma_i}(W))$, containing the various observed alternative executions of f.

Example: We get for service fragments $f_1^\uparrow$, $z_1^\uparrow$, $hr_1^\uparrow$, $gr_1^\uparrow$, $h_2^\uparrow$, and $g_3^\uparrow$, their respective task sets: $\{\langle f_1^\uparrow, g_1^\uparrow, g_1^\downarrow, h_1^\uparrow, h_1^\downarrow, f_1^\downarrow \rangle, \langle f_1^\uparrow, h_1^\uparrow, h_1^\downarrow, g_1^\uparrow, g_1^\downarrow, f_1^\downarrow \rangle\}$, $\{\langle z_1^\uparrow, z_1^\downarrow \rangle\}$, $\{\langle hr_1^\uparrow, hr_1^\downarrow \rangle\}$, $\{\langle gr_1^\uparrow, fr_1^\uparrow, fr_1^\downarrow, gr_1^\downarrow \rangle\}$, $\{\langle h_2^\uparrow, h_2^\downarrow \rangle\}$, $\{\langle g_3^\uparrow, g_3^\downarrow \rangle\}$.

4.3 Step 3: Service Fragment Model Inference

Informal Description: We infer a DFA per service fragment. Assuming services may be requested repeatedly, each DFA allows its service fragment to repeatedly be completely executed from start to end.

Formalization: For service fragment $f \in \Sigma^s$, $TDFA_f$ constructs a *Task DFA* (TDFA) from $T_f(\pi_{\Sigma_i}(W))$, i.e., $TDFA_f(W, \Sigma^{s,o,e}, f) = A'_f = (Q, \Sigma_i, \delta, q_0, \{q_0\})$. Its has language $\mathcal{L}(A'_f) = T_f(\pi_{\Sigma_i}(W))^*$, with repeated executions of its observed set of tasks. An efficient way to implement $TDFA_f$ is to build

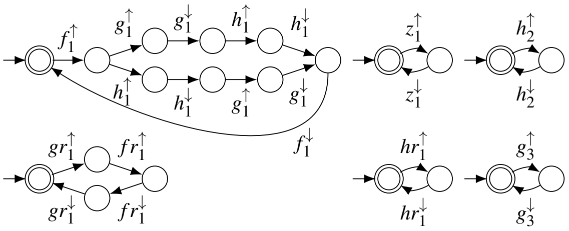


Figure 3: Task DFAs for the service fragments of the running example.

$PTA(T_f(\pi_{\Sigma_i}(W)))$ with root q_0. Then, optionally, minimize the PTA to reduce any redundancy. Finally, merge all accepting states into initial state q_0, which is then the one and only accepting state.

Example: The inferred Task DFAs for the running example's service fragments are shown in Figure 3.

4.4 Step 4: Service Fragment Generalization

Informal Description: We assume components can repeatedly handle requests of their services. We also assume (for now, but we revisit this assumption in Section 4.5) that service executions carry no observable state and are therefore mutually independent. We infer component models from service fragment models, generalizing the component behavior to repeated non-preemptive executions of the various service fragments. A component is thus assumed to be able to execute any number of any of its service fragments in arbitrary order.

Formalization: For a component C_i, its service fragments are $\Sigma_i^s = \Sigma_i \cap \Sigma^s$. $TDFA_i$ constructs TDFA A_i for component C_i, i.e., $TDFA_i(W, \Sigma^{s,o,e}, \Sigma_i^s) = A_i'$. It does so by merging the initial states of all TDFAs $A_f' = TDFA_f(W, \Sigma^{s,o,e}, f)$ for service fragments $f \in \Sigma_i^s$. Then $\mathcal{L}(A_i') = T(\pi_{\Sigma_i}(W))^*$. A_i' is deterministic, as all outgoing transitions from the initial states of TDFAs A_f', i.e., $f_i^\uparrow$, are unique.

Example: The component models for C_2 and C_3 are identical to their service fragment models, for $h_2^\uparrow$ and $g_3^\uparrow$, respectively, in Figure 3. For C_1, the initial states of its four service fragment models are merged.

4.5 Step 5: Stateful Behavior Injection

Informal Description: In Step 4 we assumed service fragments to be mutually independent. This is not always the case in practice. Consider our running example (Figure 2b in Section 2). Service fragment f handles the responses for asynchronous calls g and h in service fragments gr and hr, respectively. Therefore, handling gr always comes after call g in f. This is ensured by the interaction with C_3, but it is not captured in the model for C_1.

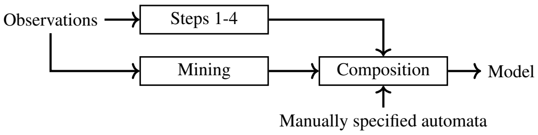


Figure 4: Generic approach to inject specific domain knowledge in Step 5.

Optional Step 5 allows to inject stateful behavior to obtain stateful models that exclude behavior that cannot occur in the real system. As many varieties of component-based systems exist, our structured approach allows to improve the inferred models based on injection of domain knowledge. This allows customization to fit a certain architecture or target system, and is not specific to our case of Section 6. E.g., it allows to capture the general property that 'a response must follow a request' (gr after g).

Figure 4 visualizes the approach. We compose TDFAs inferred in Steps 1 – 4 with additional, typically small, automata, which specify explicitly which behavior we add, constrain or remove. Multiple different composition operators are supported: union, intersection, synchronous composition and (symmetrical) difference (see Section 3). The injected automata are manually specified, or obtained by a miner (Beschastnikh et al., 2013), an automated procedure on the observations.

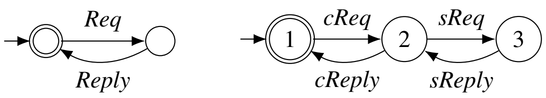
Additional properties that should be added often apply in identical patterns across the whole system. To allow modeling them only once, DFAs with parameters are used, e.g., for the pattern of a request and reply. The parameterized DFA (template) is instantiated for specific symbols, e.g. user-provided request/reply pairs, to obtain the DFAs to inject.

Formalization: We define substitution:

Definition 4.3 ***(Substitution)*** *Given a parameterized DFA $A = (Q, \Sigma, \delta, q_0, F)$, and symbols $p \in \Sigma$, $a \notin \Sigma$, the substitution of a for p in A, denoted $A[p := a]$, is defined as $A[p := a] = (Q, (\Sigma \setminus \{p\}) \cup \{a\}, \delta[p := a], q_0, F)$, with $\delta[p := a](q, c) = \delta(q, p)$ if $c = a$, and $\delta(q, c)$ otherwise.*

In general the order of substitutions matters. We apply them in the given order, and only replace parameter symbols by concrete symbols, assuming both sets are disjoint.

Example 1 (Request before Reply): For asynchronous calls, a reply must follow a request, e.g., gr after g. We model this property as DFA P_1 in Figure 5a. Parameter *Req* represents a request, *Reply* a reply. To



(a) (b)

Figure 5: Parameterized property automata: (a) request before reply (P_1), (b) server replies before client reply (P_2).

enforce the property, we compose the inferred TDFAs A'_i with P_1, for every request and its corresponding reply in the system being considered: $A''_i = A'_i \parallel (\parallel_{a \in reqs} P_1[Req := a][Reply := R(a)])$, where *reqs* is the set of requests, and $R : \Sigma_i \rightarrow \Sigma_i$ maps requests to their replies. E.g., for our example, $g_1^{\downarrow} \in reqs$ and $(g_1^{\downarrow}, gr_1^{\uparrow}) \in R$. By Corollary 3.6, for any $a \in reqs$: $\pi_{\{a,R(a)\}}(\mathcal{L}(A'')) = \{(a.R(a))^n \mid n \in \mathbb{N}\}$. This correctly models the informally given property, assuming at most one outstanding request for each a.

Example 2 (Server Replies before Client Reply): For a service execution, often nested requests should have been replied before finishing the service. We model this property as DFA P_2 in Figure 5b. Upon receiving a client request *cReq*, P_2 goes to state 2. If during this service, a request *sReq* is sent to a server, it must be met with reply *sReply* to get out of state 3 and allow the original service to reply to its client (*cReply*).

The examples show that domain knowledge can be added explicitly, straightforwardly, and with parameterized DFAs and miners also scalably.

4.6 Step 6: Component Composition

Informal Description: The last step is to form system models by composing the obtained stateless and/or stateful component models (from Steps 4 and 5).

Formalization: So far we have used unique symbols per component, such as, e.g., $f_1^{\uparrow}$. Properly capturing component synchronization requires that we ensure the correct communications when one components uses the services of another component. For our running example (see Figure 2b), we have a communication (arrow) from $g_1^{\uparrow}$ to $g_3^{\uparrow}$. To ensure correct synchronization, we use the same symbol for both of them. We combine $g_1^{\uparrow}$ and $g_3^{\uparrow}$ to the synchronization action $g_{1,3}^{\uparrow,\uparrow}$, representing the start of call g on C_1 leading to the immediate start of a handler for g on C_3. Then synchronous composition can be applied to directly obtain $A' = A'_1 \parallel A'_2 \parallel \ldots \parallel A'_n$ as per Definition 3.2.

Example: We omit the resulting system model automaton for brevity.

4.7 Analysis of the CMI Method

Steps 1 and 6: These steps together reduce the problem of inferring system models from system observations to inferring component models from component observations. We first analyze four aspects purely for this reduction, without considering that, e.g., other steps allow repeated (task) execution. Hence, for now we consider PTAs, not TDFAs. And word length of observations is preserved, even after commutations for Mazurkiewicz trace equivalence.

1) We prove that composition $A' = A'_1 \parallel \ldots \parallel A'_n$ of inferred components A'_i accepts the original system observations W from which it was inferred, i.e., $W \subseteq \mathcal{L}(A')$. By definition, if $A' = PTA(W)$ then $\mathcal{L}(A') = W$. We have $A'_i = PTA(\pi_{\Sigma_i}(W))$ instead, for each i. Then by Corollary 3.6 this follows directly.

2) We show that composition A' generalizes to traces, using Mazurkiewicz trace theory. That is, it properly uses concurrency between components to accept all valid alternative interleavings that were not directly observed, i.e., learning a PTA per component generalizes $\mathcal{L}(A')$ from W to $\mathrm{lin}[W]_D$.

Proposition 4.4 *If dependency $D = \bigcup_{i=1}^{n}(\Sigma_i^2)$, $u, v \in \Sigma^*$, $\Sigma = \bigcup_{i=1}^{n} \Sigma_i$, then $u \equiv_D v \Leftrightarrow \forall_i : \pi_{\Sigma_i}(u) = \pi_{\Sigma_i}(v)$.*

Corollary 4.5 *For a synchronous composition $A = A_1 \parallel \ldots \parallel A_n$ and $D = \bigcup_{i=1}^{n}(\Sigma_i^2)$, $\mathcal{L}(A) = \mathrm{lin}[\mathcal{L}(A)]_D$.*

Per Proposition 4.4 and Corollary 4.5, decomposing the system to components preserves commutations, and all commutated words are indeed in $\mathcal{L}(A')$.

3) We show A' does not over-generalize A. That is, the inferred model has no behavior that the original system does not have, i.e., $\mathcal{L}(A') \subseteq \mathcal{L}(A)$. As A and A' both generalize to traces per Corollary 4.5, and given $W \subseteq \mathcal{L}(A)$, this follows directly. Together this leads to the following theorem:

Theorem 4.6 *Consider DFA $A = \parallel_{i=1}^{n} A_i$, $W \subseteq \mathcal{L}(A)$, DFA $A' = \parallel_{i=1}^{n} A'_i$ with $A'_i = PTA(\pi_{\Sigma_i}(W))$, and $D = \bigcup_{i=1}^{n}(\Sigma_i^2)$. Then $\mathcal{L}(A') = \mathrm{lin}[W]_D$ and $\mathcal{L}(A') \subseteq \mathcal{L}(A)$.*

Generally, it is sufficient to show this per component:

Proposition 4.7 *If $\mathcal{L}(A'_1) \subseteq \mathcal{L}(A_1)$ and $\mathcal{L}(A'_2) \subseteq \mathcal{L}(A_2)$, then $\mathcal{L}(A'_1 \parallel A'_2) \subseteq \mathcal{L}(A_1 \parallel A_2)$.*

4) A' is robust under additional observations, i.e., if inference uses additional observations, then the language of the inferred model can only grow:

Theorem 4.8 *Consider DFA $A' = \parallel_{i=1}^{n} A'_i$ with $A'_i = PTA(\pi_{\Sigma_i}(U))$, obtained from observations U, and DFA A'', similarly composed and obtained from observations V, with $U \subseteq V$. Then $\mathcal{L}(A') \subseteq \mathcal{L}(A'')$.*

Steps 2–4: Steps 2 and 4 together reduce the problem of inferring component models from component

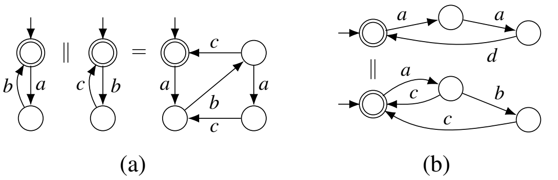


Figure 6: (a) TDFAs $X_1' \,||\, X_2' = X'$, (b) TDFAs $Y_1' \,||\, Y_2'$.

observations to inferring service fragment models from service fragment observations, which is realized by Step 3. We analyze the four aspects as before, plus an extra one. Unlike before, we now do consider repeated (task) executions and TDFAs.

1) We show that our approach produces correct component Task DFAs A_i', and that composition $A' = A_1' \,||\, \dots \,||\, A_n'$ then still accepts W:

Proposition 4.9 *Consider DFA A with $\Sigma^{s,o,e}$ composed of Task DFAs, $A = ||_{i=1}^{n} A_i$ with each a $\Sigma_i^s = \Sigma_i \cap \Sigma^s$, observations $W \subseteq \mathcal{L}(A)$, and DFAs $A_i' = TDFA_i(W, \Sigma^{s,o,e}, \Sigma_i^s)$. Then $\mathcal{L}(A_i') = T(\pi_{\Sigma_i}(W))^*$.*

Proposition 4.9 follows by construction. Then, by Corollary 3.6, also $W \subseteq \mathcal{L}(A')$ still holds.

2) We consider how A' generalizes observations. A composition A' of TDFAs A_i' is not generally a Task DFA. Even if a symbol is only in Σ^e for all components, transitions for that symbol might not go to the accepting state (e.g. *aba***c** over Figure 6a). Yet, by Proposition 3.5, we know that any word in the language of A represents executions of tasks on components. Hence, we extend tasks to task traces and task sequences to task sequence traces:

Definition 4.10 ***(Task (Sequence) Trace)*** *Consider a dependency $D = \bigcup_{i=1}^{n} \Sigma_i^2$ and alphabet $\Sigma^{s,o,e}$. A trace $[t] \in [\Sigma_D^*]$ is a* task sequence trace *iff $\forall_{i=1}^{n}\ \pi_{\Sigma_i}(t)$ is a task sequence for $\Sigma^{s,o,e}$. Its task set $T([t])$, is given as $\{[t_j] \mid [t_1 \dots t_m] = [t]$ and t_j a task for $1 \leq j \leq m\}$. If additionally, no task sequences $[u], [v] \in [\Sigma_D^+]$ exist such that $[uv] = [t]$, then $[t]$ is not a concatenation of multiple task traces, but a single* task trace*.*

Using task traces, we define how a composition of concurrently executing components generalizes, i.e., what commutations are possible:

Proposition 4.11 *Consider a composition $A = ||_{i=1}^{n} A_i$ of n Task DFAs, with dependency $D = \bigcup_{i=1}^{n} \Sigma_i^2$. Then any $[t] \in \mathcal{T}(A)$ is a task sequence, and furthermore $[t] \in \mathcal{T}(A) \Leftrightarrow T([t]) \subseteq \mathcal{T}(A)$, and $\mathcal{T}(A) = T(\mathcal{T}(A))^*$.*

For the TDFAs in Figure 6a, Mazurkiewicz traces allow us to define $\mathcal{T}(X') = T([abacbc])^* = \{abc, abacbc\}^*$. Clearly, $\mathcal{T}(X')$ allows commutations, as $abc \in \mathcal{T}(X')$, while $abc \notin T(abacbc)^*$.

Corollary 4.5 earlier showed that A' generalizes to trace equivalence, $[W] \in \mathcal{T}(A')$. Proposition 4.11 proves even more generalization: all task traces in W can be repeated in any order $T([W])^* \subseteq \mathcal{T}(A')$.

However, A' generalizes beyond $T([W])^*$. For Figure 6a, $\mathcal{L}(X') = ab(acb)^*c$, beyond $\mathcal{T}(X')$. Consider also Y_1' and Y_2' in Figure 6b, inferred from $W = \{abcacd\}$ for $\Sigma_1 = \{a, d\}$ and $\Sigma_2 = \{a, b, c\}$. Here, $[w]$ is a task, as *aad* cannot be split. Yet $Y' = Y_1' \,||\, Y_2'$ also accepts $w' = abcabcd$ and $w'' = acacd$, outside $T([W])^*$. Y_2' accepts tasks *ac* and *abc* that synchronize equally with Y_1' and can thus be interchanged. Therefore, characterizing the full generalization remains an open problem.

3) The inferred component models A_i' do not overgeneralize, i.e., $\mathcal{L}(A_i') \subseteq \mathcal{L}(A_i)$, per Proposition 4.9 and since $\pi_{\Sigma_i}(W) \subseteq T(\pi_{\Sigma_i}(W))^*$. Then for A' also still $\mathcal{L}(A') \subseteq \mathcal{L}(A)$, per Proposition 4.7.

4) Allowing for repeated task executions, A' is still robust under additional observations:

Theorem 4.12 *Consider DFA $A' = ||_{i=1}^{n} A_i'$, with $A_i' = TDFA_i(U, \Sigma^{s,o,e}, \Sigma_i^s)$ obtained from observations U, and DFA A'' similarly composed and obtained from observations V, with $U \subseteq V$. Then $\mathcal{L}(A') \subseteq \mathcal{L}(A'')$.*

5) Finally, we consider completeness. To infer the complete behavior of system A, its component task sets $T(\mathcal{L}(A_i))$ should at least be finite, as we enumerate these tasks in approximations A_i'. If all tasks of A are observed, the full system behavior is inferred.

5 CONSTRUCTIVE MODEL INFERENCE (ASYNCHRONOUS COMPOSITION)

In Section 4 we considered systems consisting of synchronously composed components. This section considers the CMI approach applied to systems with asynchronously composed components.

We now assume the system is a DFA A, asynchronously composed of components A_i, using buffers B_i bounded to capacity b_i. Per Section 3.3 we model this as a synchronous composition with buffer automata, $A = ||_{i=1}^{n} (A_i \,||\, B_i^{b_i})$. Then A_i has alphabet $\Sigma_i^{!,?,\tau}$, $\Sigma_{B_i} = \Sigma_{B_i}^? \cup \Sigma_{B_i}^!$, $\Sigma_{B_i}^? = \{a! \mid a? \in \Sigma_i^?\}$, and $\Sigma_{B_i}^! = \Sigma_i^?$. We assume system observations $W \subseteq \mathcal{L}(A)$ as before.

To infer A', we first infer the components A_i', reusing Steps 1–5 from the synchronous case of Section 4. Then, we model DFA buffers $B_i'^{b_i}$ as in Section 3.3, according to the buffer type (FIFO or bag) and capacity bound b_i. A capacity lower bound

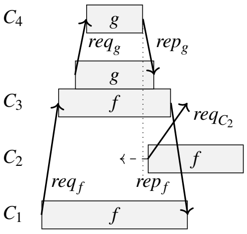


Figure 7: Example of limited commutations for FIFO buffers.

$b_{i,w}$ for each buffer B_i is inferred from W, by considering the maximally occupied buffer space along each observed word $w \in W$. That is $b_{i,w} = \max_{1 \leq i \leq |w|}(|\pi_{\Sigma^?_{B_i}}(w_1 \dots w_i)| - |\pi_{\Sigma^!_{B_i}}(w_1 \dots w_i)|)$. Then, the overall lower bound b_i for B_i is inferred, as $b_i = \max_{w \in W} b_{i,w}$. Finally, A' is given as synchronous composition $A' = \|_{i=1}^{n}(A'_i \,\|\, B_i'^{b_i})$.

Just as for the synchronous case, we prove for this asynchronous version of Step 6, that A' accepts W, does not over-generalize, and is robust under additional observations:

Proposition 5.1 *Consider DFA* $A = \|_{i=1}^{n}(A_i \,\|\, B_i)$ *with* B_i *a FIFO (or bag) buffer,* $W \subseteq \mathcal{L}(A)$, *DFA* $A' = \|_{i=1}^{n}(A'_i \,\|\, B_i'^{b_i})$ *with* $A'_i = TDFA_i(W, \Sigma^{s,o,e}, \Sigma^s_i)$, *and* $B_i'^{b_i}$ *a FIFO (or bag) buffer with* $b_i = \max_{w \in W} b_{i,w}$. *Then* $W \subseteq \mathcal{L}(A') \subseteq \mathcal{L}(A)$. *For DFA* A'', *similarly composed and obtained from observations* V, *with* $W \subseteq V$, *then holds* $\mathcal{L}(A') \subseteq \mathcal{L}(A'')$.

With this approach we need to know a-priori the kind of buffers used in our system. To understand whether our choices were correct we experimented with various buffer types. For instance, using a single FIFO buffer between each pair of components can lead to an issue shown in Figure 7. If req_{C_2} is sent before rep_g, the FIFO order enforces reception of req_{C_2} before rep_g, while the TDFA for C_3 has to handle rep_g before starting a new service with req_{C_2}, leading to a deadlock. In order to resolve this problem, and other mismatches with ASML's middleware, we use a FIFO buffer per client that a component communicates with, and a bag per server.

6 CMI IN PRACTICE

We demonstrate our CMI approach by applying it to a case study at ASML. ASML designs and builds machines for lithography, which is an essential step in the manufacturing of computer chips.

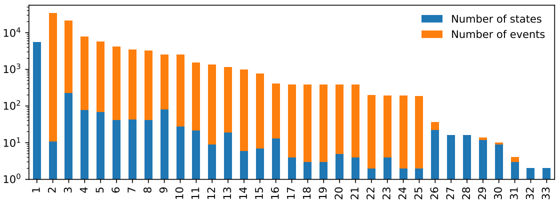


Figure 8: Per component, the number of states in its component model after Step 4, compared to the number of events in its observations.

6.1 System Characteristics

Our CMI approach requires as input system observations. The middleware in ASML's systems has been instrumented to extract Timed Message Sequence Charts (TMSCs) from executions. A TMSC (Jonk et al., 2020) is a formal model for system observations, akin to what we described in Section 2. The TMSC formalism implies that the system can be viewed as a composition of sequential components bearing nested, fully observed, non-preemptive function executions. We can therefore obtain observations W, component alphabets Σ_i, and partitioned alphabet $\Sigma^{s,o,e}$, from TMSCs, to serve as inputs to our method.

For this case study, we infer a model of the exposure subsystem, which exposes each field (die) on a wafer. By executing a system acceptance test, and observing its behavior during the exposure of a single wafer, which spans about 11 seconds, a TMSC is obtained that consists of around 100,000 events for 33 components.

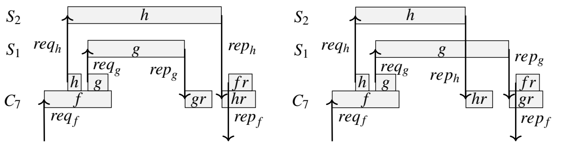
6.2 Model Inference Steps 1 – 4

We apply Steps 1 – 4 for our case study. Figure 8 shows the sizes of the resulting component models in terms of the number of events in the observations and the number of states of the inferred models. For most components the inferred model is two orders of magnitude more compact than its observations, showing repetitive service fragment executions in these components.

Component 1 orchestrates the wafer exposure. Its model has the same size as its observation, as it spans a single task with over 5,000 events. For components 26 – 33, the small reduction is due to their limited observations.

6.3 Model Inference Step 5

We analyze the inferred models, assessing whether their behavior is in accordance with what we expect



(a) (b)

Figure 9: Component C_7 requires both rep_g and rep_h to reply to its client. (a) *gr* before *hr*, with *fr* in *hr*, and (b) *hr* before *gr*, with *fr* in *gr*.

from the actual software and if not, what knowledge must be added (according to Step 5).

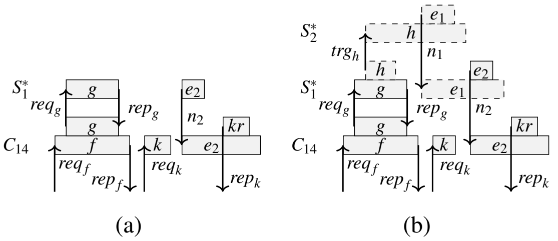
Although the system matches an asynchronous composition of components as discussed in Section 5, we first apply our CMI approach for synchronous systems (Step 6 from Section 4). A synchronous composition has more limited behavior compared to an asynchronous composition, and hence a smaller state space. The resulting model can therefore be analyzed for issues more easily, while many such issues apply to both the synchronous and the asynchronous compositions.

We observe that component model 1, A'_1, has over 5,000 states, in a single task w. To reduce the model size, we analyze w for repetitions and reduce it (Nakamura et al., 2013). We look for short $x, y, z \in \Sigma_1^*$ such that $w = xy^n z$ for some n. Then, we create reduced model A''_1, with $\mathcal{L}(A''_1) = (xyz)^*$. This reduces the model size by $|y| * (n-1)$ states to about 600 states for Component 1, without limiting its synchronization with the other components.

As a second reduction, we remove DFA transitions that do not communicate and originate from a state which has a single outgoing transition, i.e. does not allow for a choice in the process. This is akin to process algebra axiom $a.\tau.b = a.b$, where τ is a non-synchronizing action (Milner, 1989). Examples in Figure 9 are $g_7^\downarrow$ and $f_7^\downarrow$. This further reduces A''_1 from about 600 to about 300 states. Other components are reduced as well.

With these two reductions, exploring the resulting state space becomes feasible, being approximately 10^{10} states. We look for deadlocks: states without outgoing transitions. As our learned model represents actual software, we expect no deadlocks. If a deadlock arises, it is due to a synchronizing symbol, the counterpart of which cannot be reached.

A first issue is illustrated in Figure 9. Component C_7 requests services g and h concurrently. Both are called asynchronously and can return in either order, with only the last reply leading to rep_f. The inferred 'stateless' model does not capture the dependency between replies rep_g, rep_h and rep_f. When in the learned model both, or neither, incoming replies are followed by rep_f, the system deadlocks, as the calling environment expects exactly one reply. This is solved by enforcing nested services to be finished before finishing f, as in Example 2 from Step 5 of Section 4.5.



(a) (b)

Figure 10: Component C_{14} communicates with components outside the observations, causing missing dependencies.

A second issue is illustrated in Figure 10, where component C_{14} deals with server S_1^*. However, we do not observe S_1^* directly, but merely through the messages obtained at C_{14}. The server S_2^*, which S_1^* uses, is not observed at all. The observation is shown in Figure 10a, and a possible perspective of the actual system is shown in Figure 10b.

Since we do not observe S_1^* and S_2^*, the model misses the dependency between functions g and e_2. Now, e_2 has no incoming message and it is able to start 'spontaneously'. The actual system relies on the dependency, and as it is missing in the learned behaviour, it has deadlocks. Knowing the missing dependency from domain knowledge, we inject it through Step 5 as a one-place buffer similar to Figure 5a, extended to multiple places as needed.

After resolving these issues, we analyze choices. From other observations we know that g is optional when performing f in component C_7 of Figure 9a. The inferred model thus contains tasks which have a common prefix, i.e. $\langle f_7^\uparrow, h_7^\uparrow, h_7^\downarrow, f_7^\downarrow \rangle$ (g is skipped), and $\langle f_7^\uparrow, h_7^\uparrow, h_7^\downarrow, g_7^\uparrow, g_7^\downarrow, f_7^\downarrow \rangle$ (g is called). After $h_7^\downarrow$ a choice arises, to either call g by $g_7^\uparrow$ or finish f by $f_7^\downarrow$. Such choices can enlarge the state space, and may not apply in all situations. We therefore asked domain experts on which information in the software such a choice could depend. Together, we concluded that g is only skipped once, and this relates to the repetitions we observed for C_1, as g is skipped for one particular such iteration. We ensured the correct choice is made, by constraining the two options to their corresponding iterations on C_1, thus removing some non-system behavior from the inferred model.

6.4 Model Inference Step 6

To the result of Step 5, we apply asynchronous composition (Step 6, Section 5). Using FIFO and bag buffers as described in that section, service fragments are allowed to commute due to execution and communication time variations.

For our case study, all inferred buffer capacities are either one or two, except component C_2, which has buffers with capacities up to 24. The low number of buffer places is due to the extensive synchronization on replies. C_2, the log component, only receives 'fire-and-forget'-type logging notifications without replies.

We verified that the resulting, improved, asynchronously composed model indeed accepts its input TMSC, i.e., the inferred model accepts the input observation from which it was inferred. This gives trust that the practically inferred model is in line with that observation. The inferred models were confirmed by domain experts as remarkably accurate, allowing them to discuss and analyze their system's behavior. This in stark contrast to models that were previously inferred using process mining and heuristics-based model learning, where they questioned the accuracy of the models instead.

7 CONCLUSIONS AND FUTURE WORK

We introduced our novel method, Constructive Model Inference (CMI), which uses execution logs as input. Relying on knowledge of the system architecture, it allows learning the behavior of large concurrent component-based systems. The trace-theoretical framework provides a solid foundation. ASML considers the state machine models resulting from our method accurate, and the service fragment models in particular also highly intuitive. They see many potential applications, and are already using the inferred models to gain insight into their software behavior, as well as for change impact analysis.

Future work includes among others extending the CMI method to inferring Extended Finite Automata and Timed Automata, further industrial application of the approach, and automatically deriving interface models from component models.

ACKNOWLEDGEMENTS

This research is partly carried out as part of the Transposition project under the responsibility of ESI (TNO) in co-operation with ASML. The research activities are partly supported by the Netherlands Ministry of Economic Affairs and TKI-HTSM.

REFERENCES

Akdur, D., Garousi, V., and Demirörs, O. (2018). A survey on modeling and model-driven engineering practices in the embedded software industry. *Journal of Systems Architecture*, 91(October):62–82.

Akroun, L. and Salaün, G. (2018). Automated verification of automata communicating via FIFO and bag buffers. *Formal Methods in System Design*, 52(3):260–276.

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

Aslam, K., Cleophas, L., Schiffelers, R., and van den Brand, M. (2020). Interface protocol inference to aid understanding legacy software components. *Software and Systems Modeling*, 19(6):1519–1540.

Bera, D., Schuts, M., Hooman, J., and Kurtev, I. (2021). Reverse engineering models of software interfaces. *Computer Science and Information Systems*.

Beschastnikh, I., Brun, Y., Abrahamson, J., Ernst, M. D., and Krishnamurthy, A. (2013). Unifying FSM-inference algorithms through declarative specification. *Proceedings - International Conference on Software Engineering*, pages 252–261.

Brand, D. and Zafiropulo, P. (1983). On Communicating Finite-State Machines. *Journal of the ACM (JACM)*, 30(2):323–342.

de la Higuera, C. (2010). *Grammatical Inference*. Cambridge University Press, New York, 1st edition.

Genest, B., Kuske, D., and Muscholl, A. (2007). On communicating automata with bounded channels. *Fundamenta Informaticae*, 80(1-3):147–167.

Heule, M. J. and Verwer, S. (2013). Software model synthesis using satisfiability solvers. *Empirical Software Engineering*, 18(4):825–856.

Hooimeijer, B. (2020). Model Inference for Legacy Software in Component-Based Architectures. Master's thesis, Eindhoven University of Technology.

Howar, F. and Steffen, B. (2018). Active automata learning in practice. In Bennaceur, A., Hähnle, R., and Meinke, K., editors, *Machine Learning for Dynamic Software Analysis: Potentials and Limits: International Dagstuhl Seminar 16172, Dagstuhl Castle, Germany, April 24-27, 2016, Revised Papers*, pages 123–148, Cham. Springer International Publishing.

Jonk, R., Voeten, J., Geilen, M., Basten, T., and Schiffelers, R. (2020). SMT-based verification of temporal properties for component-based software systems. Technical Report ES Reports, Eindhoven University of Technology, Department of Electrical Engineering, Electronic Systems Group, Eindhoven.

Kuske, D. and Muscholl, A. (2019). Communicating Automata. *Submitted for peer review (Journal unknown) Retrieved from http://eiche.theoinf.tu-ilmenau.de/kuske/Submitted/cfm-final.pdf.*

Leemans, M., van der Aalst, W. M., van den Brand, M. G., Schiffelers, R. R., and Lensink, L. (2018). Software Process Analysis Methodology – A Methodology based on Lessons Learned in Embracing Legacy Software. In *2018 IEEE International Conference on*

Software Maintenance and Evolution (ICSME), pages 665–674. IEEE.

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

Mazurkiewicz, A. (1995). Introduction to Trace Theory. In Diekert, V. and Rozenberg, G., editors, *The Book of Traces*, chapter 1, pages 3–41. World Scientific, Singapore.

Milner, R. (1989). *Communication and Concurrency*. Prentice-Hall, Inc., River, NJ, United States.

Muscholl, A. (2010). Analysis of communicating automata. In *LATA 2010: Language and Automata Theory and Applications*, pages 50–57.

Nakamura, A., Saito, T., Takigawa, I., and Kudo, M. (2013). Fast algorithms for finding a minimum repetition representation of strings and trees. *Discrete Applied Mathematics*, 161(10-11):1556–1575.

Schuts, M., Hooman, J., and Vaandrager, F. (2016). Refactoring of Legacy Software using Model Learning and Equivalence Checking: an Industrial Experience Report. In *Integrated Formal Methods: 12th International Conference*, pages 311–325.

van der Aalst, W. (2016). *Process Mining*. Springer-Verlag Berlin Heidelberg, Berlin Heidelberg, 2nd edition.

van der Aalst, W. M., van Dongen, B. F., Herbst, J., Maruster, L., Schimm, G., and Weijters, A. J. (2003). Workflow mining: A survey of issues and approaches. *Data and Knowledge Engineering*, 47(2):237–267.

Yang, N., Aslam, K., Schiffelers, R., Lensink, L., Hendriks, D., Cleophas, L., and Serebrenik, A. (2019). Improving Model Inference in Industry by Combining Active and Passive Learning. *SANER 2019 - Proceedings of the 2019 IEEE 26th International Conference on Software Analysis, Evolution, and Reengineering*, pages 253–263.

Yang, N., Schiffelers, R., and Lukkien, J. (2021). An interview study of how developers use execution logs in embedded software engineering. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 61–70.