Model inference for legacy software in component-based architectures

Hooimeijer, Bram J.

Award date:
2020

Series title:
Master graduation paper, Electrical Engineering

Commisioned by professors	Group & chair	Date of final presentation	Report number
Jan Friso Groote, prof.dr.ir. (W&I, FSA) Marc Geilen, dr.ir. (EE, ES)	Jeroen Voeten, prof.dr.ir. Electronic Systems Group	21/10/2020	-

Model Inference for Legacy Software in Component-Based Architectures

by

B.J. (Bram) Hooimeijer

EINDHOVEN UNIVERSITY OF TECHNOLOGY

Department of Electrical Engineering

Den Dolech 2, 5612 AZ Eindhoven
P.O. Box 513, 5600 MB Eindhoven
The Netherlands

https://www.tue.nl/en/

Internal supervisors: J.F. (Jan Friso) Groote, prof.dr.ir.,
M.C.W. (Marc) Geilen, dr.ir.

External supervisors : R.R.H. (Ramon) Schiffelers, dr.ir., ASML
L.J. (Bram) van der Sanden, dr.ir., TNO-ESI

Model Inference for Legacy Software in Component-Based Architectures

Thesis

Bram Hooimeijer
0852038, b.j.hooimeijer@student.tue.nl
bhooimei, bram.hooimeijer@asml.com

***Abstract*—Given their complexity, modern cyber physical systems would greatly benefit from model-based engineering techniques. Yet for these systems, models are often not available, while model inference from observations only has been proven to be impossible without either counter-examples or additional assumptions.**

We infer models of component-based cyber physical systems by relying on the architecture, deployment and characteristics of the system. Using these, we decompose the system in components, which are in turn decomposed in services. We show that, by inferring services, we can build up models which cover generalizations that are valid for such systems.

We demonstrate the method using a case study at ASML, and thereby give a practical approach to assess where assumptions are violated and how to improve models in these cases. Finally, we arrive at a model, orders smaller than the observation, with a rich class of explainable generalizations, matching the characteristics of component based systems.

I. Introduction

By using models for the design of software systems, model-based systems engineering has been able to cope with the increasing complexity of software [1]. The use of models allows for e.g. formal verification [14, 15, 19], or synthesis of software [39, 43, 50]. This increases the efficiency, reliability and maintainability of software.

Model based techniques cannot be directly applied to software of which no models exist. For example, complex cyber physical systems are continuously improved, and hence new revisions often rely on existing software components [40]. Problems arise when these components are based on obsolete technologies or lack up-to-date documentation [47, 49]. For such *legacy components*, models are not available, making it hard to predict performance or verify correctness automatically.

We focus on component-based software systems, in which legacy components are often complex. As this makes manual modeling time consuming and error-prone, the question arises whether it is possible to infer models algorithmically.

Model inference has been studied in the fields of *model learning* [16] and *process mining* [44]. Both encompass a vast body of research, and do not focus specifically on software. Still, the techniques have been applied to software [20, 21, 45], and specifically to legacy components [3, 28, 40].

There are key limitations to model inference: Mark Gold has proven that generalizing a model beyond observations is impossible based on observations alone [30]. Counter examples, i.e. behavior outside the model, prevent the model from over-generalizing. Though counter examples have been used in e.g. software synthesis applications [20, 21], these are often not directly available for legacy software.

Alternative to counter-examples, queries on the source code ('active learning') can guarantee that the inferred models match the software. Al Duhaiby concludes that this captures both observed behavior, as well as 'hidden behavior' [3], i.e. behavior which, though present in the software, is practically never observed and thus deemed irrelevant. The additional complexity can make models difficult to interpret, can lead to scaling issues [7, 51], or can be problematic if the hidden behavior is faulty, as can be the case with legacy components [3, 7].

To learn from observations only, model inference and process mining rely on heuristics [16, 44]. Often, these are model assumptions which cannot be related to the system, e.g. in [8], which makes it hard to justify the approach. Additionally, parameters required for some heuristics [8, 16] become difficult to tune, changing the issue of finding a model into deciding which model is correct. Even when assumptions relate to the system, there is often no formal justification of the approach, as concluded in the survey as part of [28].

Naturally, heuristics which do not capture characteristics of the system lead to generalizations which do not capture these characteristics either. For these algorithms, the order in which generalizations are applied heavily influences the result [21, 27]. This is especially the case for sparse systems, i.e. where allowed behavior is limited w.r.t. all behavior over the observed actions, such as software [13]. As a consequence of this, additional observations can have a large and unpredictable impact on the model [13], making it difficult to iteratively refine the results.

One characteristic that is not captured well is concurrency, which allows independent actions to commute, or 'interleave'. Active learning does infer all different commutations explicitly, yet this makes the inferred models large and complex to interpret [3]. When learning without queries on the contrary, a commutation must be observed to infer concurrency [11, 28], and might therefore infer false sequential dependencies between events when the commutation is not observed.

We conclude that, though traditional inference approaches generalize based on observations only, these generalizations are not justified based on the target system. Thereby, generalizations do not capture characteristics of the system, are not relatable to the observation, and are not robust under new observations.

This work addresses the issue of model inference for complex cyber physical systems based on a component based software architecture, using observations and without relying on queries or counter examples.

Considering that observations alone are not sufficient to generalize, we rely on the architecture, deployment and characteristics of the system to propose a method for model inference. Hence, we pose the following research question: *How can we utilize knowledge of the architecture, deployment and characteristics of the target system to formulate a justified model inference approach for component based cyber physical systems?*

The contribution of this work is a method which is:

Justified: We prove that generalizations made belong to a system matching our assumptions, and discuss how to assess whether the system adheres to our assumptions.

Appropriate: There is a clear relation between the observation and the inferred model. The model clearly relates to the architecture of the system and captures essential characteristics.

Robust: If the set of observations is expanded, the impact on the inferred models is predictable.

Extensible: Domain-specific knowledge or insights can be used to manually extend the inference method.

This work is organized as follows. In Section II we recall basic definitions, required for the literature review in Section III. In Sections IV and V we propose and analyze an approach to infer models of synchronous- and asynchronous component-based systems respectively, which are the main contributions of this work. In Section VI, we outline a method to apply the approach, taking a case study at ASML N.V. as example. Finally, we discuss our approach and draw our conclusions.

II. Preliminary Definitions

Let Σ be a finite set of symbols, called an alphabet. A word w over Σ is a finite concatenation of symbols, with Σ^* the set of all finite words over Σ, including empty word ε.

Given a word w, we define its length to be $|w|$, with w_i the i^{th} symbol in w. The function $\#_a(w)$ denotes the number of occurrences of $a \in \Sigma$ in w. Iff there exists words u, v such that $uv = w$, then $u\,(v)$ is a prefix (resp. suffix) of w.

Given two languages K, L over the same alphabet, set KL is $\{uv \mid u \in K, v \in L\}$. The repetition of a language is recursively defined to be $L^0 = \{\varepsilon\}, L^{i+1} = L^i L$. Similarly, the repetition of a word w is $w^0 = \varepsilon, w^{i+1} = ww^i$. We define the iteration of L to be $L^* = \bigcup_{n=0}^{\infty} L^n$, with $L^+ = \bigcup_{n=1}^{\infty} L^n$.

We model the inferred systems using DFAs:

Definition II.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 an alphabet, $\delta : Q \times \Sigma \to Q$ the 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 $\delta : Q \times \Sigma^* \to Q$, by inductively defining $\delta(q, \varepsilon) = q$ and $\delta(q, wa) = \delta(\delta(q, w), a)$, for $w \in \Sigma^*, a \in \Sigma$. Word w is accepted by DFA $A = (Q, \Sigma, \delta, q_0, F)$ iff state $\delta(q_0, w) \in F$. If w is not accepted by A, it is rejected. The language of A is the set $\mathcal{L}(A) = \{w \in \Sigma^* \mid A \text{ accepts } w\}$.

We allow a DFA to have a *partial* transition function, $\delta : Q \times \Sigma \hookrightarrow Q$, which is often denoted a *partial DFA*. Partial DFAs are completed by adding a state q_e, and extending the transition function such that undefined edges go to q_e. A partial DFA accepts a word w iff its completed counterpart does.

A DFA with a finite language is acyclic, and denoted a prefix tree acceptor (PTA). These are often used in a learning context, where the automata are *augmented* to ternary accepting conditions, i.e. $A = (Q, \Sigma, \delta, q_0, F, R)$. In that case, word w is accepted iff $\delta(q_0, w) \in F$, rejected iff $\delta(q_0, w) \in R$ and acceptance is unknown otherwise, with $\mathcal{L}_R(A)$ the set of rejected words.

Given two DFAs A_1, A_2 we define operations on the DFAs as operations on their language, such that $A_1 \cap A_2, A_1 \cup A_2, A_1 \setminus A_2$ are given as DFAs with language $\mathcal{L}(A_1) \cap \mathcal{L}(A_2), \mathcal{L}(A_1) \cup \mathcal{L}(A_2), \mathcal{L}(A_1) \setminus \mathcal{L}(A_2)$ respectively. In addition, we define synchronized parallel composition:

Definition II.2 ***(Synchronized 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 synchronized 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:

$(\delta_1(q_1, a), \delta_2(q_2, a))$	if $\delta_1(q_1, a), \delta_2(q_2, a)$ are both defined.
$(\delta_1(q_1, a), q_2)$	if $\delta_1(q_1, a)$ is defined, and $a \notin \Sigma_2$.
$(q_1, \delta_2(q_2, a))$	if $\delta_2(q_2, a)$ is defined, and $a \notin \Sigma_1$.
undefined	otherwise.

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 associative, i.e. $(A_1 \diamond A_2) \diamond A_3 \Leftrightarrow_L A_1 \diamond (A_2 \diamond A_3)$ and commutative, i.e. $A_1 \diamond A_2 \Leftrightarrow_L A_2 \diamond A_1$.

In order to reason about components of a synchronized composition, we define word projection:

Definition II.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:

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

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 II.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 II.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. Proofs are found in Appendix B. □

Proposition II.6 *Given DFAs A, A_1, A_2 such that $A = A_1 \| A_2$, over alphabets $\Sigma, \Sigma_1, \Sigma_2$, resp., then we have: $w \in \mathcal{L}(A) \Leftrightarrow \pi_{\Sigma_1}(w) \in \mathcal{L}(A_1) \wedge \pi_{\Sigma_2}(w) \in \mathcal{L}(A_2)$*

III. Literature Review

A. Model Inference

We review literature on active- and passive model learning.

Dana Angluin introduced learning from queries (Active Learning) [6], where the method aims to infer a target language using two queries: Membership queries tell whether a given word is in the target language. Equivalence queries tell whether a given language is equal to the target language and give a counter example otherwise.

The required queries are answered by the *oracle*, which relies on inspection of the source code [41]. Due to this, all behavior specified in the source code is learned, even when it is not observable in practice [3]. Hence, though results are exact, active learning is computationally intensive [7, 51].

E. Mark Gold defined the problem of learning from observations (Passive Learning) [30]. Consider a target language L in language class $\mathbb{L}$, and a (possibly infinite) sequence containing all, and only, words in L (a 'text sequence'). Gold defined that an algorithm can identify L in the limit, iff for every such sequence, there exists a natural number m such that for any $n \geq m$, the algorithm correctly identifies L from the first n words of the sequence.

Gold proved that any language class containing at least one language of infinite cardinality, cannot be identified in the limit from a text sequence. Instead, to identify such language, the sequence should contain all words, with annotation to denote which words are in L and which are outside L (counter examples) [5].

To give some intuition to why this is true, consider an infinite cardinality language L. The first n words of a text sequence are all in L. However, those words are also in the finite language L', spanning exactly those n words. The algorithm can never distinguish between L and L', and therefore cannot identify any language in the class containing infinite cardinality language L. Put differently, the algorithm cannot justify a generalization beyond the n words it observed, without any counter examples.

Gold's results imply that a DFA, which represents a regular language, cannot be identified from observations alone, as the class of regular language contains infinite cardinality languages.

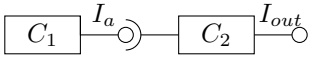

Fig. 1. A component-based system consisting of components C_1 and C_2.

State-of-the-art passive learning approaches, such as EDSM [16, 27], first build a PTA which only accepts the observation. Then it merges states to generalize, where loops might be introduced. Counter examples are used to prevent a merge when it would introduce a counter example in the language of the automaton.

Without counter examples, heuristics are used to prevent merging too aggressively [8, 16]. Examples are limiting the number of states in the resulting model, or allowing the state of the automaton to depend on at most the k most recently observed symbols (k-parsability).

Passive learning methods are generally evaluated on a small alphabet Σ, e.g. $\{a, b\}$ in [27], where a large part of behavior Σ^* is allowed. For software, the opposite holds: software uses a large alphabet where only sparse behavior is allowed. This makes it harder to make correct generalizations [13, 21, 48].

B. Component-Based Software

The component-based architecture is attributed to McIlroy [33]. It is composed of *components*, which are independent *units* of deployment and encapsulate functionality [42]. This allows for re-use, independent development and easier verification [46].

Szyperski argues that components must be without *observable state* [42], and that data should be part of the component 'instance', separated from the component 'plan'. Szyperski admits that this leads to discussion [18, 42]. Still, we use this in our approach in Section IV-B.

A component offers some pre-defined functionality, called a *service*, over an *interface*. The interface is an implementation of an *interface contract*, which specifies the allowed behavior on the interface. This allows for independent development of components [42, 46].

To offer a service over an interface, the component might require another interface. As example, take the system in Figure 1, where component C_1 offers interface I_a, which is required by component C_2 to offer interface I_{out}. With respect to I_a, we denote C_1 the server and C_2 the client.

A component-based architecture can be implemented in various domains, and on various levels of granularity, see e.g. [25, 29, 31, 37]. Though each systems comes with its own constraints, component-based systems are generally concurrent due to the independent deployment of components.

C. Modeling Systems and Behavior

Modeling Behavior: Automata rely on strings to model behavior as a sequential sequence of symbols. To model concurrent behavior, string theory is generalized to Mazurkiewicz Trace theory [32], which we introduce shortly.

Let a *dependency* D be any *symmetric, reflexive* relation. The domain of D is denoted by the *dependency alphabet* Σ_D, and the relation $I_D = (\Sigma_D \times \Sigma_D) \setminus D$ is the *independency* induced by D. *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 such that, in addition to the above, $u_1 \equiv_D u_2 \wedge v_1 \equiv_D v_2 \Rightarrow u_1 v_1 \equiv_D u_2 v_2$.

Equivalence classes over $\equiv_D$ are called *traces*, with the trace corresponding to a word w denoted $[w]_D$. This definition is lifted to languages, such that $[\mathcal{L}]_D = \{[w]_D \mid w \in \mathcal{L}\}$.

Concatenation of $[u]_D, [v]_D \in [\Sigma_D^*]_D$ is defined as $[u]_D[v]_D = [uv]_D$, where $[u]_D$ is prefix of $[w]_D = [uv]_D$. This definition extends language iteration to traces.

Given a set of traces T, $\lin T$ is the set $\{w \in \Sigma_D^* \mid [w]_D \in T\}$. Generally, for any string language $\mathcal{L}$, we have $L \subseteq \lin[\mathcal{L}]_D$. If $\mathcal{L} = \lin[\mathcal{L}]_D$, then $\mathcal{L}$ is *consistent* with D. If the language $\mathcal{L}(A)$ of automaton A is consistent with D, A has trace language $\mathcal{T}(A) = [\mathcal{L}(A)]_D$.

As example, take $D = \{a, b\}^2 \cup \{a, c\}^2$. Then, $I_D = \{(b, c), (c, b)\}$, which shows b and c can occur independently. The word $abbca$ is part of the trace $[abbca]_D = \{abbca, abcba, acbba\}$, which again shows 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 $w_{i-1} \sim_D w_i$ for $0 < i \leq n$.

In this text, we drop subscript D if it is clear from context. We rely on an additional result from Mazurkiewicz [32]:

Proposition III.1 *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.*

Component based systems and their concurrent behavior are intuitively visualized using message sequence charts [22]. We consider a subclass called Timed Message Sequence Charts [24].

An example TMSC is visualized in Figure 2, with Figure 2a the intuitive visualization of the TMSC, and Figure 2b a representation closer to the formal definition, which only captures events and dependencies.

The TMSC shows function executions, e.g. l, on components, e.g. C_1. Function executions are represented by a start- (closed-) and end event (open dot), and can nest on the same component. Communication between events is visualized as arrow, which represents a dependency. Events on a component are totally ordered, and hence dependent.

Formally, a TMSC is defined over a universe of events $\mathcal{E} = \mathcal{C} \times \mathcal{F} \times \mathbb{N}^+ \times \mathcal{S} \times \mathbb{R}_0^+$, where $\mathcal{C}$ is a finite set of component labels, $\mathcal{F}$ is a finite set of function labels, $\mathbb{N}^+$ is the set of positive numbers, $S = \{\uparrow, \downarrow\}$ and, $\mathbb{R}_0^+$ the set of non-negative real numbers.

An event $(c, f, i, s, t) \in \mathcal{E}$ describes the start of the i^{th} execution of function f on component c if $s = \uparrow$ at time t, or end if $s = \downarrow$. Functions $c(e), f(e), i(e), s(e), t(e)$ are used to refer to c, f, i, s, t of e respectively.

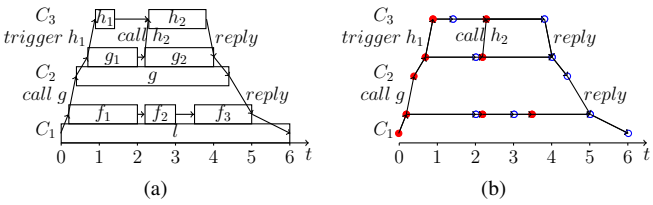

Fig. 2. Two visualizations of the same TMSC.

Then, a *Timed Message Sequence Chart* T is a three tuple $T = (E, \rightsquigarrow, m)$, with $E \subset \mathcal{E}$ a finite set of events, $\rightsquigarrow \subseteq E \times \mathbb{R}_0^+ \times E$ the set of dependencies, and $m : \rightsquigarrow \hookrightarrow \mathcal{M}$ a partial function that maps dependencies to messages.

We use $e_1 \overset{d}{\rightsquigarrow} e_2$ to denote $(e_1, d, e_2) \in \rightsquigarrow$, where d models the minimum time interval between e_1, e_2. We omit d if it is not relevant. With E_c we denote $\{e \in E \mid c(e) = c\}$.

A *path* is a sequence of events $e_0, e_1, \ldots, e_n$ with $n \geq 1$ where $e_0 \rightsquigarrow e_1 \rightsquigarrow \ldots \rightsquigarrow e_n$. We write $e_0 \rightsquigarrow^+ e_n$ iff there exists a path from e_0 to e_n. We write $e_0 \rightsquigarrow_c^+ e_n$ iff there is a path from e_0 to e_n with $e_0, \ldots, e_n \in E_c$.

A TMSC $T = (E, \rightsquigarrow, m)$ satisfies the following conditions:

1) Relation $\rightsquigarrow^+$ is a strict partial order on E and for $c \in \mathcal{C}$, relation $\rightsquigarrow_c^+$ is a strict total order on E_c.
2) If $(c, f, i, s, t) \in E$ and $(c, f, i, s, t') \in E$, then $t = t'$.
3) If $c.f(i)^\uparrow \in E$, then for all $0 < j < i$ an event $c.f(j)^\uparrow \in E$ exists such that $c.f(j)^\uparrow \rightsquigarrow_c^+ c.f(i)^\uparrow$.
4) Event $c.f(i)^\uparrow \in E$ iff $c.f(i)^\downarrow \in E$, and for all $c.f(i)^\uparrow, c.f(i)^\downarrow \in E, c.f(i)^\uparrow \rightsquigarrow^+ c.f(i)^\downarrow$.
5) For all $c.f(n)^\uparrow, c.f(n)^\downarrow, c.g(m)^\uparrow, c.g(m)^\downarrow \in E$, if $c.f(n)^\uparrow \rightsquigarrow_c^+ c.g(m)^\uparrow$ and $c.g(m)^\uparrow \rightsquigarrow_c^+ c.f(n)^\downarrow$ then $c.g(m)^\downarrow \rightsquigarrow_c^+ c.f(n)^\downarrow$.
6) For all $e_1, e_2 \in E, d \in \mathbb{R}_0^+$, if $e_1 \overset{d}{\rightsquigarrow} e_2$, then $t(e_2) \geq t(e_1) + d$.

The first condition ensures that dependencies are acyclic. Due to this, and the finite size of E, a TMSC represents a finite execution of the system. The second condition prescribes that an event can be uniquely determined by its component, function, index and type. Therefore we refer to an event $(c, f, i, s, t) \in E$ as $c.f(i)^s$. The third condition ensures that events are properly indexed. We define $\sigma : \mathcal{E} \to \Sigma$ as $\sigma(e) = \{(c(e), f(e), s(e)\}$, such that $\mathcal{E}$ is a multiset over Σ, with index $i(e)$ and timing annotation. The fourth condition ensures that every end event $c.f(i)^\downarrow$ is preceded by a start event of the corresponding function execution, $c.f(i)^\uparrow$. Therefore, we denote $(c.f(i)^\uparrow, c.f(i)^\downarrow)$ as function execution $c.f(i)$. The fifth condition ensures proper nesting of function executions, yielding well-formed call stacks. If function executions $c.f(n), c.g(m)$, nest as defined in this condition, this is denoted $c.f(n) \succ c.g(m)$. Execution $c.f(n)$ is a root execution iff there is no $c.g(m)$ such that $c.g(m) \succ c.f(n)$. Finally, the sixth condition ensures the timing constraints prescribed by $\rightsquigarrow$ are met in the TMSC.

The Mazurkiewicz dependency corresponding to a TMSC modulo timing annotation, $t(e)$, is the symmetric, reflexive

closure of $\rightsquigarrow$ [35]. If the context requires us to, we distinguish between the two types of dependencies explicitly. For brevity, we assume the universe of events $\mathcal{E}$ to be defined implicitly when dealing with a TMSC.

Modeling Systems: As component-based systems are compositions of components, we consider modeling techniques to preserve the separation of components as a model of the system is inferred. In addition to synchronous composition, we introduce asynchronous composition [2, 12].

For an asynchronous composition of DFAs $A_1, \ldots, A_2$, we assume the following: Component A_i has alphabet Σ_i, partitioned in sending-, receiving- and internal-symbols, $\Sigma_i^!, \Sigma_i^?, \Sigma_i^\tau$ resp. Each message has a unique sender, $\Sigma_i^! \cap \Sigma_j^! = \emptyset$ for $i \neq j$, a unique receiver, $\Sigma_i^? \cap \Sigma_j^? = \emptyset$ for $i \neq j$, and this receiver is assumed to exist, $a \in \Sigma_i^! \Rightarrow \exists_{j \neq i} : a \in \Sigma_j^?$. 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 M_i, which represent either a FIFO buffer or a bag buffer. FIFO buffers are modeled as a list of symbols, with ε the empty buffer, where messages are added to the tail of the list and consumed from the head of the list. Then, the asynchronous composition using FIFO buffers is defined as:

Definition III.2 Consider n DFAs $A_1, \ldots, A_n$, with $A_i = (Q_i, \Sigma_i^{!,?,\tau}, \delta_i, q_{0,i}, F_i)$. The asynchronous composition A using FIFOs of $A_1, \ldots, A_n$, denoted $A = \|_i(A_i \| M_i)$, is given as (possibly infinite) state machine:

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

with $Q = Q_1 \times (\Sigma_1^?)^* \times \ldots Q_n \times (\Sigma_n^?)^*$ and $\delta \subseteq Q \times \Sigma \times Q$ such that for $q = (q_1, c_1, \ldots, q_n, c_n)$ and $q' = (q'_1, c'_1, \ldots, q'_n, c'_n)$ 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) $M'_j = M_j a$, (iv) $\forall_k : k \neq j \Rightarrow M_k = M'_k$ and (v) $\forall_k : k \neq i \Rightarrow q'_k = q_k$.

(consume) $(q, a?, q') \in \delta$ if $\exists_i$: (i) $a \in \Sigma_i^?$, (ii) $(q_i, ?a, q'_i) \in \delta_i$, (iii) $M_i = aM'_i$, (iv) $\forall_k : k \neq i \Rightarrow M_k = M'_k$ and (v) $\forall_k : k \neq i \Rightarrow q'_k = q_k$.

(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) $\forall_k : M_k = M'_k$ and (iv) $\forall_k : k \neq i \Rightarrow q'_k = q_k$.

Bag buffers are defined as multiset over $\Sigma_i^?$. To use bag- instead of FIFO buffers, we change: the states corresponding to a single buffer to multiset $\Sigma_i^?$, the send rule clause (iii) to $M'_j = M_j \cup \{a\}$, the consume rule clause (iii) to $a \in M_j, M'_j = M_j - \{a\}$.

With unbounded buffers, asynchronous compositions have infinite state spaces and are Turing expressive. When channels are bounded, the communication models can be represented by a DFA [36], and hence the composition is equal to a DFA. We bound buffer M_i to k places, denoted M_i^k, by adding the requirement $|M_j| < k$ (s.t. $|M'_j| = k$) to the send rule.

We do not discuss the construction of a DFA B_i^k representing M_i^k, as it follows from the definition above. Using B_i^k, synchronous composition $A_i \| B_i^k$ is equivalent to asynchronous composition $A_i \| M_i^k$, where we note that a synchronous composition requires us to differentiate $a!$ and $a?$ to prevent synchronization.

The question whether there is a bound on the buffers such that every word on the unbounded composition can be computed on the bounded composition is called *boundedness* and is generally undecidable [17]. However, if the asynchronous composition is 'deadlock free', i.e. a final state can be reached from every reachable state [26], then it is decidable for a given b whether the asynchronous composition is bounded to b.

We rely on two tools to model and analyse automata: CIF is an interchange format defined in [43] which incorporates finite automata, timed automata [4], and data expressions. As data types in CIF are limited in range, state-spaces are finite in size. mCRL2 is a process algebra with fast support for state-space exploration and verification [19]. Translations from CIF to mCRL2 are available [43].

IV. Inferring Component Models

In Section III-B we concluded that component-based systems are composed of components, which in turn offer a collection of (mutually independent) services. We use this two-layered architecture to break down the inference problem. We first discuss decomposition in components, followed by decomposition in services.

Throughout this section we assume the system is a DFA $A = (Q, \Sigma, \delta, q_0, F)$ synchronously composed of n DFAs, $A = A_1 \| \ldots \| A_n$, and that observation $W \subseteq \mathcal{L}(A)$ is available to infer an approximation A' of A. Hence, we assume communication is modeled by synchronized events, which we revisit in Section V. We use subscripts and accents to refer to component- and inferred instances resp.

A. Decomposition in Components

To decompose the system in components we assume component alphabets Σ_i are known a-priori and can be used to project the observation on each component. With these, we prove composition A' of inferred components generalizes to traces, while maintaining under-generalization, i.e. $\mathcal{L}(A') \subseteq \mathcal{L}(A)$, in Propositions IV.2 and IV.3 resp.

The methods discussed in Section III used an algorithm $PTA(W)$ to obtain a DFA A' with $\mathcal{L}(A') = W$ [20, Alg 1]. We show our results for this algorithm, by learning components A'_i instead, using $PTA(\pi_{\Sigma_i}(W))$. As then $\mathcal{L}(A'_i) = \pi_{\Sigma_i}(W)$ for each i, and by Proposition II.6, we know that composition $A' = A'_1 \| \ldots \| A'_n$ still accepts W.

We use trace theory to show how DFA A' generalizes:

Proposition IV.1 *Given a dependency $D = \bigcup_{i=1}^n (\Sigma_i^2)$ and two words $u, v \in \Sigma^*$, $\Sigma = \bigcup_{i=1}^n \Sigma_i$, we have:*

$$u \equiv_D v \Leftrightarrow \forall_i : \pi_{\Sigma_i}(u) = \pi_{\Sigma_i}(v)$$

Proposition IV.2 *Consider a synchronous composition $A = A_1 \| \ldots \| A_n$, then $\mathcal{L}(A) = \text{lin}[\mathcal{L}(A)]_D$ for $D = \cup_i (\Sigma_i^2)$.*

Proposition IV.2 shows that learning a PTA per component generalizes $\mathcal{L}(A')$ from W to $[W]_D$. As the same holds for

A, we conclude A' is an under-generalization. Generally, it is sufficient to show this for the components only:

Proposition IV.3 *If $\mathcal{L}(A'_1) \subseteq \mathcal{L}(A_1)$ and $\mathcal{L}(A'_2) \subseteq \mathcal{L}(A_2)$, then $\mathcal{L}(A'_1 \| A'_2) \subseteq \mathcal{L}(A_1 \| A_2)$.*

For component decomposition, we can describe $\mathcal{L}(A')$, show it be an under-generalization, $\mathcal{L}(A') \subseteq \mathcal{L}(A)$ and prove it to be robust under additional observations:

Theorem IV.4 *Consider DFA $A = \|_i A_i$ and a set $W \subseteq \mathcal{L}(A)$. DFA $A' = \|_i A'_i$, with $A'_i = PTA(\pi_{\Sigma_i}(W))$ has $\mathcal{L}(A') = [W]_D$ for $D = \cup_i(\Sigma_i^2)$ and $\mathcal{L}(A') \subseteq \mathcal{L}(A)$.*

Theorem IV.5 *Consider DFA $A' = \|_i A'_i$, with $A'_i = PTA(\pi_{\Sigma_i}(U)$ and U an observation. DFA A'', composed and obtained similarly from observation V, $U \subseteq V$, has $\mathcal{L}(A') \subseteq \mathcal{L}(A'')$.*

The results show that component decomposition generalizes to trace-equivalence and does so correctly. However, the allowed behavior is still limited to words of length equal to some observation, as trace equivalence preserves word length.

B. Decomposition in Tasks

To generalize component behavior to longer and (possibly) shorter executions, we further decompose components in the collection of services offered. In Section III-B, we discussed how components offer services based on some request, that service executions carry no observable state and are therefore mutually independent. We model these as tasks for now, which we assume can be distinguished from observations:

Definition IV.6 ***(Task)*** Let Σ be an alphabet partitioned in function- and return-alphabets Σ^f, Σ^r resp., denoted $\Sigma^{f,r}$. Word $w \in \Sigma^*$ is a *task* iff $w = va$ with $v \in \Sigma^{f*}$ and $a \in \Sigma^r$.

We model DFAs without observable state beyond tasks:

Definition IV.7 ***(Task DFA)*** A DFA $A = (Q, \Sigma^{f,r}, \delta, q_0, F)$ is a *Task DFA* iff $F = \{q_0\}$ and $\forall_{q \in Q} : a \in \Sigma^r \Leftrightarrow \delta(q, a) = q_0$.

From the definition, we see that after computing a task, a Task DFA A returns to the initial, accepting, state. Hence, language $\mathcal{L}(A)$ is the *iteration* of some set T of tasks, $\mathcal{L}(A) = T^*$. Elements of T^* are *task sequences*:

Definition IV.8 ***(Task Sequence)*** Given alphabet $\Sigma^{f,r}$, a word $w \in \Sigma^*$ is a *Task sequence* iff $w = w_1 \dots w_n$ with each $w_i, 1 \leq i \leq n$, a task w.r.t. $\Sigma^{f,r}$. The set $\{w_i\}$ is the *task set* corresponding to w and is denoted $T(w)$.

The definition is lifted to sets of strings, $T(W) = \bigcup_{w \in W} T(w)$, with $T(\mathcal{L}(A))$ written as $\mathcal{L}_T(A)$. Hence, for a Task DFA A we have $\mathcal{L}(A) = \mathcal{L}_T(A)^*$.

A composition of Task DFAs is not generally a Task DFA. If a symbol is only in return alphabets, transitions over that symbol might not go to the accepting state (e.g. aba**c** over Figure 3a). Yet, by Proposition II.5, we know that any word in the language of A represents executions of tasks on components. Therefore we extent task words to traces:

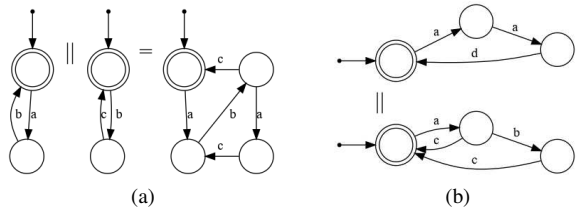

Fig. 3. (a) Two TDFAs and their composition. (b) Two TDFAs.

Definition IV.9 ***(Task Trace)*** Consider a dependency $D = \bigcup \Sigma_i^2$ and partitions $\Sigma_i^{f,r}$. A trace $[t] \in [\Sigma_D^*]$ is a *task sequence (trace)* iff $\forall_i : \pi_{\Sigma_i}(t)$ is a task sequence w.r.t. $\Sigma_i^{f,r}$. Iff, additionally, there are no task sequences $[u], [v] \in [\Sigma_D^+]$ such that $[uv] = [t]$, then $[t]$ is a *task (trace)* and $T([t])$ is given as $\{[t_j] \mid [t_1 ... t_m] = [t]$ and $[t_j]$ a task for $1 \leq j \leq m\}$

Proposition IV.10 *Given a composition A of n task DFAs, $A = \|_{i=0}^n A_i$ and $D = \bigcup \Sigma_i^2$, then $[t] \in \mathcal{T}(A)$ is a task sequence, $[t] \in \mathcal{T}(A) \Leftrightarrow T([t]) \subseteq \mathcal{T}(A)$ and $\mathcal{T}(A) = T(\mathcal{T}(A))^*$.*

Considering the DFAs in Figure 3a, Mazurkiewicz traces allow us to define $\mathcal{T}(A) = T([abacbc])^*$, as we have $T([abacbc]) = \{[abc]\}$ while $T(abacbc) = \{abacbc\}$.

With these definitions it is straightforward to infer a Task DFA from an observation W: First, we collect the observed tasks on a component, $T(\pi_{\Sigma_i}(W))$, and then construct DFA A'_i which accepts the iteration of these tasks, i.e. $\mathcal{L}(A'_i) = T(\pi_{\Sigma_i}(W))^*$. The algorithm, $TDFA(W, \Sigma_i^{f,r})$, is given in Appendix A. We prove the algorithm to be correct, and A'_i to be a Task DFA and an under-generalization:

Proposition IV.11 *Consider DFA A composed of Task DFAs, $A = \|_i A_i$ and a set $W \subseteq \mathcal{L}(A)$. Algorithm $TDFA(W, \Sigma_i^{f,r})$, returns a Task DFA A'_i such that $\mathcal{L}(A'_i) = T(\pi_{\Sigma_i}(W))^*$ and $\mathcal{L}(A'_i) \subseteq \mathcal{L}(A_i)$.*

Proposition IV.3 shows that composition $A' = \|_i A'_i$ is an under-generalization. From Propositions II.6 and IV.11 we see that $W \in \mathcal{L}(A')$. Additionally, we prove that $TDFA$ is robust under additional observations:

Theorem IV.12 *Consider DFA A' composed of Task DFAs, $A = \|_i A'_i$, with $A'_i = TDFA(U, \Sigma_i^{f,r})$ and U an observation. DFA A'', composed and obtained similarly from observation V, $U \subseteq V$, has $\mathcal{L}(A') \subseteq \mathcal{L}(A'')$.*

To characterise generalizations made, Proposition IV.2 shows that A' generalizes to trace equivalence, $[W] \in \mathcal{T}(A')$. Proposition IV.10 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 $T([W])^*$. Consider the DFAs in Figure 3b, which are inferred from $w = abcacd$. Here, $[w]$ is a task, as aad cannot be split, yet the composition also accepts $w' = abcabcd$ and $w'' = acacd$, outside $T([W])^*$. If a component accepts multiple tasks (here ac, abc) which synchronize equally, the tasks can be interchanged. These generalizations make characterising the full generalization an open problem.

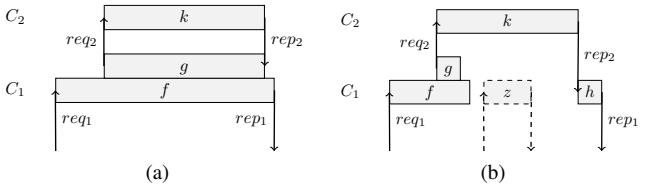

Fig. 4. Two ways of handling a message req_1. Figure (a) Message req_1 is handled synchronously through task fg. Figure (b) Message req_1 is handled asynchronously through tasks fg and h.

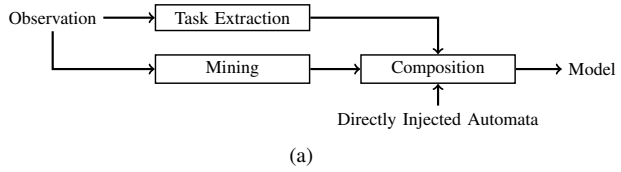

Fig. 5. Approach to add additional information to model inference approach.

Concluding, we decomposed component behavior in a sequence of mutually independent tasks. This allows us to infer models which generalize to longer, and possibly shorter, executions covering the same tasks. We have proven these generalizations to be valid for the original system.

To infer the complete system A, the task sets of its components, $\mathcal{L}_T(A_i)$ should at least be finite, as we enumerate these tasks in approximations A'_i. If all tasks of A are observed, the full system is inferred.

C. Stateful Task Inference

To infer a component as a collection of tasks, we assumed the tasks to be mutually independent. This does not always hold in practice, as visualized by an example in Figure 4, using a TMSC as introduced in Section III-C. To infer a model, a full function stack is considered a task.

In Figure 4a, g sends a request and receives a reply. To allow for other computations during req_1, C_1 collects the reply in a separate task, h, in Figure 4b, and req_1 is *asynchronously handled*. Now, h can only be initiated once fg is executed, which is ensured by C_2 but not captured in the model for C_1.

We conclude that we cannot model a service as a single task, but rather as a, possibly singleton, sequence. Put differently, we should infer a component model A which allows a subset of $\mathcal{L}_T(A)^*$. To connect tasks, we require additional information.

Injecting Additional Information: We give a structural approach to improve the inferred models based on injection of domain knowledge. Given that many varieties of component-based systems exist, this allows the approach to be adopted to the target system, rather than being tailored to our case of Section VI. The approach is based on Beschastnikh's InvariMint [10] and visualized in Figure 5.

For the approach we compose the results of the task extraction with additional, small, automata which show explicitly which behavior we add, constrain or remove. Composition is done using the operators introduced in Section II: union, intersection, synchronized composition and (symmetrical) difference. The automata are either directly injected, or obtained using a procedure on the observation, denoted a miner conform with [10].

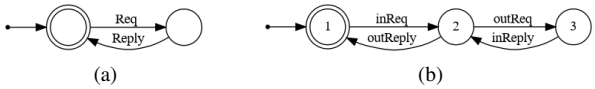

Fig. 6. Automata used to model domain properties: (a) B_1 (b) B_2.

We demonstrate the approach using two examples. A first characteristic of component-based systems is that a service is requested by one request and returns one reply, which we model with automaton B_1 in Figure 6a. DFA B_1 runs over $\{Req, Reply\}$, which represent a request and reply resp.

To enforce the property on the system, we should compose the inferred Task DFAs, A'_i, with B_1 for every request, reply in the system. To do so, we define substitution:

Definition IV.13 ***(Substitution)*** Given a DFA $A = (Q, \Sigma, \delta, q_0, F)$, a symbol $a \in \Sigma$ and a symbol b, the substitution of a with b in A, denoted $A[a := b]$, is defined as $A[a := b] = (Q, (\Sigma \setminus \{a\}) \cup \{b\}, \delta[a := b], q_0, F)$, with $\delta[a := b]$ defined as:

$$\delta[a := b](q, c) = \begin{cases} \delta(q, a) & \text{if } c = b \text{ and } \delta(q, a) \text{ is defined.} \\ \delta(q, c) & \text{otherwise.} \end{cases}$$

We note that $A[a := a] = A$, which is an identity operation. However, $A[a := b][b := a] \neq A$ in general, as for a state q in A with transitions over both a and b, the transition over b will be lost. The order of substitutions is important, and are applied left to right.

With the substitution operator, we can use B_1 as a 'blueprint' for the domain knowledge, which captures a relation between domain elements. Its use in the composition function allows us to specify to which domain elements the blueprint applies:

$$A'' = (\|_i A'_i) \parallel (\|_{a \in reqs} B_1[Req := a][Reply := R(a)]) \quad (1)$$

where $reqs$ is the set of requests, and $R : \Sigma \to \Sigma$ is a map relating requests to their corresponding replies. We assume both $reqs$ and R are available from domain knowledge.

Proposition II.6 shows that for any $a \in reqs$ we have $\pi_{\{a, R(a)\}}(\mathcal{L}(A'')) = \{a.R(a)^n \mid n \in \mathbb{N}\}$. This specification corresponds to the assumption given informally, with the addition that only one request a can be outstanding.

The second example property enforces that while executing a service, any nested request should be returned before finishing the service. It is modeled using DFA B_2 in Figure 6b, which captures incoming- and outgoing requests ($inReq$, $outReq$ resp.) as well as incoming- and outgoing replies ($inReply$, $outReply$ resp.).

Upon requesting a service, the incoming request puts DFA B_2 in state 2. If during this service, a request is sent, this request has to be met with a reply to get DFA B_1 out of state 3 and allow the original service to reply.

Two issues are at play here. First, as we substitute over two sets of requests, the number of automata grows considerably. Many automata might not be relevant, as

during a given incoming requests, not all outgoing requests can occur. This makes the composition unnecessarily large.

Secondly, it is possible that two requests are handled asynchronously and interleave. Then, the outgoing request of one service might be wrongly attributed to the other service. This requires us to actually constrain the composition to those nested requests that are actually relevant, and assume they nest consistently.

To cover both issues, it is essential to use knowledge of the deployment to formulate a composition function which only captures incoming-, outgoing-request pairs which can nest. Whether an incoming and outgoing request can nest can also be determined from the observation, if we assume that the outgoing request is placed in the same task as the incoming request is received. In that case, a procedure on the observation can be used to select which incoming-, outgoing-request pairs are relevant, which is denoted a miner in Figure 5. Consider that in this case, the miner assesses which instances run the risk of over-generalizing, and the injected automaton constrains the actual over-generalization from occurring.

The examples covered show that additional information can be added explicitly and in a straightforward manner. The substitution operator allows assumptions to be bound to a wide class of instances. Note that this approach does not infer any state by itself, nor does it infer relations between these states, as all information is supplied.

V. Modeling Inter-Component Communication

Whereas in Section IV we inferred systems communicating synchronously, we now discuss asynchronous systems.

Throughout the section, we assume the target system is a DFA A, asynchronously composed of components A_i and buffers M_i each bounded to b_i, $A = \|_i(A_i \| M_i^{b_i})$. We assume A_i has alphabet $\Sigma_i^{!,?,\tau}$ as discussed in Section III-C, and M_i has alphabet $\Sigma_{M_i} = \Sigma_{M_i}^{!} \cup \Sigma_{M_i}^{?}$. Clearly, $\Sigma_{M_i}^{?} = \Sigma_i^{?}$ and $\Sigma_{M_i}^{!} = \{a! \mid a? \in \Sigma_i^{?}\}$. We base our inference on observation $W \subseteq \mathcal{L}(A)$, and specifically, a symbol $a \in \Sigma$ along $w \in W$, such that $w = uav$, $u, v, \in \Sigma^*$.

We infer A' by inferring A_i' using the techniques of Section IV. In addition, we model each buffer as DFA $B_i'^{b_i}$, as discussed in Section III-C, for which we require the buffer type and bound. A lower bound of buffer M_i is inferred from W, by considering the required buffer space along each word, that is $b_{i,w} = \max_{a \in w} |\pi_{\Sigma_i^!}(ua)| - |\pi_{\Sigma_i^?}(ua)|$ with u the prefix of w up to a. The overall lowerbound of M_i is then $b_i = \max_{w \in W} b_{i,w}$. Assuming the buffer type is known, A' is given as synchronous composition $A' = \|_i(A_i' \| B_i'^{b_i})$.

Just as for the synchronous case, we prove A' is an under-generalization trace conform with W, robust under additional observations:

Proposition V.1 *Given DFA $A = \|_i(A_i \| M_i)$ with M_i a FIFO (bag), and set $W \subseteq \mathcal{L}(A)$, then DFA $A' = \|_i(A_i' \| B_i'^{b_i})$, with $A_i' = TDFA(W, \Sigma_i^{?,!})$ and $B_i'^{b_i}$ a FIFO (bag) with $b_i = \max_{w \in W} b_{i,w}$, has $W \subseteq \mathcal{L}(A') \subseteq \mathcal{L}(A)$. DFA A'', similarly obtained from observation $V, W \subseteq V$, has $\mathcal{L}(A') \subseteq \mathcal{L}(A'')$.*

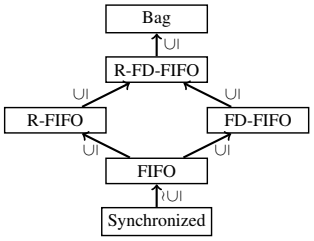

Fig. 7. Language inclusion for buffers.

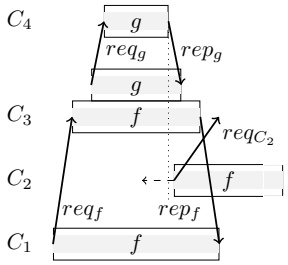

Fig. 8. Limited commutations.

A remaining issue is that we require to know a-priori what kind of buffers are used in our system. Conversely, we can use trace conformance as an experiment to assess the communication structure. We discuss various alternatives, shown in Figure 7. Given that we check our observation for trace conformance to these models, we include the language inclusion relation between buffer types in our discussion.

Synchronous communication is included in FIFO communication, if we substitute a synchronized event a by two events '$a!$ followed by $a?$'. This constitutes a send followed immediately by a receive. Similarly, the language of a bag buffer is a superset of that of a FIFO buffer, as in Definition III.2 a FIFO receive requires a symbol to be at the head of the buffer queue ($M_j' = aM_j$), whereas a bag buffer requires it to be anywhere in the buffer ($a \in M_j$).

Using a FIFO buffer in a component-based system leads to an issue shown in Figure 8. Would in this situation req_{C_2} be sent before rep_g, a strict FIFO order would enforce reception of the former before the latter, while the Task DFA itself has to handle the latter before starting a new service with the former.

To solve the issue, we partition the function alphabet in a call alphabet and a reduced function alphabet, such that any task starts with a symbol from the call alphabet. Then, we can split the buffer in a FIFO buffer handling symbols from the call alphabet, which start a new task, and a bag buffer handling symbols from the reduced function and return alphabet, which are required to finish the task. We denote this an R-FIFO.

Another issue, highlighted in [2, 12], is that travel time variations might change the order of incoming messages from different components. To abstract from this, the FIFO buffer for component i can be composed of $n-1$ FIFO buffers, one for each peer component, also called Full Duplex and here denoted FD-FIFO. We use subscript i, j to denote the link from A_i to A_j. The R-FD-FIFO assesses both issues, and hence has two buffers per component pair, totaling in up to $2(n-1)$ buffers per component.

From Proposition IV.2, we know the inferred system $A' = \|_i(A_i' \| B_i'^{b_i})$ is consistent with $D = \cup_i(\Sigma_i^2 \cup \Sigma_{M_i}^2)$. As $\Sigma_{M_i}^2 \subseteq D$, we cannot commute a send- and receive action, e.g. $C_2.f^{\uparrow}$ and $C_3.g^{\downarrow}$ in Figure 8, even though we know this is allowed.

To better characterise the inferred generalizations, we give alternative formulations for D. We only consider the buffers, and hence maintain $\Sigma_i^2 \subseteq D$ for any component A_i'.

1) FD-FIFO: We first give the definition from [26, 35].

Dependency D for $B_{i,j}'^{b}$ is defined over $\Sigma_D = \Sigma_{M_{i,j}} \times \{0, \ldots, b-1\}$. Word $w = uav$ is mapped to Σ_D, by mapping

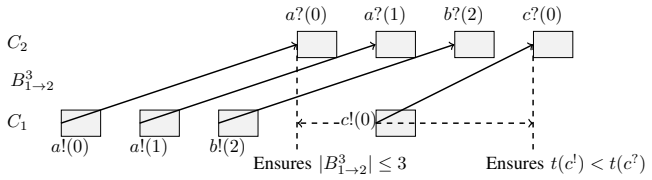

Fig. 9. Commutation bounds for $c!$ given by Mazurkiewicz dependency D.

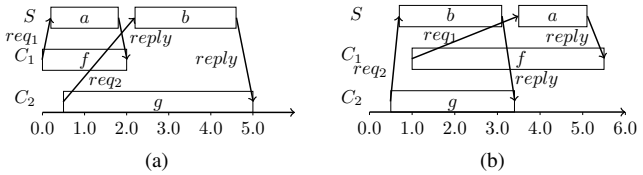

Fig. 10. Two clients C_1, C_2 making concurrent requests to a server S, shown under two different timings.

a to (a, n) with $n = |\pi_{\Sigma^!_{M,i,j}}(u)| \mod b$ if $a \in \Sigma^!_{M,i,j}$, and $n = |\pi_{\Sigma^?_{M,i,j}}(u)| \mod b$ for $a \in \Sigma^?_{M,i,j}$.

Two symbols $(a, n), (b, m) \in \Sigma_D$ are related in D, $(a, n)D(b, m)$, iff (i) $n = m$, *or*, (ii) a and b are on the same component: $a, b \in \Sigma_i$ or $a, b \in \Sigma_j$.

By preventing commutation of send (resp. receive) actions, requirement (ii) ensures FIFO ordering is maintained over trace equivalence. As communicating events have the same annotation, they cannot commute by requirement (i), ensuring the send event precedes the receive event. As we annotate modulo b, equally annotated events which do not communicate are b buffer places apart. As these cannot commute, requirement (i) prevents buffer overflow. This is visualized in Figure 9.

2) FIFO Buffer: We extend the work of [26, 35] to a regular FIFO buffer B'^b_i, where we rely on the same annotation, though with alphabets $\Sigma^!_{M_i}, \Sigma^?_{M_i}$ which are larger for a FIFO receiving from all peers.

Requirement (i) is unchanged to prevent overflow and ensure sending precedes receiving. Requirement (ii) has to change to $a, b \in \Sigma^!_{M_i}$ or $a, b \in \Sigma^?_{M_i}$, to prevent send (resp. receive) actions of commuting, thereby breaking the FIFO ordering, as the FIFO serves multiple sending components. Note that this still allows and a send and a receive symbol to commute, withstanding requirement (i).

3) Bag buffer: Extending the dependency to bags can be done using the same annotation though with a larger alphabet.

A bag buffer does not enforce a FIFO ordering. Hence, we drop requirement (ii). Requirement (i) is still required to prevent overflow and ensure sending precedes receiving.

We discussed Mazurkiewicz Dependencies with which Bag, FIFO and full duplex FIFO buffers are consistent. Given that the R-FIFO is a composition of a bag and a FIFO, the Mazurkiewicz dependency for this buffer follows from the union of the dependencies of both parts.

The formulations given for D, corresponding to each buffer model, ensure that whenever a word w is accepted by a buffer $B^{b_i}_i$, i.e. $\pi_{\Sigma_{B_i}}(w) \in \mathcal{L}(B^{b_i}_i)$, we have $[\pi_{\Sigma_{B_i}}(w)]_D \in \mathcal{T}(B^{b_i}_i)$. Similarly, for an inferred component A'_i, we know $[w]_D$ is accepted by A'_i too. Therefore, for inferred composition A', we have $[w]_D \subseteq \mathcal{T}(A')$ by Proposition II.5. This shows the characterization of traces under the weaker dependencies introduced in this section.

VI. Model Inference in Practice

We demonstrate the inference 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. We infer a model of the exposure subsystem, which exposes each field (die) on a wafer.

System Characteristics & Assumptions

A TMSC is used as observation of the system, spanning approx. 100k events over 33 components. This implies the system is a composition of sequential components bearing nested, fully traced, function executions. The TMSC is obtained using methods described in [24].

Message dependencies in the TMSC represent either a request, reply, trigger or notification. Requests and trigger request a service from a component, whereas a request is answered by a reply, and a trigger is unanswered. Notifications are sent by a server to a client during service execution, to notify the client. Each service has one incoming request (or trigger) and one (resp. zero) outgoing replies.

Given that components are sequential, requests can only be handled once the component is idle, and prior requests are finished, that is, requests are handled non-preemptively. Hence, the order in which services are executed depend on the order in which messages are received, as visualized in Figure 10. We assume requests and triggers are handled at the start of a root execution, i.e. a function execution which is not nested.

A. Stateless Task Extraction

Though a TMSC describes asynchronous communication, we first apply synchronous task extraction. From Section V, we know a synchronous composition has the most limited behavior, and hence the smallest state space. The resulting model can therefore be analyzed for issues more easily, while these issues apply to both the synchronous and the asynchronous case.

To apply synchronous inference, we require an observation W and alphabets $\Sigma^{f,r}_i$. We obtain these from as follows:

Function σ maps TMSC events to alphabet Σ as discussed in Section III-C. We define alternative mapping $\sigma'(e) = \sigma(e) \cup \{\sigma(e') \mid e \rightsquigarrow e' \text{ or } e' \rightsquigarrow e,\ c(e) \neq c(e')\}$, to ensure communicating events synchronize. We extend σ to $\sigma : 2^{\mathcal{E}} \to \Sigma^*$, such that $\sigma(E) = \sigma(e)\Sigma(E \setminus \{e\})$, where e is the minimal event according to $\rightsquigarrow$ and $t(e)$.

Given that the order of root executions depends on communication, we define a root execution to be a task. Hence, $\Sigma^r_c = \{\sigma(e) \mid (e', e) \text{ a root execution on } c\}$.

With partition $\Sigma^{r,f}_c$ for each component, and $w = \sigma(E)$ as observation, we apply task extraction. Figure 11 shows the size of the resulting model, in states, for our case study. Clearly, for most components, the model is two orders of magnitude

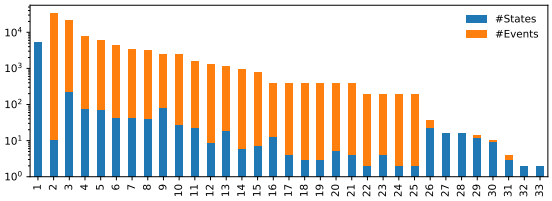

Fig. 11. Number of states in the task extracted model of each component, compared to the number of events in the observation of this component.

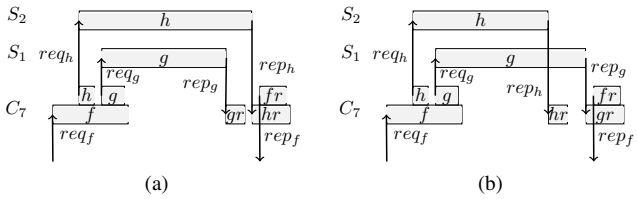

Fig. 12. Component C_7 requires both rep_g and rep_h to reply to its client.

more compact than the observation itself, showing repetitive execution of root executions in these components.

Component 1 has a model of the same size as the original observation, because it spans a single, 5000 event, task. This component orchestrates the wafer exposure. For components 26–33, the small reduction is due to the small observation.

B. Model Analyses & Reductions

We analyze the resulting models to assess whether these are indeed stateless across tasks and if not, which knowledge must be added. Though we cannot guarantee it to be stateless across tasks, we aim to give general handles on how to find issues.

First, we analyze the structure of the resulting model. For instance, the number of tasks is interesting to consider. If the number of tasks is large, but the reduction is small, this could imply that there is a lot of variation in the execution of a task, and therefore execution depends on some state which is not captured. This is especially the case when tasks share a common prefix.

Another structural issue to look out for is tasks starting without request, even without incoming message. Given that services are assumed to start with a request, this indicates that this task relies on another task to initiate the service.

Secondly, we analyze the behavior of the model by exploring its state space. We use mCRL2 [19] as it is very efficient in exploring state spaces. As mCRL2 does not distinguish accepting states, we apply a prefix closure and partial minimization [9] to the resulting DFAs, to match the semantics of mCRL2. This is a post-processing step, as domain injection relies on accepting states to correct models. We utilize CIF for the conversion [43].

Given that the size of the resulting state space is still too large for the tools to explore, we discuss two reductions.

We concluded that component 1, C_1, still spans > 5000 states, in a single task w. To reduce the size of this component, we analyze w for repetitions and reduce these repetitions using [38]. In other words, we aim to find $x, y, z \in \Sigma^*_{C_1}$ such that $w = xy^n z$ for some n. Then, we can create a reduced model $A_{C'_1}$ with language $\mathcal{L}(A_{C'_1}) = (xyz)^*$, reducing the number of states by approximately $|y|^{n-1}$ to 600.

As a second reduction, we remove DFA transitions which do not communicate and originate from a state which has a single successor state, i.e. does not allow for a choice in the process. Hence, we apply the following process algebra axiom: $a.\tau.b = a.b$, where τ is an action which does not synchronize [34]. For asynchronous composition, these symbols are in Σ^τ. Examples in Figure 12, are $C_7.g^\downarrow$ and $C_7.f^\downarrow$. This reduces C_1 from 600 to approx. 300 states. All other components are reduced as well, and therefore exploring the resulting state space is considerably easier.

With the two reductions discussed, we explore the state space of the composition and analyze it for deadlocks: states in the partial model from which no outgoing transition is present. Given that Task DFAs are strongly connected, we do not expect deadlocks. If a deadlock arises, it is due to a synchronizing symbol, which counterpart cannot be reached.

Any deadlocks found are either an issue in the system itself, an issue arising from applying synchronous communication to an asynchronously communicating system, or issues arising due to the stateless across tasks assumption. We mainly uncover the third type, which we use to improve the model. Only a single deadlock of the second type has been found, which has been dealt with pragmatically to continue searching for other deadlocks.

C. Model Improvements: Adding State

We discuss some issues found using state space exploration.

The first type of issue is shown in Figure 12, where component C_7 requests two services g, h concurrently. Both are handled asynchronously and can return in either order, with the last reply allowing f to reply.

The inferred, stateless, model does not capture the dependency between replies rep_g, rep_h and rep_f. When both, or neither, incoming replies are followed by rep_f, the system deadlocks. This is solved by enforcing nested services to be finished before finishing f, as discussed in Section IV-C.

If we only observe replies coming in in one order, we do not infer how to respond when the replies commute, which is allowed by the generalization made. This requires us to compose the union of the model and the missing task.

The second issue is shown in Figure 13, 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}, with S^*_2 not observed at all. This is because the lower logical layers are not observed. A possible perspective of the actual system is shown in Figure 13b.

As we do not observe S^*_1 and S^*_2, we miss the dependency between functions g and e_2, in S^*_1. Now, e_2 can start 'spontaneously', which leads to deadlocks as well. From domain knowledge, we know the dependency between g and e_2. The dependency is added with a one-place buffer similar to Figure 6a, extended to multiple places as needed.

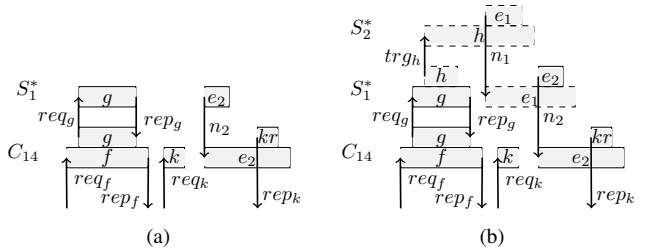

Fig. 13. Component C_{14} communicates with components outside the observation, which leads to a lack of dependencies.

D. Model Improvements: Analyzing Choice

The model inferred for Figure 12 contains tasks which have a common prefix. Namely, we have $hr^{\uparrow}hr^{\downarrow}$ as well as $hr^{\uparrow}fr^{\uparrow}fr^{\downarrow}hr^{\downarrow}$. One could say that a choice arises after the first symbol, $hr^{\uparrow}$. The inferred systems still contains some of these choices, as for instance g is optional in Figure 12. Regardless of the absence of deadlocks, this 'choice' still enlarges the state space and might not represent reality.

To gain insight in these choices, we rerun the inference approach. During construction of the PTA, a word can either move from $h^{\downarrow}$ to $g^{\uparrow}$, or skip g and go to $f^{\downarrow}$. Then, we assert whether the choice depends on another piece of state in the observation. This could be a state that is observable in the model, as for example the state from which f on C_7 is called might vary. This can propagate to the caller of the caller, etc.

The choice might also depend on a piece of state that is not inferred, such as the duration of the previous function. Domain experts can easily pose an hypothesis, which can be verified against the observation.

For the example, g was only skipped on a single occurrence. This happens to be related to the repetitions we observed for C_1, with g skipped on one particular iteration. Domain experts confirm and understand this conclusion. The model is corrected by renaming the prefix of f, both in C_7 as well as in the right iteration of C_1.

E. Asynchronous Communication

We extend the model using asynchronous communication as described in Section V. Due to the issues described there, we modeled the communication buffers as R-FD-FIFOs: we should allow for travel time variations due to hick ups in the communication network, and allow the task to define the order of messages that are not handled at a root execution.

The buffers in the model require a bound of either one- or two-, with the exception of component C_2 requiring 24. The low number of buffer places required is due to the extensive synchronization on replies, with C_2, the log component, only receiving logging triggers.

We verified that the resulting, improved, asynchronous model is trace-conform with the input TMSC.

VII. Discussion

In this work, we introduced an approach to infer a model of a component-based system, by considering the system to be a composition of components and buffers. The components have been further decomposed in a collection of tasks.

By giving a trace-theoretical foundation to our method, we have shown the benefits when inferring a system matching the assumptions of the method. Decomposition in components allows us to generalize to trace equivalence, while avoiding the need to observe or capture concurrency explicitly. Conventional methods cannot do so, let alone prove the generalization to be included in language of the system, again withstanding assumptions.

Further decomposition in tasks is also important, given the length of the observation as seen in Section VI. Decomposing a component in tasks allow us to generalize to executions of longer or shorter length, while our inference result is based on a single observation.

More importantly, we have discussed in Section VI that tasks can be interchanged depending on the order of incoming messages. Such generalizations are not made by conventional methods, and require us to use characteristics of the system. Conversely, if a task f on some client requests tasks, say g followed by h, on the same server, conventional methods would infer a dependency between g and h on this server. After all, these tasks always follow each other. Only by relying on system characteristics, we recognize that this is due the structure of f on the client, and says nothing about the server.

To decompose a component in a collection of tasks required us to assume that these tasks are mutually independent. Section VI has shown this assumption to be false for our target system. As it holds for a large portion of the tasks, and allows for the generalizations discussed, we mended it by analyzing the state space for missing dependencies rather than dropping the assumption altogether.

We conclude that the method can benefit from a better approach to relate tasks and services. Hence, the system should be considered as three-layers instead of two, where task sequences make up services, which in turn make up components. The results in this work show that two features can aid in inferring services from tasks on a component: (1) tasks with common symbols, and especially common prefixes, are often related. (2) tasks which are connected through message dependencies *running transitively through servers* are often related.

In Section IV, we made limited assumptions on the system, to aid the generality of the results. To distinguish services and tasks, future work might be aided by additional assumptions. Examples are explicitly distinguishing a call alphabet, in addition to function and return alphabets (see Section V), and relating call symbols to requests and triggers as introduced in Section VI.

Given that our target system is unknown, it is difficult to assess whether the inferred models are 'correct'. For the 33 components, only a single model was available for inspection (which was a superset of the inferred model due to lack of observations). Hence, the only notion of correctness available in model inference is obtained by posing assumptions on the target system, and showing generalizations to be 'correct', i.e. valid for such a system, under these assumptions. This is

the approach we took in Sections IV and V. Still, this is a theoretical notion of correctness.

In practice, we do not know whether the system is correct, as we do not know whether the services of the improved model are all mutually independent. Given the absence of deadlocks, and tasks sharing a prefix which is not directly constrained by communication, we do have strong confidence in the results.

It is our strong believe that a good inference approach is proven to be correct under assumptions, where the assumptions 1) capture the essential characteristics of the system closely and 2) are easily verifiable. The first requirement allows for a rich generalization, while the second ensure the proofs to hold on the system in practice. Decomposition in components succeeded well on the latter, where decomposition in tasks succeeded well on the former.

Above correctness, it is most important that the models serve their need, as inference is only a means to an end. In this work, we did not discuss a specific use case. However, future work is already exploring whether the inferred models can be used to assess the impact of changes made to the software, before deploying the software at a customer. In addition, the inference of interface contracts (see Section III-B) from the models is explored.

In addition to the opportunities mentioned, future work can extend the models with timing. Given that each symbol represents multiple dependencies in the TMSC, these dependencies can be analyzed for typical timings, e.g. average or minimal timings, for use in the timed model. Together with a client-server hierarchy, this might allow analyzing the system for minimal timespan schedules. While there is work considering the makespan of a TMSC [23], this work does not allow for task commutations. Our work does allow tasks to be commutated based on the order of incoming messages. As can be seen in Figure 10, this can influence the makespan. Similarly, an assumption on a client-server hierarchy might allow more specific results on the asynchronous composition, e.g. to analyze the inferred model for boundedness.

Finally, we revisit our research question: *How can we utilize knowledge of the architecture, deployment and characteristics of the target system to formulate a justified model inference approach for component based cyber physical systems?*

We conclude that the knowledge can be used to decompose the system in components, which are decomposed in services, and allows us to capture essential characteristics of such system, such as commutations of function executions and concurrency between components.

More generally, we note that the approach allowed us to iteratively refine the models using analysis, as we proved it to be robust under additional observations. For such complex systems and without the availability of counter examples, this iterative approach with a human in the loop is essential, as we do not expect to formulate a method that can capture all intricacies of such systems.

We argue that our method is

Fast and scalable: Given that our method has linear complexity, this is guaranteed.

Justified: We have clear proofs demonstrating which properties are ensured by the inferred models, and under which assumptions.

Appropriate: As a component model captures the Kleene-star closure of the set of observed tasks, the relation between observation and model is very clear. Essential characteristics such as commutations of tasks or independent symbols are captured.

Robust: Adding observations will preserve the tasks that have already been inferred, while it adds new tasks to the component model.

Extensible: The domain injection approach, combined with substitution, can add new knowledge at various places in the model.

These characteristics are essential to iteratively improve models, and obtain a clear and insightful description of the system.

VIII. Conclusions

This work introduces a new approach for inferring models from observations, by relying on the architecture of the observed system. This limits the method to system with a component-based architecture, and thereby ensures that the inferred models account for features specific to this architecture.

The approach infers concurrency among components, without needing to observe commutations, or capture the concurrency explicitly in the model. Rather, the inference approach resembles the original composition to capture concurrency.

The method also accounts for the non-preemptive nature of components, and allows services executed on these components to be commutated. Here, the approach accounts for influence of communication on the behavior of the system.

By capturing the characteristics in assumptions, we have proven that the generalizations belong to a system matching these assumptions. Thereby, we gave a trace theoretical foundation to our work.

The method has been demonstrated by applying it to a case study at ASML. We showed how to assess the validity of the assumptions, though a definite method to verify the assumption is still required. Still, we have shown that the issues are easily mended, and that we arrive at a model where all generalizations are explained.

Contrary to many traditional approaches, the inference approach does not rely on parameters. Instead, a single model is inferred, even from a single observation. There is a clear link between the model and the observation, as we capture all observed services in a component model. This makes the approach robust under new observations.

Combined with the demonstrated approach to refine the model, the robustness and transparency allow for iterative refinements of the model based on new insights. This is essential to create models of complex, cyber-physical systems.

ASML stakeholder feedback:

- *"The ability to analyze the behavior of our software at a very high level of abstraction, as well as zooming in on the details is extremely cool"*
- *"With many possible applications, ranging from architectural reasoning to patch qualification, this technology has the potential to contribute significantly to improving the quality of our software"*
- *"Very useful method, can be applied to several use cases"*

APPENDIX A
ALGORITHMS

Algorithm 1 Constructs a PTA from a set of samples: $PTA\,(W)$.

Require: A set W of positive samples belonging to target language $W \subseteq \mathcal{L}_* \subseteq \Sigma^*$.
Ensure: DFA $A = (Q, \Sigma, \delta, q_0, F)$ for which $\mathcal{L}(A) = W$.

$A \leftarrow (Q = \{q_0\}, \Sigma, \delta = \emptyset, q_0, F = \emptyset)$ ▷ Create DFA.
for all $w \in W$ **do** ▷ Iterate over all samples.
  $q \leftarrow q_0$ ▷ Track the current state of the automaton.
  for all $w_j \in \Sigma : w_1 \dots w_{|w|} = w$ **do**
    if $(q, w_j, q') \notin \delta$ for some $q' \in Q$ **then**
      create q' ▷ Ensure transition with w_j exists.
      $Q \leftarrow Q \cup \{q'\}$
      $\delta \leftarrow \delta \cup (q, w_j, q')$
    end if
    $q \leftarrow \delta(q, w_j)$ ▷ Take the transition with w_j.
  end for ▷ After processing all of w
  $F \leftarrow F \cup \{q\}$ ▷ make the tracked state accepting.
end for
return $\{A_i\}$

Algorithm 2 Constructs a task DFA from executions of a system composed of task automata: $TDFAs\,(W, \Sigma_i^{f,r})$.

Require: A set W of words from a composition A of n Task DFAs, $A = A_1 \| \dots \| A_n, W \subseteq \mathcal{L}(A)$, and the (function,return) alphabet of a component A_i in the composition of $A, 1 \leq i \leq n$.
Ensure: A Task DFAs A_i', for which $\mathcal{L}(A_i') = T(\pi_{\Sigma_i}(W))^*$.

$A_i' = (Q_i', \Sigma_i, \delta_i', q_{0,i}', F_i') \leftarrow PTA(T(\pi_{\Sigma_i}(W)))$
for all $(q, a, q') \in \delta_i', q \in Q, a \in \Sigma_i^r, q' \in F$ **do**
  $\delta_i' \leftarrow (\delta_i' \setminus (q, a, q')) \cup (q, a, q_{0,i}')$
end for
$Q, F \leftarrow Q \setminus (F \setminus \{q_{0,i}'\}), \{q_{0,i}'\}$
return A_i'

APPENDIX B
PROOFS

Proposition B.1 *Given two DFAs A_1, A_2, the synchronous composition of these DFAs is homomorphic with the synchronization of their languages.*

$$\mathcal{L}(A_1 \| A_2) = \mathcal{L}(A_1) \| \mathcal{L}(A_2)$$

Algorithm 3 Determine buffer bound in asynchronous composition: $B(w, \Sigma, \Sigma_i^{!,?})$

Require: A sample $w \in \mathcal{L}(A)$, alphabets $\Sigma_i^{!,?}$ of a buffer M_i of synchronous composition A.
Ensure: For a buffer M_i the maximal occupancy b_i during execution of w.

$b_i \leftarrow 0$
$b_{i,max} \leftarrow 0$
for all $1 \leq j \leq |w|$ **do** ▷ For each letter in w
  if $w_j \in \Sigma_i^!$ **then** ▷ If message is sent
    $b_i \leftarrow b_i + 1$ ▷ Increase occupancy
  else if $w_j \in \Sigma_i^?$ **then** ▷ If message is received
    $b_i \leftarrow b_i - 1$ ▷ Decrease occupancy
  end if
  if $b_i > b_{i,max}$ **then** ▷ Track the maximum occupancy
    $b_{i,max} \leftarrow b_i$
  end if
end for
return $b_{i,max}$

Proof. From Definitions II.2 and II.3 it is clear that

$$\delta_{A_1\|A_2}(q_0, w) \in F_{A_1\|A_2} \Leftrightarrow \delta_{A_1}(q_{0,A_1}, \pi_{\Sigma_{A1}}(w)) \in F_{A_1} \wedge \delta_{A_2}(q_{0,A_2}, \pi_{\Sigma_{A2}}(w)) \in F_{A_2}$$

as transitions are made for a state in the state tuple of $A_1 \| A_2$ iff the symbol is in the alphabet of the corresponding component. The rest follows from Definition II.4. □

Corollary B.2 *Given a DFA A composed of two automata A_1, A_2, i.e. $A = A_1 \| A_2$, over alphabets Σ_1, Σ_2, respectively, then we have*

$$w \in \mathcal{L}(A) \Leftrightarrow \pi_{\Sigma_1}(w) \in \mathcal{L}(A_1) \wedge \pi_{\Sigma_2}(w) \in \mathcal{L}(A_2)$$

Proof. Follows directly from Proposition II.5 and Definition II.4. □

Proposition B.3 *Given a dependency $D = \bigcup_{i=1}^n (\Sigma_i^2)$ and two words $u, v \in \Sigma^*$, $\Sigma = \bigcup_{i=1}^n \Sigma_i$, we have:*

$$u \equiv_D v \Leftrightarrow \forall_i : \pi_{\Sigma_i}(u) = \pi_{\Sigma_i}(v)$$

Proof. For $\Rightarrow$: As $u \equiv_D v$, we have $\pi_{\Sigma_i}(u) \equiv_D \pi_{\Sigma_i}(v)$ by Proposition III.1. Then there exists a sequence $\pi_{\Sigma_i}(u) \sim_D \dots \sim_D \pi_{\Sigma_i}(v)$. We assume, towards a contraction, that $\pi_{\Sigma_i}(u) \neq \pi_{\Sigma_i}(v)$. However, as any symbols a, b in $\pi_{\Sigma_i}(u)$ are constrained in D, any commutation does not affect the projection, and hence we arrive at a contradiction.

For $\Leftarrow$: For $u = v$ the proposition is trivially true, so we consider $u \neq v$. As $u, v \in (\bigcup_i \Sigma_i)^*$, and $\bigcup_i \Sigma_i$ spans the whole domain of D, $\pi_{\Sigma_i}(u)$ and $\pi_{\Sigma_i}(v)$ can be equal for all i iff $|u| = |v|$, and $\#a(u) = \#a(v)$ for all $a \in \Sigma$. Hence if $u \neq v$, u must be a permutation of v, i.e. there exists a sequence $u \sim_{D'} \dots \sim_{D'} v$, with D' the identity relation on Σ. Assume, towards a contradiction, that $u \not\equiv_D v$. Then at least one commutation along this sequence has $(a, b) \in I_{D'}$ and $(a, b) \notin I_D$. If $(a, b) \notin I_D$, then $a, b \in \Sigma_i$ for some i, by

definition of D. This implies $\pi_{\Sigma_i}(u) \neq \pi_{\Sigma_i}(v)$, a contradiction. □

Corollary B.4 *Consider a synchronous composition A of n DFAs, $A = A_1 \| \dots \| A_n$, with each component a DFA A_i over alphabet Σ_i. Language $\mathcal{L}(A)$ is consistent with $D = \bigcup_{i=1}^{n}(\Sigma_i^2)$, i.e. $\mathcal{L}(A) = \text{lin}[\mathcal{L}(A)]_D$, and therefore A has trace language $\mathcal{T}(A) = [\mathcal{L}(A)]_D$.*

Proof. This follows from Proposition IV.1 and Corollary II.6. □

Theorem B.5 *Consider a synchronous composition A of n DFAs A_i, $A = A_1 \| \dots \| A_n$, as well as a set W of words, $W \subseteq \mathcal{L}(A)$. The approximation A' of A, composed of components A_i', $A' = A_1' \| \dots \| A_n'$, obtained through $PTAs(W, \Sigma_0, \dots, \Sigma_n)$, has the following language:*

$$\mathcal{L}(A') = \text{lin}[W]_D, \text{ with } D = \bigcup_{i=0}^{n}(\Sigma_i^2).$$

Proof. By correctness of Algorithm 1 and Proposition II.5 we know $W \subseteq \mathcal{L}(A')$ and hence $\text{lin}[W]_D \subseteq \mathcal{L}(A')$ by Corollary IV.2. To prove $\mathcal{L}(A') \subseteq [W]_D$, take any $w \in \mathcal{L}(A')$ and any $1 \leq i \leq n$. Then we have $\pi_{\Sigma_i}(w)$ in $\mathcal{L}(A_i')$, and thus in $\pi_{\Sigma_i}(W)$ by correctness of Algorithm 1. Then we know $w \in [W]_D$ by Proposition IV.1. □

Theorem B.6 *Consider a synchronous composition A of n DFAs A_i, $A = A_1 \| \dots \| A_n$, as well as a set W of words, $W \subseteq \mathcal{L}(A)$. The approximation A' of A, composed of components A_i', $A' = A_1' \| \dots \| A_n'$, each obtained through $PTA(W, \Sigma_i)$, is an under-generalization, i.e. $\mathcal{L}(A') \subseteq \mathcal{L}(A)$.*

Proof. We have: $W \subseteq \mathcal{L}(A) \Rightarrow \text{lin}[W] \subseteq \mathcal{L}(A) \Rightarrow \mathcal{L}(A') \subseteq \mathcal{L}(A)$ by Corollary IV.2 and Theorem IV.4 respectively. □

Theorem B.7 *Consider DFA A' composed of DFAs A_i', $A = A_1' \| \dots \| A_n'$, obtained from observation U through $PTAs(U, \Sigma_0, \dots, \Sigma_n)$. DFA A'', composed and obtained similarly from observation V, $U \subseteq V$, has $\mathcal{L}(A') \subseteq \mathcal{L}(A'')$.*

Proof. From Definition II.3 and correctness of Algorithm 1, we know $U \subseteq V \Rightarrow \pi_{\Sigma_i}(U) \subseteq \pi_{\Sigma_i}(V) \Rightarrow \mathcal{L}(A_i') \subseteq \mathcal{L}(A_i'')$ for any $1 \leq i \leq n$. Apply Proposition IV.3 to get $\mathcal{L}(A') \subseteq \mathcal{L}(A'')$. □

Proposition B.8 *Given a composition A of n task DFAs, $A = \|_{i=0}^{n} A_i$ and $D = \bigcup \Sigma_i^2$, then $[t] \in \mathcal{T}(A)$ is a task sequence, $[t] \in \mathcal{T}(A) \Leftrightarrow T([t]) \subseteq \mathcal{T}(A)$ and $\mathcal{T}(A) = T(\mathcal{T}(A))^*$.*

Proof. For $[t] \in \mathcal{T}(A)$, we have $\forall_i : \pi_{\Sigma_i}(t) \in \mathcal{L}(A_i)$ by Corollaries II.6 and IV.2. As $\mathcal{L}(A_i)$ is a task DFA, $\pi_{\Sigma_i}(t)$ is a task sequence and hence $[t]$ is a task sequence trace.

($\Rightarrow$) As $[t] \in \mathcal{T}(A)$ is a task sequence, $[t] = [t_1..t_m]$, with $[t_j]$ a task word for $1 \leq j \leq m$. By Corollaries II.6 and IV.2, we know $\forall_i : \pi_{\Sigma_i}(t) \in \mathcal{L}(A_i)$, and hence $\forall_i : \pi_{\Sigma_i}(t_1..t_m) \in \mathcal{L}(A_i)$. From Definition IV.9 we know $\pi_{\Sigma_i}(t_j)$ is a task sequence w.r.t $\Sigma_i^{f,r}$ for any j, hence $\forall_i : \pi_{\Sigma_i}(t_j) \in \mathcal{L}(A_i)$ by definition of a Task DFA. Finally, this implies $[t_j] \in \mathcal{T}(A)$ for all j, and therefore $T([t]) \subseteq \mathcal{T}(A)$.

($\Leftarrow$) If $T([t]) \subseteq \mathcal{T}(A)$, then $[t]$ is a task sequence trace with $[t] = [t_1..t_m]$, and $\forall_j : [t_j] \in \mathcal{T}(A)$. As A is a composition of Task DFAs, it has a single accepting state, which is also the initial state, by Definition II.2 and IV.7. Hence $\forall_j : [t_j] \in \mathcal{T}(A)$ implies $[t] \in \mathcal{T}(A)$.

The equation $\mathcal{T}(A) = T(\mathcal{T}(A))^*$ follows directly from this. □

Proposition B.9 *Consider a DFA A composed of n Task DFAs A_i such that $A = A_1 \| \dots \| A_n$. Given a set W of words $W \subseteq \mathcal{L}(A)$, algorithm $TDFAs(W, \Sigma_{1..n}^{f,r})$ returns Task DFAs A_i' such that:*

$$\mathcal{L}(A_i') = T(\pi_{\Sigma_i}(W))^*, \text{ and } \mathcal{L}(A_i') \subseteq \mathcal{L}(A_i)$$

Proof. By correctness of Algorithm 1, we know A_i' in line 2 has $\mathcal{L}(A_i') = T(\pi_{\Sigma_i}(W))$. As a task cannot have a prefix which is also a task by Definition IV.6, every final state has no outgoing transitions. Therefore, the Kleene star closure is applied by making the initial state accepting (line 6) and routing all transitions with symbols $a \in \Sigma_i^r$ to this initial state (line 4). By construction, the resulting DFA is a Task DFA, adhering to Definition IV.7. As $W \subseteq \mathcal{L}(A)$ and A_i is a Task DFA, we have $T(\pi_{\Sigma_i}(W))^* \subseteq \mathcal{L}(A_i)$. □

Proposition B.10 *If $\mathcal{L}(A_1') \subseteq \mathcal{L}(A_1)$ and $\mathcal{L}(A_2') \subseteq \mathcal{L}(A_2)$, then $\mathcal{L}(A_1' \| A_2') \subseteq \mathcal{L}(A_1 \| A_2)$.*

Proof. We use Definition II.4 and the premise, then we have:

$$\begin{aligned} w \in \mathcal{L}(A_1' \| A_2') &\Leftrightarrow \pi_{\Sigma_1}(w) \in \mathcal{L}(A_1') \wedge \pi_{\Sigma_2}(w) \in \mathcal{L}(A_2') \\ &\Rightarrow \pi_{\Sigma_1}(w) \in \mathcal{L}(A_1) \\ &\wedge \pi_{\Sigma_2}(w) \in \mathcal{L}(A_2) \\ &\Leftrightarrow w \in \mathcal{L}(A_1 \| A_2) \end{aligned}$$

□

Theorem B.11 *Consider a DFA A composed of n Task DFAs A_i such that $A = A_1 \| \dots \| A_n$. Given a set W of words $W \subseteq \mathcal{L}(A)$, algorithm $TDFA(W, \Sigma_i^{f,r})$ returns Task DFAs A_i' such that DFA $A' = A_1' \| \dots \| A_n'$ has $\mathcal{L}(A') \subseteq \mathcal{L}(A)$*

Proof. From Propositions IV.11 and IV.3 we know $\mathcal{L}(A_i') \subseteq \mathcal{L}(A_i)$, and $\mathcal{L}(A') \subseteq \mathcal{L}(A)$. □

Theorem B.12 *Consider DFA A' composed of DFAs A_i', $A = A_1' \| \dots \| A_n'$, obtained from observation U through $TDFAs(U, \Sigma_i^{f,r})$. DFA A'', composed and obtained similarly from observation V, $U \subseteq V$, has $\mathcal{L}(A') \subseteq \mathcal{L}(A'')$.*

Proof. By Definition IV.8, we see for any $1 \leq i \leq n$, $U \subseteq V \Rightarrow T(\pi_{\Sigma_i}(U)) \subseteq T(\pi_{\Sigma_i}(V))$. Hence $\mathcal{L}(A_i') \subseteq \mathcal{L}(A_i'')$ by Proposition IV.11, and $\mathcal{L}(A') \subseteq \mathcal{L}(A'')$ by Proposition IV.3. □

Proposition B.13 *Given a composition A of n Task DFAs $A_1, \dots, A_n$ communicating through bounded FIFO (bag) buffers M_i. Given a set $W \subseteq \mathcal{L}(A)$, synchronous composition $A' = \|_i(A_i' \| B_i'^{b_i})$, with A_i' obtained through $TDFAs(W, \Sigma_i^{r,f})$ and $B_i'^{b_i}$ a FIFO (bag) buffer approximating M_i with bound b_i given by $b_i = \max_{w \in W} B(w, \Sigma_i^{!,?,\tau})$, has $W \subseteq \mathcal{L}(A')$.*

Proof. From Proposition IV.11, we know $\pi_{\Sigma_i}(W) \subseteq \mathcal{L}(A_i')$. By Algorithm 3, we know computing any $w \in W$ over $B_i'^{b_i}$

requires $B(w, \Sigma_i^{!,?,\tau})$ spaces in the buffer, which are available by construction. Hence we know $\pi_{\Sigma_{M_i}}(W) \subseteq \mathcal{L}(B_i'^{b_i})$, and by Proposition II.5, $W \subseteq \mathcal{L}(A')$. □

Theorem B.14 *Given a composition A of n Task DFAs $A_1, \ldots, A_n$ communicating through bounded FIFO (bag) buffers M_i. Given a set $W \subseteq \mathcal{L}(A)$, synchronous composition $A' = \|_i(A_i' \| B_i'^{b_i})$, with A_i' obtained through $TDFAs(W, \Sigma_i^{r,f})$ and $B_i'^{b_i}$ a FIFO (bag) buffer approximating M_i with bound b_i given by $b_i = \max_{w \in W} B(w, \Sigma_i^{!,?,\tau})$, has $W \subseteq \mathcal{L}(A') \subseteq \mathcal{L}(A)$.*

Proof. From Theorem B.11 we know $\mathcal{L}(\|_i A_i') \subseteq \mathcal{L}(\|_i A_i)$. Given that observation W requires M_i to have at least bound b_i, and we know the type of buffer a-priori, we have $\mathcal{L}(B_i'^{b_i}) \subseteq \mathcal{L}(M_i)$. Applying Proposition IV.3 results in $\mathcal{L}(A') \subseteq \mathcal{L}(A)$. □

Theorem B.15 *Consider DFA A' synchronously composed of n Task DFAs, $A_i', 1 \leq i \leq n$, and buffers $B_i'^{b_i'}$, $A' = \|_i(A_i' \| B_i'^{b_i'})$, with $A_i', B_i'^{b_i'}$ reconstructed using $TDFAs(U, \Sigma_i^{f,\tau})$ and $b_i' = \max_{w \in U} B(w, \Sigma_i^{!,?,\tau})$ resp. from observation U. Given A'' composed and obtained similarly from observation V, $U \subseteq V$, we have $\mathcal{L}(A') \subseteq \mathcal{L}(A'')$.*

Proof. The proof of Theorem IV.12 is extended to the asynchronous case by considering $U \subseteq V \Rightarrow \max_{w \in U} B(\ldots) \leq \max_{w \in V} B(\ldots)$, hence $\mathcal{L}(B_i'^{b_i'}) \subseteq \mathcal{L}(B_i''^{b_i''})$, together with Proposition IV.3. □

REFERENCES

[1] D. Akdur, V. Garousi, and O. Demirörs, "A survey on modeling and model-driven engineering practices in the embedded software industry," *Journal of Systems Architecture*, vol. 91, no. October, pp. 62–82, 2018.

[2] L. Akroun and G. Salaün, "Automated verification of automata communicating via FIFO and bag buffers," *Formal Methods in System Design*, vol. 52, no. 3, pp. 260–276, 2018.

[3] O. al Duhaiby, A. Mooij, H. van Wezep, and J. F. Groote, "Pitfalls in Applying Model Learning to Industrial Legacy Software," in *Leveraging Applications of Formal Methods, Verification and Validation. 8th International Symposium, ISoLA*, T. Margaria and B. Steffen, Eds., vol. 11247 LNCS. Springer International Publishing, 2018, pp. 121–138.

[4] R. Alur and D. L. Dill, "A theory of timed automata," *Theoretical Computer Science*, vol. 126, no. 2, pp. 183–235, 1994.

[5] D. Angluin, "Inductive inference of formal languages from positive data," *Information and Control*, vol. 45, no. 2, pp. 117–135, 1980.

[6] ——, "Queries and Concept Learning," *Machine Learning*, vol. 2, pp. 319–342, 1988.

[7] K. Aslam, L. Cleophas, R. Schiffelers, and M. van den Brand, "Interface protocol inference to aid understanding legacy software components," *Software and Systems Modeling*, vol. 19, no. 6, pp. 1519–1540, 2020. [Online]. Available: https://doi.org/10.1007/s10270-020-00809-2

[8] F. Avellaneda and A. Petrenko, "Inferring DFA without Negative Examples," in *14th International Conference on Grammatical Inference (ICGI)*, 2019, pp. 17–29.

[9] M.-P. Béal and M. Crochemore, "Minimizing incomplete automata," *Finite-State Methods and Natural Language Processing (FSMNLP'08)*, no. April 2008, pp. 9–16, 2008. [Online]. Available: http://igm.univ-mlv.fr/{~}beal/Recherche/Publications/minimizingIncomplete.pdf

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

[11] I. Beschastnikh, Y. Brun, M. D. Ernst, and A. Krishnamurthy, "Inferring models of concurrent systems from logs of their behavior with CSight," *Proceedings - International Conference on Software Engineering*, pp. 468–478, 2014.

[12] D. Brand and P. Zafiropulo, "On Communicating Finite-State Machines," *Journal of the ACM (JACM)*, vol. 30, no. 2, pp. 323–342, 1983.

[13] O. Cicchello and S. C. Kremer, "Beyond EDSM," in *The 6th International Colloquium on Grammatical Inference (ICGI 2002)*, vol. LNAI 2484, 2002, pp. 37–48.

[14] E. Clarke, "Model Checking," in *International Conference on Foundations of Software Technology and Theoretical Computer Science*, S. Ramesh and G. Sivakumar, Eds. Berlin Heidelberg: Springer Berlin Heidelberg, 1997, pp. 54–56.

[15] E. Clarke, O. Grumberg, D. Kroening, and D. A. Peled, *Model Checking*. Cambridge, Massachusetts: MIT Press, 2000.

[16] C. de la Higuera, *Grammatical Inference*, 1st ed. New York: Cambridge University Press, 2010.

[17] B. Genest, D. Kuske, and A. Muscholl, "On communicating automata with bounded channels," *Fundamenta Informaticae*, vol. 80, no. 1-3, pp. 147–167, 2007.

[18] B. Gröne, A. Knöpfel, and P. Tabeling, "Component vs. component: why we need more than one definition [system component and software component]," *Engineering of Computer-Based Systems, 2005. ECBS'05. 12th IEEE International Conference and Workshops on the*, pp. 550–552, 2005.

[19] J. F. Groote and M. R. Mousavi, *Modeling and Analysis of Communicating Systems*. Cambridge, Massachusetts: The MIT Press, 2014.

[20] M. J. Heule and S. Verwer, "Exact DFA Identification Using SAT Solvers," in *Grammatical Inference: Theoretical Results and Applications, 10th International Colloquium, ICGI 2010, Valencia, Spain, Proceedings*, 2010, pp. 66–79.

[21] ——, "Software model synthesis using satisfiability solvers," *Empirical Software Engineering*, vol. 18, no. 4, pp. 825–856, 2013.

[22] ITU-TS, "Recommendation Z.120, Message Sequence Charts," Geneva, 1999.

[23] R. Jonk, J. Voeten, M. Geilen, T. Basten, and R. Schiffelers, "SMT-based verification of temporal properties for component-based software systems," Eindhoven University of Technology, Department of Electrical Engineering, Electronic Systems Group, Eindhoven, Tech. Rep. ES Reports, 2020. [Online]. Available: http://www.es.ele.tue.nl/esreports/esr-2020-01.pdf

[24] R. Jonk, J. Voeten, M. Geilen, and R. Theunissen, "Inferring Timed Message Sequence Charts from Execution Traces of Large-scale Component- based Software Systems," Eindhoven University of Technology, Department of Electrical Engineering, Electronic Systems Group, Eindhoven, Tech. Rep. ES Reports, 2019.

[25] I. Kurtev, M. Schuts, J. Hooman, and D. J. Swagerman, "Integrating interface modeling and analysis in an industrial setting," *Proceedings of the 5th International Conference on Model-Driven Engineering and Software Development*, no. Modelsward, pp. 345–352, 2017.

[26] D. Kuske and A. Muscholl, "Communicating Automata," *Submitted for peer review (Journal unknown)*, 2019. [Online]. Available: http://eiche.theoinf.tu-ilmenau.de/kuske/Submitted

[27] K. J. Lang, B. A. Pearlmutter, and R. A. Price, "Results of the abbadingo one DFA learning competition and a new evidence-driven state merging algorithm," in *ICGI 1998: Grammatical Inference*, vol. LCNS 1433, 1998, pp. 1–12.

[28] M. Leemans, "Hierarchical Process Mining for Scalable Software Analysis," Ph.D. dissertation, Eindhoven University of Technology, 2018.

[29] R. Loose, B. van der Sanden, M. Reniers, and R. Schiffelers, "Component-wise Supervisory Controller Synthesis in a Client/Server Architecture," *IFAC-PapersOnLine*, vol. 51, no. 7, pp. 381–387, 2018. [Online]. Available: https://doi.org/10.1016/j.ifacol.2018.06.329

[30] E. Mark Gold, "Language identification in the limit," *Information and Control*, vol. 10, no. 5, pp. 447–474, 1967.

[31] V. Matena, B. Stearns, and L. Demichiel, *Applying Enterprise JavaBeans: Component-Based Development for the J2EE Platform*, 2nd ed. Pearson Education, 2003.

[32] A. Mazurkiewicz, "Introduction to Trace Theory," in *The Book of Traces*, V. Diekert and G. Rozenberg, Eds. Singapore: World Scientific, 1995, ch. 1, pp. 3–41.

[33] M. McIlroy, "Mass Produced Software Components," in *Software Engineering, Report on a conference sponsored by the NATO Science Committee 7th to 11th October 1968*, P. Naur and B. Randell, Eds. Garmisch, Germany: Scientific Affairs Division, NATO, 1968, pp. 138–155.

[34] R. Milner, *Communication and Concurrency*. River, NJ, United States: Prentice-Hall, Inc., 1989.

[35] R. Morin, "On regular message sequence chart languages and relationships to Mazurkiewicz trace theory," in *Foundations of Software Science and Computation Structures, 4th International Conference*, vol. 2030, Genova, Italy, 2001, pp. 332–346.

[36] A. Muscholl, "Analysis of communicating automata," in *LATA 2010: Language and Automata Theory and Applications*, 2010, pp. 50–57.

[37] J. Muskens, M. R. Chaudron, and J. J. Lukkien, "A component framework for consumer electronics middleware," *Lecture Notes in Computer Science*, vol. 3778, pp. 164–184, 2005.

[38] A. Nakamura, T. Saito, I. Takigawa, and M. Kudo, "Fast algorithms for finding a minimum repetition representation of strings and trees," *Discrete Applied Mathematics*, vol. 161, no. 10-11, pp. 1556–1575, 2013. [Online]. Available: http://dx.doi.org/10.1016/j.dam.2012.12.013

[39] P. J. Ramadge and W. M. Wonham, "Supervisory Control of a Class of Discrete Event Processes." *SIAM Journal on Control and Optimization*, vol. 25, no. 1, pp. 206–230, 1987.

[40] M. Schuts, J. Hooman, and F. Vaandrager, "Refactoring of Legacy Software using Model Learning and Equivalence Checking: an Industrial Experience Report," in *Integrated Formal Methods: 12th International Conference*, 2016, pp. 311–325.

[41] W. Smeenk, J. Moerman, F. Vaandrager, and D. N. Jansen, "Applying Automata Learning to Embedded Control Software," in *ICFEM 2015: Formal Methods and Software Engineering*, M. Butler, S. Conchon, and F. Zaïdi, Eds. Springer International Publishing, 2015, pp. 67–83.

[42] C. Szyperski, D. Gruntz, and S. Murer, *Component Software: Beyond Object-Oriented Programming*, 1st ed. Boston, MA, USA: Addison-Wesley, 1998.

[43] D. A. van Beek, W. J. Fokkink, D. Hendriks, A. Hofkamp, J. Markovski, J. M. Van De Mortel-Fronczak, and M. A. Reniers, "CIF 3: Model-based engineering of supervisory controllers," *Lecture Notes in Computer Science*, vol. 8413 LNCS, pp. 575–580, 2014.

[44] W. van der Aalst, *Process Mining*, 2nd ed. Berlin Heidelberg: Springer-Verlag Berlin Heidelberg, 2016.

[45] W. M. van der Aalst, B. F. van Dongen, J. Herbst, L. Maruster, G. Schimm, and A. J. Weijters, "Workflow mining: A survey of issues and approaches," *Data and Knowledge Engineering*, vol. 47, no. 2, pp. 237–267, 2003.

[46] P. Vitharana, "Risks and Challenges of Component-Based Software Development," *Communications of the ACM*, vol. 46, no. 8, pp. 67–72, 2003.

[47] C. Wagner, *Model-Driven Software Migration: A Methodology*. Springer Vieweg, 2014.

[48] N. Walkinshaw, K. Bogdanov, C. Damas, B. Lambeau, and P. Dupont, "A framework for the competitive evaluation of model inference techniques," in *MIIT 2010 - Proceedings of the 1st International Workshop on Model Inference In Testing, Held in Conjunction with ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2010*, 2010, pp. 1–9.

[49] I. Warren, *The Renaissance of Legacy Systems*. London: Springer-Verlag London, 1999.

[50] W. M. Wonham, K. Cai, and K. Rudie, "Supervisory control of discrete-event systems : A brief history," *Annual Reviews in Control*, vol. 45, pp. 250–256, 2018.

[51] N. Yang, K. Aslam, R. Schiffelers, L. Lensink, D. Hendriks, L. Cleophas, and A. Serebrenik, "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*, pp. 253–263, 2019.