Automated semantics-preserving parallel decomposition of finite component and connector architectures

Oliver Kautz[1] · Bernhard Rumpe[1] · Andreas Wortmann[1]

Received: 6 February 2019 / Accepted: 20 March 2020

Abstract

For the systematic development of logical, message-driven architectures, automating parallel decomposition of software components is important to achieve efficient modular and parallel system development. During development, monolithic components that realize multiple independent concerns need to be decomposed to obtain a higher quality architecture of cohesively encapsulated, better comprehensive components. Previous work did not address automated parallel decomposition of finite message-driven and logically timed components with respect to the influence of messages received via input channels on the messages sent via output channels. This, however, is a necessary prerequisite to enable the analysis of event chains across logically distributed architectures. To address this, we present a concept of influence between channels of components that supports automated semantics-preserving parallel decomposition of finite deterministic component implementations into independent, more comprehensible components that are better accessible for analysis and development. Therefore, we extend the Focus theory of time-synchronous components with the concept of influence, present a decomposition procedure leveraging this, and prove that the resulting system is semantically equivalent. This enables automatically decomposing monolithic software components (e.g., for stepwise refinement or refactoring) into smaller components of better cohesion and comprehensibility and thus facilitates automated software engineering.

Keywords Automated modeling · Architecture decomposition · Refinement · Refactoring

✉ Oliver Kautz
kautz@se-rwth.de

Bernhard Rumpe
rumpe@se-rwth.de

Andreas Wortmann
wortmann@se-rwth.de

[1] Software Engineering, RWTH Aachen University, Aachen, Germany

1 Introduction

Component-based software engineering (Naur et al. 1968) promises efficient software development through reuse of independently developed and validated components. Usually, these components are realized in general-purpose programming languages (GPLs) and are hence subject to the conceptual gap between the problem domains and the software development, which arises from addressing problem domain challenges through programming complexities (France and Rumpe 2007).

Model-driven development (MDD) (Völter et al. 2013) reduces this gap by lifting domain-specific, abstract models to primary development artifacts. These models are specified in terms of domain-specific vocabulary to be better comprehensible, more abstract, and, hence, better suited towards analysis and transformation than the programs of GPLs.

Architecture description languages (ADLs) (Medvidovic and Taylor 2000) leverage the potential of MDD for the description of software architectures. Research has produced over 120 ADLs (Malavolta et al. 2013 for different domains, such as automotive (Debruyne et al. 2005), avionics (Feiler and Gluch 2012), consumer electronics (Van Ommering et al. 2000), or robotics (Schlegel et al. 2011). In domains, where ADLs are popular, explicating the precise semantics of architecture models is crucial, e.g., due to safety concerns. Nonetheless, many ADLs provide translational semantics, i.e., ground the meaning of architectures through their transformation into better-understood formalisms (e.g., GPL code), only. And even where the ADL's semantics are explicitly available, the MDD related processes rarely exploit these to facilitate modeling.

Where the semantics of an ADL is made explicit, semantically sound system analyses and automated refactorings and refinements become possible. Focus (Broy and Stølen 2001; Broy 2010; Ringert and Rumpe 2011), for instance, is a framework and semantic foundation that captures logical component and connector software architectures as stream-processing functions. Stream processing functions describe the histories of messages communicated over communication channels established by connectors between the components' interfaces. Architecture modeling formalisms explicating component semantics, such as Focus, communicating sequential processes (CSPs) (Hoare 1978), or the π-calculus (Milner 1999) enable systematic *stepwise refinement* (Broy 2010), a software engineering methodology for continuous architecture modeling based on controlled evolution and progressive improvement of components: each subsequent version of a component model must adhere to properties already proven for its predecessors. Ideally, this process starts with several underspecified components which are iteratively refined according to their requirements. Focus is one of the rare frameworks, where refinement and decomposition are compatible, i.e., refining a single component of an architecture always refines the complete architecture. A *component refactoring* is a special refinement step where the resulting component's semantics is equal to the semantics of the original. With this, from an external observer's viewpoint, the behaviors of the original and the resulting components are indistinguishable.

Manual refinement and refactoring without tool support, however, is tedious and error-prone. To facilitate this, we present a method for *automated refactoring* via parallel component decomposition based on the notion of *influence* between channels of components. This method uses time-synchronous port automata to represent the essence (i.e., reduced abstract syntax) of common ADLs, such as AADL (Feiler and Gluch 2012), AutoFocus (Hölzl and Feilkas 2010), EAST-ADL (Debruyne et al. 2005), MontiArc (Butting et al. 2017a), SysML's blocks (Friedenthal et al. 2011), and similar languages. Given a component implementation, we propose to automatically decompose it into subcomponents according to their influence relation. To this effect, we assume the availability of a model that describes the external interface of the component (e.g., an ADL model) and a description of the implementation of the component. It is irrelevant whether the description of the component implementation is available in source code that can be transformed to a time-synchronous port automaton or whether the implementation is directly described by a time-synchronous port automaton. To this end, our contributions are:

- A notion of *influence* between channels of a logical software architecture that is grounded in the Focus theory.
- A method to *automatically refactor* components with finite state spaces via parallel decompositions according to the influence relation.

The resulting architecture can be evolved more efficiently by different stakeholders, yet is guaranteed to be semantically equivalent to the previous architecture. Hence, all original properties still hold, despite being less complicated and better to evolve and maintain.

In the following, Sect. 2 sketches the idea of automated decomposition based on the influence relations between channels. Section 3 presents the system model that underlies the approach and has been introduced in previous work. Section 4 presents the notion of *influence* formalized in the Focus terminology and the process of decomposition based on it. Section 4.3 shows that the influence relation is decidable in the context of finite-state components. Afterwards, Sect. 5 presents its application on the example of the elevator control system presented and evaluated in Butting et al. (2017b). Section 6 discusses observations. Section 7 highlights related work, before Sect. 8 concludes.

2 Example

Modern software systems comprise hundreds or thousands of components. Starting development with the correct and final software architecture structure is phantasmal. Consequently, agile methods call for methods to iteratively evolve and complete software architectures. Stepwise refinement is such a method for agile software architecture modeling. With stepwise refinement, properties proven for a specific version of a component hold for all its refined successors. Hence, even early versions of architectures can be used to prove properties relevant to the customers without the burden of proving these for each new version again as long as refinement is respected.

Consider, for example, developing the software architecture for a cyber-physical system in terms of its components through stepwise refinement, such as the elevator control system (ECS) presented in Strobl et al. (1999). At some point, the team developed an initial `ECS` architecture that consists of a single, monolithic component managing elevator requests, lights on the floors, cabin movement, as well as opening and closing the elevator cabin's door based on messages received from its environment. This component yields a single state-based behavior implementation realizing parts of the customers' requirements, i.e., is potentially shippable. Figure 1 illustrates the `ECS` component, which receives environmental messages through its input channels and outputs messages via its output channels. Engineering the (initial) software of such a system monolithically is valid with respect to stepwise refinement, but raises two challenges:

1. *Analysis challenge:* Proving architectural properties, for example, that the elevator control system eventually serves each floor for which the request button was pressed, already for initial architectures enables fixating properties relevant to customers early. However, model checking the complete `ECS` architecture might be challenging to impossible due to its complex implementation intertwining the different concerns unrelated to the property under consideration (here, e.g., management of floor lights).
2. *Implementation challenge:* Evolving functionalities implemented by such a monolith usually is overly complicated: in the worst case, parts of the implementation are scattered over different places and are hardly documented. This makes evolution error-prone and costly.

Addressing both challenges can be facilitated by properly decomposing the monolithic software architecture prior to analyzing or evolving it. For instance, decomposing the `ECS` architecture into subcomponents focusing on the *influence* between channels related to the property under consideration (such as `btn1` and `at1`) can facilitate model checking and implementation evolution. However, this

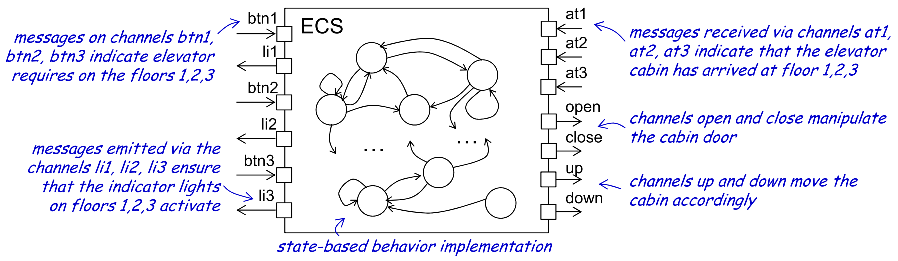

ECS
btn1
li1
btn2
li2
btn3
li3
at1
at2
at3
open
close
up
down
messages on channels btn1, btn2, btn3 indicate elevator requires on the floors 1,2,3
messages emitted via the channels li1, li2, li3 ensure that the indicator lights on floors 1,2,3 activate
messages received via channels at1, at2, at3 indicate that the elevator cabin has arrived at floor 1,2,3
channels open and close manipulate the cabin door
channels up and down move the cabin accordingly
state-based behavior implementation
...
...

Fig. 1 Initial software architecture of the elevator control system

raises challenges in properly decomposing the architecture at hand, such that the resulting decomposition into interconnected subcomponents is actually a refactoring of the original.

An automated procedure for decomposing component and connector architectures that supports both challenges must ensure that resulting subcomponent configurations are a valid refactoring of the input architecture, and support selecting input channels and output channels that should be considered as bundles to capture the developers' knowledge about channel semantics and, hence, ultimately lead to useful subcomponents. The following sections present a procedure that supports both.

3 Preliminaries

This section presents a system model for time-synchronous systems. The system model has been introduced in previous work (Butting et al. 2017b). Architectures are networks of autonomous components that interact with each other via messages on typed channels. A time-synchronous (Broy and Stølen 2001; Broy 2010; Ringert and Rumpe 2011; Butting et al. 2017b; Grosu and Rumpe 1995) architecture is interpretable as a system where execution is divided into time-units. Time units are an abstract modeling concept. In implementations, components may be unaware of time, have local times, or even synchronize mimicking a global clock. In each time unit, each component receives finitely many input messages, performs finitely many internal computations, and then eventually outputs finitely many messages to its environment.

Notation We denote by $[X \to Y]$ the set of all functions from a set X to a set Y. For a function $f \in [X \to Y]$ and a set $Z \subseteq X$, we denote by $f|_Z \in [Z \to Y]$ the function that satisfies $f|_Z(x) = f(x)$ for all $x \in Z$, called the restriction of f to Z.

3.1 Streams

The history of messages received or emitted by a component is formally described by a stream (sequence/word) of messages. Let M be an arbitrary non-empty set. Similar to (Broy and Stølen 2001; Butting et al. 2017b), M^* denotes the set of all finite streams over M. M^∞ denotes the set of all infinite streams over M and $M^\omega = M^* \cup M^\infty$ denotes the set of all finite and infinite streams over M. We denote the empty stream by $\varepsilon \in M^*$. The concatenation of two streams $s, t \in M^\omega$ is denoted by $s \cdot t$. If $s \in M^\infty$, then $s \cdot t = s$ for all $t \in M^\omega$. The prefix relation $\sqsubseteq$ over streams is defined as usual: $s \sqsubseteq t \Leftrightarrow \exists u \in M^\omega : s \cdot u = t$. For $t \in \mathbb{N}$, the $(t+1)$-th element of a stream s is denoted by $s.t$. Similarly, $s \downarrow_t$ denotes the prefix of s of length t. For example,

- $p = 3, 1, 4 \in \mathbb{N}^*$ is a finite stream over the natural numbers where $p.0 = 3$, $p.1 = 1$ and $p.2 = 4$.
- The stream $s = 7, 8, 9, \cdots \in \mathbb{N}^\infty$ where $s.0 = 7$ and $s.(t+1) = 1 + s.t$ for all $t \in \mathbb{N}$ is an example for an infinite stream of natural numbers.

- It holds that $7, 8 \sqsubseteq s$, i.e., the stream $7, 8$ is a prefix of the stream s.
- The concatenation $p \cdot s$ yields the infinite stream $p \cdot s = 3, 1, 4, 7, 8, 9, \cdots \in \mathbb{N}^\infty$.
- The prefix of length two of s is the stream $s \downarrow 2 = 7, 8$.

3.2 Messages, types

In the remainder, let M denote an arbitrary but fixed set of data elements (messages) that contains a designated element $\xi \in M$ modeling the mathematical concept of an empty pseudo-message. In a time-synchronous setting, where in each time unit at most one message is communicated via each channel, the empty message ξ can be used to model the progress of time, i.e., the message ξ is not explicitly communicated. It is important to emphasize that this communication model permits logical time while abstracting from real time. We model data types by sets of messages. Each message type contains the empty message. With this, it is possible to explicitly model the absence of a message on a communication channel in a time unit. Let *Type* denote a set of types where each type $t \in \mathit{Type}$ satisfies $t \subseteq M$ and $\xi \in t$. Types are used to restrict the set of messages that are allowed to be sent via a communication channel. As a concrete example, the type $\mathit{Nat} \in \mathit{Type}$ containing all natural numbers and the empty message ξ can be defined by $\mathit{Nat} = \{\xi\} \cup \mathbb{N}$.

3.3 Channels, histories

A channel is a communication link between components. Each channel has a unique name. In the remainder, let C denote a set of channels names. The function $\mathit{type} \in [C \rightarrow \mathit{Type}]$ maps each channel $c \in C$ to its type $\mathit{type}(c) \in \mathit{Type}$. A channel assignment is a function that maps channels to messages of the channels' types: A channel assignment for a set of channels $B \subseteq C$ is a function $a \in [B \rightarrow M]$ that satisfies $\forall b \in B : a(b) \in \mathit{type}(b)$. We denote by $B^{\rightarrow}$ the set of all channel assignments over B. A communication history for a set of channels $B \subseteq C$ is an infinite stream $h \in (B^{\rightarrow})^\infty$. The set of all communication histories for a set of channels $B \subseteq C$ is denoted by B^Ω. Thus, a communication history is a function that maps time units to channel assignments over their types. With this, each communication history models a full observation of the messages sent and received by a component. It should be noted that the set of communication histories $B^\Omega = (B^{\rightarrow})^\infty$ is isomorphic to the set $[B \rightarrow M^\infty]$ that satisfies $\forall b \in B : h(b) \in \mathit{type}(b)^\infty$, i.e., the set of all functions that map the channels in B to infinite streams of messages of their types. For a communication history $b \in B^\Omega$ and a time unit $t \in \mathbb{N}$, the prefix $b \downarrow_t$ thus models the communication history observed up to time t. We lift the $\downarrow$ operator to sets of communication histories in a point-wise manner: For $H \subseteq B^\Omega$, we define $H \downarrow_t = \bigcup_{h \in H} h \downarrow_t$. The restriction of a communication history $h \in B^\Omega$ to the channels in $R \subseteq B$ is defined as the communication history $h|_R$ that satisfies $(h|_R).t = (h.t)|_R$ for all $t \in \mathbb{N}$, i.e., each channel assignment in h is restricted to the channels in R.

As concrete examples,

- If $a, b \in C$ are channels typed with the natural numbers, then $type(a) = type(b) = Nat$.
- A channel assignment for the set of channels $\{a, b\}$ is given by $\alpha = \{a \mapsto 7, b \mapsto 8\} \in \{a, b\}^{\rightarrow}$. This assignment maps the channel a to the message 7 and the channel b to the message 8.
- The infinite stream $h = \alpha^{\infty} \in \{a, b\}^{\Omega}$ is a communication history for the set of channels $\{a, b\}$. In each time unit, this communication history maps the channel a to the message 7 and the channel b to the message 8.
- The prefix $h \downarrow 2 = \alpha, \alpha$ models the part of the communication history h observed in the first two time units.
- The restriction of the communication history h to the set of channels $\{a\}$ is given by $h|_{\{a\}} = \alpha|_{\{a\}}, \alpha|_{\{a\}}, \ldots = \{a \mapsto 7\}, \{a \mapsto 7\}, \cdots \in \{a\}^{\Omega}$.

3.4 Finite time-synchronous port automata

A finite time-synchronous port automaton (TSPA) specifies (an excerpt of) an interactive system architecture (Butting et al. 2017b; Grosu and Rumpe 1995). We assume a white-box view on components where each component implementation is represented by a finite TSPA. Complex system architectures are modeled via component composition, i.e., via the composition of the TSPAs representing the individual components' implementations.

A finite TSPA is a tuple $A = (I, O, S, \iota, \delta)$ where

- $I, O \subseteq C$ with $I \cap O = \emptyset$ are finite and disjoint sets of the TSPA's input and output channels,
- the type $type(c)$ of each channel $c \in I \cup O$ is finite,
- S is a finite set of states,
- $\iota \in S$ is the initial state, and
- $\delta \subseteq S \times (I \cup O)^{\rightarrow} \times S$ is the transition relation.

In the following, we simply refer to a finite TSPA by TSPA. We sometimes reference the syntactic elements of A as follows: $I_A = I$, $O_A = O$, $C_A = C(A) = I_A \cup O_A$, $S_A = S$, $\iota_A = \iota$, and $\delta_A = \delta$. A TSPA may fire a transition $(s, \theta, t) \in \delta$ if it receives $\theta|_I$ while residing in state s. When firing the transition, the automaton changes its internal state to t and outputs $\theta|_O$.

A TSPA A is called *reactive* iff $\forall s \in S_A : \forall i \in I_A^{\rightarrow} : \exists (u, a, v) \in \delta_A : u = s \wedge a|_I = i$. Reactive TSPAs are adequate models for interactive components as they define a possible reaction to every possible input and every possible state. If a TSPA is not reactive, then it may be in a state in which it receives an input for which no reaction in terms of a transition is defined. This behavior is erroneous as components are required to be able to react to every possible input in every time unit. A TSPA A is called *deterministic* iff $\forall s \in S_A : \forall i \in I_A^{\rightarrow} : |\{t \in S_A \mid \exists \theta \in C_A^{\rightarrow} : (s, \theta, t) \in \delta_A \wedge \theta|_{I_A} = i\}| = 1$, i.e., it defines exactly one transition for each possible input it may receive for each of its states. A nondeterministic TSPA resembles underspecification in a component that can be resolved by subsequent refinement steps and/or left open to a nondeterministic implementation. An execution σ of A is an infinite, alternating sequence of states and channel assignments starting with the initial state ι: $\sigma = s_0, \theta_0, s_1, \theta_1, \ldots$ such that $s_0 = \iota$ and $\forall i \in \mathbb{N} : (s_i, \theta_i, s_{i+1}) \in \delta$. We denote by $execs(A)$ the set of all executions of A. The behavior of an execution $\sigma = s_0, \theta_0, s_1, \theta_1, \ldots$ of A is defined as the infinite sequence $beh(\sigma) = \theta_0, \theta_1, \ldots$ containing only the channel assignments in σ. An execution comprises a TSPA's internal behavior, which is invisible to its environment, whereas a behavior represents an execution from a black-box viewpoint. We denote by $behs(A)$ the set of all behaviors of A.

As concrete examples, Fig. 2 depicts two TSPAs. As usual, circles represent states and edges between states represent transitions. Initial states are marked with an arrow that originates from a back dot. The transitions are labeled with their channel valuations. The TSPA A has a single input channel i and a single output channel o. The TSPA A is not deterministic and thus highly underspecified. In fact, it models all possible behaviors over the channels $i, o \in C$ where $type(i) = type(o) = \{\xi, 1\}$. The other TSPA *Switch* can be interpreted to model a simple light control switch.

Initially, the TSPA is in state `off`, which models that the light is turned off. In case, the switch is not pressed, the TSPA does not receive a message via its input channel i, represented by the empty message ξ. If the switch is pressed, the TSPA receives the message 1 via its input channel i. If the TSPA is in state `off` and the switch is not pressed, the TSPA outputs the empty pseudo-message ξ via its output channel o and remains in state `off`. This represents that the light remains turned off. In case the TSPA is in state `off` and the switch is pressed, the TSPA outputs the message 1 via its output channel o and switches its state to `on`. This represents that the light is turned on. Vice versa, the TSPA remains in state `on` and the light remains turned on as long as the switch is not pressed. As soon as the switch is pressed while the TSPA is in state `on`, the TSPA switches to state `off` and the light is turned off. A possible execution of the TSPA *Switch* is the infinite alternating sequence of states and transitions $e = (\mathit{off}, \{i \mapsto 1, o \mapsto 1\}, \mathit{on}, \{i \mapsto 1, o \mapsto \xi\})^{\infty}$. In the execution e, the light is frequently turned on and off. The behavior $beh(e)$ of the execution e is given

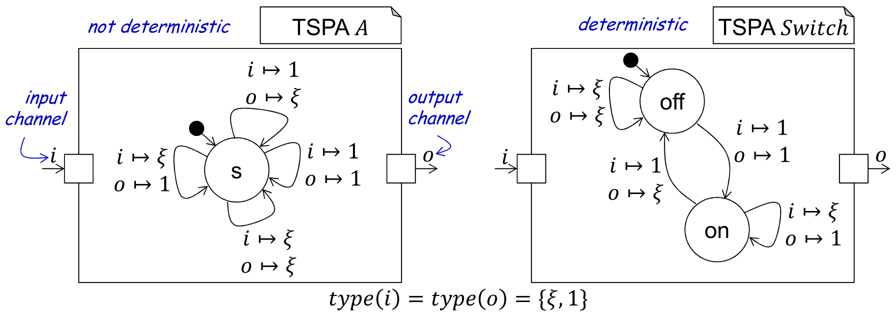

not deterministic
TSPA A
input channel
output channel
i
o
$i \mapsto 1$, $o \mapsto \xi$
$i \mapsto \xi$, $o \mapsto 1$
$i \mapsto 1$, $o \mapsto 1$
$i \mapsto \xi$, $o \mapsto \xi$
s
deterministic
TSPA *Switch*
off
on
$i \mapsto \xi$, $o \mapsto \xi$
$i \mapsto 1$, $o \mapsto 1$
$i \mapsto 1$, $o \mapsto \xi$
$i \mapsto \xi$, $o \mapsto 1$
$type(i) = type(o) = \{\xi, 1\}$

Fig. 2 An underspecified TSPA A and a deterministic TSPA *Switch*

by $beh(e) = (\{i \mapsto 1, o \mapsto 1\}, \{i \mapsto 1, o \mapsto \xi\})^{\infty}$, which is the sequence of channel assignments that represents the externally visible behavior of the execution.

3.5 TSPA composition

The composition of two TSPAs is a TSPA that captures the behavior of the architecture resulting from synchronously executing the TSPAs simultaneously where communication is carried out via the TSPAs' channels (Butting et al. 2017b; Grosu and Rumpe 1995). Multiple TSPAs may receive messages via the same channels, whereas at most one TSPA is permitted to send messages via a channel: Two TSPAs A, B are called compatible iff $O_A \cap O_B = \emptyset$.

The composition of two compatible TSPAs A and B is defined as $A \otimes B = (I, O, S_A \times S_B, (\imath_A, \imath_B), \delta)$ where

- $O = O_A \cup O_B$,
- $I = (I_A \cup I_B) \backslash O$, and
- $\delta = \{((s_1, s_2), \theta, (t_1, t_2)) \mid (s_1, \theta|_{C(A)}, t_1) \in \delta_A \wedge (s_2, \theta|_{C(B)}, t_2) \in \delta_B\}$.

Figure 3 illustrates the composition of two TSPAs. If the two TSPAs A and B represent the implementations of two components, then the composed TSPA $A \otimes B$ represents the implementation of the system obtained from running the components in parallel.

The composition of two compatible, reactive TSPAs does not always yield a reactive TSPA (Butting et al. 2017b; Grosu and Rumpe 1995). Thus, composing two components is not always meaningful as the composition of two components represented by two TSPAs may not be well-defined. This is because of causality problems (Broy and Stølen 2001; Broy 2010; Butting et al. 2017b; Grosu and Rumpe 1995) that can only exist if each of the TSPAs has an output channel that is an input channel of the respective other TSPA. The causality problem is guaranteed to be avoided if one of the TSPAs is strongly-causal (Butting et al. 2017b; Grosu and Rumpe 1995) with respect to its connected channels. However, if two reactive TSPAs are composed in parallel (without feedback), i.e., neither of

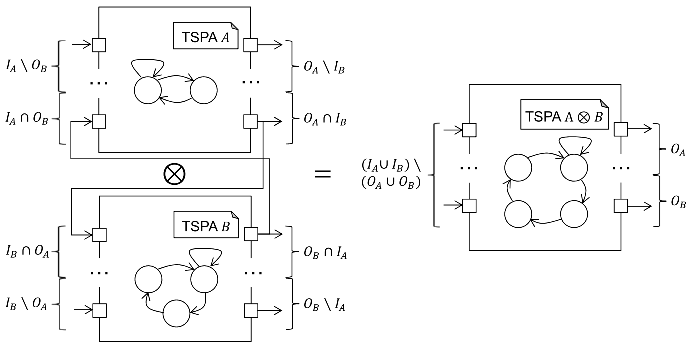


Fig. 3 General TSPA composition with feedback

the TSPAs has an output channel that is an input channel of the respective other TSPA, then the composition always yields a well-defined reactive TSPA (Butting et al. 2017b; Grosu and Rumpe 1995). As this paper is solely concerned with parallel decomposition and thus, vice versa, only with parallel composition, we refer to related work (Broy and Stølen 2001; Broy 2010; Butting et al. 2017b) for a discussion about causality complications.

3.6 TSPA restriction

Hiding is an important concept to achieve modularity (Broy and Stølen 2001; Broy 2010; Grosu and Rumpe 1995). Hiding an output channel facilitates concealing unimportant information to an environment. Similarly, it is possible to hide an input channel. Hiding an input channel does not affect the set of output histories. It relaxes the transition relation in the sense that messages on the hidden channel do not constrain the TSPA's behavior anymore. Thus, hiding an input channel effectively leads to more underspecification. Any transition is enabled independent of the messages received via the hidden channel, assumed that the messages received via the other input channels are part of the transition's channel valuation.

Let A be a TSPA and let $B \subseteq C_A$ be a set of channels. The restriction of A to the channels in B is defined as $A \upharpoonright B = (I_A \cap B, O_A \cap B, S_A, \iota_A, \delta)$ where

$$\delta = \{(s, \theta, t) \mid \exists \kappa \in C_A^{\rightarrow} \,:\, \theta = \kappa|_B \wedge (s, \kappa, t) \in \delta_A\}.$$

As concrete examples, Fig. 4 depicts the TSPAs resulting from restricting the TSPA *Switch* (cf. Fig. 2) to the set of channels $\{i\}$ and from restricting the TSPA *Switch* to the set of channels $\{o\}$. Restricting the TSPA *Switch* to its input channel yields a TSPA that is still deterministic. However, restricting the TSPA *Switch* to its output channel yields an underspecified TSPA that is not deterministic.

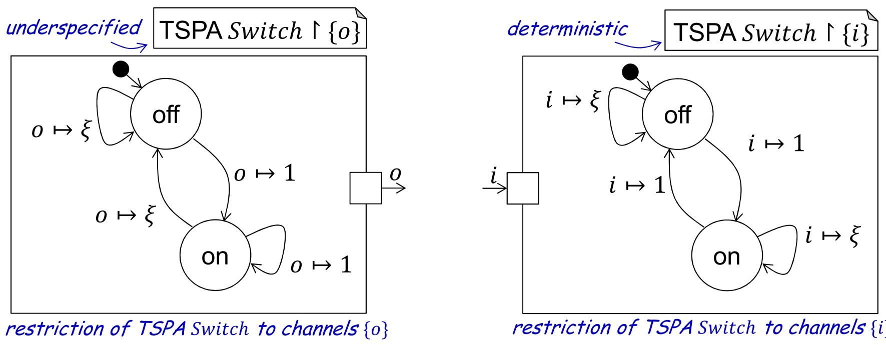


Fig. 4 The restriction of the TSPA *Switch* to the set of channels $\{o\}$ and the restriction of the TSPA *Switch* to the set of channels $\{i\}$

4 Semantics preserving parallel decomposition respecting influences between channels

This paper contributes to the parallel decomposition of deterministic TSPAs. Figure 5 overviews the key idea of the approach: The decomposition method takes a deterministic TSPA representing a component as input. Based on the influence relation between the TSPA's input and output channels, the method decomposes the component into multiple subcomponents (further TSPAs). The parallel composition of the resulting TSPAs yields a TSPA that has the same behaviors as the input TSPA. For example, Fig. 5 indicates that the output channel p is influenced by the input channels i and j. In contrast, the output channel o is solely influenced by the input channel i. The method can be fully automated. Therefore, we obtain an automatic method for refactoring monolithic components into multiple subcomponents such that the behaviors of the composition of the subcomponents are equal to the behaviors of the monolithic component.

The method may produce TSPAs that are not deterministic but *unambiguously specified* as intermediate decomposition results. Intuitively, a TSPA is unambiguously specified if it defines exactly one (infinite) output for every (infinite) input. Every unambiguously specified TSPA can be transformed to a deterministic TSPA having the same behaviors (cf. Sect. 4.1). Thus, the transformation enables the definition of a decomposition procedure for deterministic TSPAs that again yields an architecture of deterministic TSPAs.

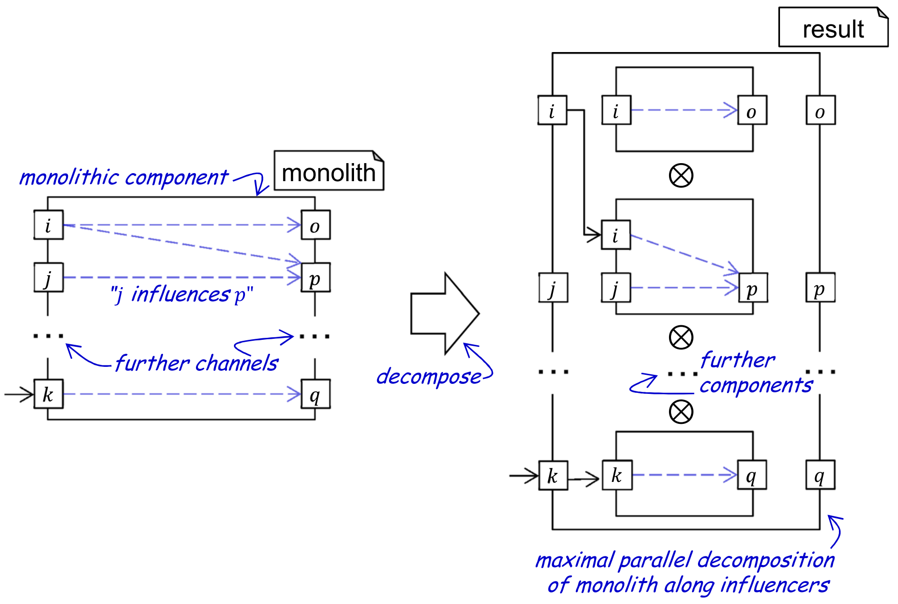


Fig. 5 Schematic representation of a monolithic component that is maximally decomposed along the influences between channels

Section 4.1 formally defines the notion unambiguously specified for TSPAs and presents properties of unambiguously specified TSPAs that are relevant to show the decomposition method's correctness. Afterwards, Sect. 4.2 defines the influence relation between channels of a TSPA. Then, Sect. 4.3 presents a decision procedure for determining whether an input channel of a TSPA influences an output channel of the same TSPA. Subsequently, Sect. 4.4 presents the fully automatic decomposition method based on the channel influence relation.

4.1 Unambiguously specified TSPAs

Hiding an input channel in a deterministic TSPA might result in a TSPA that is by definition not deterministic, but behaves as if it was deterministic from a black-box viewpoint. For example, this is because the TSPA's reachable part is deterministic and there exists a non-reachable part that is not deterministic. Figure 6 depicts a concrete example: The TSPA D is deterministic, whereas restricting it to its output channel yields a TSPA that exhibits the single behavior $\{o \mapsto \xi\}^{\infty}$, thus behaves deterministically from a black-box viewpoint. However, the restricted TSPA is internally non-deterministic, because of the non-reachable state b containing underspecification regarding the message sent via the output channel.

A TSPA might also be not deterministic and have multiple executions for the same inputs that produce the same outputs. In such a case, the TSPA is also not deterministic but behaves as if it was deterministic from a black-box viewpoint. Figure 7 depicts a concrete example: The TSPA U is not deterministic and exhibits the single behavior $\{i \mapsto \xi, o \mapsto \xi\}^{\infty}$. Therefore, it behaves deterministically from a black-box viewpoint. TSPAs that behave deterministically from a black-box viewpoint are *unambiguously specified*:

Definition 1 A TSPA A is unambiguously specified iff

$$\forall i \in I_A^{\Omega} : |\{\alpha \in behs(A) \mid \alpha|_I = i\}| = 1.$$

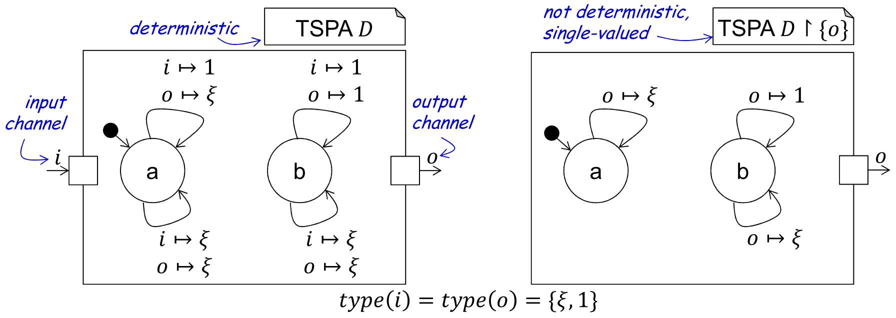


Fig. 6 Deterministic TSPA D and underspecified and unambiguously specified TSPA $D \upharpoonright \{o\}$ resulting from hiding the input channel i in D

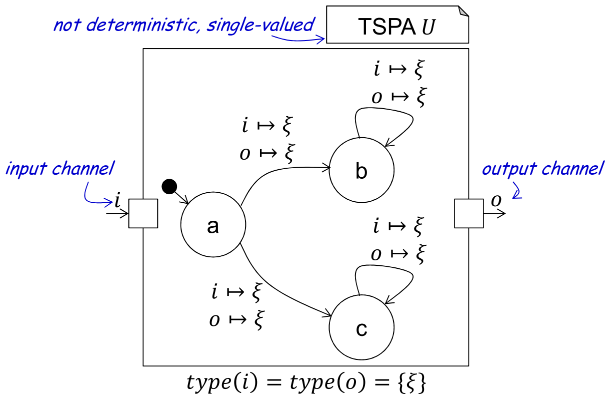


Fig. 7 Underspecified TSPA that behaves deterministically from a black-box viewpoint

The notion *unambiguously specified* for TSPAs and infinite behaviors is related to the notion *single-valued* for finite transductions of transducers (Weber and Klemm 1995; Weber 1998; Béal and Carton 2002). According to Weber and Klemm (1995), Weber (1998), and Béal and Carton (2002), a transducer is single-valued if it maps each input sequence to at most one output sequence. In contrast, we require that each input is mapped to exactly one output. Further, in each computation step, a transducer may map a single input symbol to a sequence of output symbols, whereas a TSPA maps one input channel valuation to exactly one output channel valuation.

Our approach aims at decomposing deterministic TSPAs. It is easy to see that every deterministic TSPA is also unambiguously specified but that the opposite does not necessarily hold. However, for every unambiguously specified TSPA, it is possible to construct an equivalent deterministic TSPA, i.e., the unambiguously specified and the deterministic TSPAs have the same behaviors.

Theorem 1 *For every unambiguously specified TSPA A, there exists a deterministic TSPA D with $behs(A) = behs(D)$.*

Proof (Sketch) A TSPA is interpretable as a special transducer over infinite words where all states are final. Sufficient and necessary conditions enabling the determinization of transducers over infinite words where all states are final are studied in Béal and Carton (2000, 2002).

Specifically, a TSPA is interpretable as a transducer over infinite words where

- Each transition transduces exactly one input symbol to exactly one output symbol,
- There is exactly one initial state,
- All states are final, and
- The transducer has no cyclic path with an empty output.

It has been shown that a transducer over infinite words where all states are final, the transducer has no constant states, and the transducer has no cyclic path with an empty output can be determinized, if the transducer obtained after removing all *constant states* (Béal and Carton 2000, 2002) satisfies the *twinning property* (Béal and Carton 2000, 2002). When transferring these notions to TSPAs, the TSPA A obtained from removing the constant states from an unambiguously specified TSPA is a TSPA that satisfies $\forall i \in I_A^\Omega : |\{\alpha \in behs(A) \mid \alpha|_{I_A} = i\}| \leq 1$. If this TSPA did not have the twinning property, then there would exist an input $i \in I_A^\Omega$ such that $|\{\alpha \in behs(A) \mid \alpha|_{I_A} = i\}| \geq 2$. Furthermore, Béal and Carton (2000, 2002) present a construction that can be used for transforming an unambiguously specified TSPA to an equivalent deterministic TSPA. The construction is a subset construction on the TSPA obtained from removing all unreachable states. $\square$

Thus, every unambiguously specified TSPA can be transformed to a TSPA in which the output in any time unit only depends on the current input and state.

The following introduces general properties of unambiguously specified TSPAs that are later used for proving the correctness of the decomposition method. Two unambiguously specified TSPAs are equivalent if, and only if, one of the automata is a refinement of the other automaton:

Theorem 2 *Let A and B be unambiguously specified TSPAs with $I_A = I_B$ and $O_A = O_B$. Then, $behs(A) \subseteq behs(B)$ if, and only if, $behs(A) = behs(B)$.*

Proof Let A and B be given as above.

"$\Rightarrow$": Assume $behs(A) \subseteq behs(B)$. Let $I = I_A$ and $O = O_A$. Suppose towards a contradiction $behs(B) \nsubseteq behs(A)$. Then, there exists a behavior $b \in behs(B)$ such that $b \notin behs(A)$. As A is unambiguously specified, $I_A = I_B$ and $O_A = O_B$, there exists a behavior $b' \in behs(A)$ with $b'|_I = b|_I$. As $b \notin behs(A)$ and $b'|_I = b|_I$, we have that $b'|_O \neq b|_O$. As $behs(A) \subseteq behs(B)$, it holds that $b' \in behs(B)$. This contradicts that B is unambiguously specified, because $b, b' \in behs(B)$, $b|_I = b'|_I$ and $b|_O \neq b'|_O$ implies for $i = b|_I$ that $|\{\alpha \in behs(B) \mid \alpha|_I = i\}| \geq 2$.

"$\Leftarrow$": $behs(A) = behs(B)$ implies $behs(A) \subseteq behs(B)$. $\square$

TSPAs do not influence the behaviors of each other when executed in parallel, i.e., when neither of the TSPAs has an output channel that is an input channel of the respective other TSPA. Thus, the parallel composition of two unambiguously specified TSPAs is again an unambiguously specified TSPA:

Theorem 3 *Let A and B be two compatible unambiguously specified TSPAs such that $O_A \cap I_B = O_B \cap I_A = \emptyset$. Then, $A \otimes B$ is an unambiguously specified TSPA.*

Proof Let A and B be given as above and let $K = A \otimes B$. We need to show that $|\{\alpha \in behs(K) \mid \alpha|_{I_K} = i\}| = 1$ for all $i \in I_K^\Omega$.

(1) We first show that $|\{\alpha \in behs(K) \mid \alpha|_{I_K} = i\}| > 0$ for all $i \in I_K^\Omega$: Let $i \in I_K^\Omega$. As A and B are unambiguously specified, there exist behaviors $b \in behs(A)$ and

$b' \in behs(B)$ such that $b|_{I_A} = i|_{I_A}$ and $b'|_{I_B} = i|_{I_B}$. Let $\sigma = s_0, \theta_0, s_1, \theta_1 \dots$ be an execution of A such that $beh(\sigma) = b$ and let $\tau = s'_0, \kappa_0, s'_1, \kappa_1 \dots$ be an execution of B such that $beh(\tau) = b'$. As σ and τ are executions, we have that $s_0 = \iota_A$ and $s'_0 = \iota_B$ and $(s_t, \theta_t, s_{t+1}) \in \delta_A$ and $(s'_t, \kappa_t, s'_{t+1}) \in \delta_B$ for all $t \in \mathbb{N}$. As $b|_{I_A} = i|_{I_A}$ and $b'|_{I_B} = i|_{I_B}$, we have that $i|_{I_A \cap I_B} = b|_{I_A \cap I_B} = b'|_{I_A \cap I_B}$. Hence, for all $t \in \mathbb{N}$, we have that $\theta_t|_{I_A \cap I_B} = \kappa_t|_{I_A \cap I_B}$. For all $t \in \mathbb{N}$, we define $\mu_t \in C_K^{\rightarrow}$ as follows: $\mu_t(c) = \theta_t(c)$, if $c \in C_A$, and $\mu_t(c) = \kappa_t(c)$, if $c \in C_B \backslash C_A$. Then, by definition $\mu_t|_{C_A} = \theta_t$. Further, $\mu_t|_{C_B} = \kappa_t$ because $\theta_t|_{I_A \cap I_B} = \kappa_t|_{I_A \cap I_B}$ and by definition $\mu_t|_{C_B \backslash C_A} = \kappa_t|_{C_B \backslash C_A}$. Thus, by definition of TSPA composition, we have that $((s_t, s'_t), \mu_t, (s_{t+1}, s'_{t+1})) \in \delta_K$ for all $t \in \mathbb{N}$. This implies with $s_0 = \iota_A$ and $s'_0 = \iota_B$ that $e = (s_0, s'_0), \mu_0, (s_1, s'_1), \mu_1 \dots$ is an execution of K with $beh(e)|_{I_K} = i$.

(2) We now show that $|\{\alpha \in behs(K) \mid \alpha|_{I_K} = i\}| < 2$ for all $i \in I_K^{\Omega}$:

Suppose towards a contradiction there exist $i \in I_K^{\Omega}$ and $\alpha, \beta \in behs(K)$ such that $\alpha|_{I_K} = \beta|_{I_K} = i$ and $\alpha \neq \beta$. Thus, $\alpha|_{O_K} \neq \beta|_{O_K}$. As α, β are behaviors of K, there exist executions σ and τ of K such that $\alpha = beh(\sigma)$ and $\beta = beh(\tau)$. Let $\sigma = (s_0^A, s_0^B), \alpha_0, (s_1^A, s_1^B), \alpha_1 \dots$ be an execution of K such that $beh(\sigma) = \alpha$. Further, let $\tau = (s_0'^A, s_0'^B), \beta_0, (s_1'^A, s_1'^B), \beta_1 \dots$ be an execution of K such that $beh(\tau) = \beta$. Using the definitions of execution and composition, we obtain that $\sigma_A = s_0^A, \alpha_0|_{C_A}, s_1^A, \alpha_1|_{C_A} \dots$ and $\tau_A = s_0'^A, \beta_0|_{C_A}, s_1'^A, \beta_1|_{C_A} \dots$ are execution of A. Similarly, we have that $\sigma_B = s_0^B, \alpha_0|_{C_B}, s_1^B, \alpha_1|_{C_B} \dots$ and $\tau_B = s_0'^B, \beta_0|_{C_B}, s_1'^B, \beta_1|_{C_B} \dots$ are executions of B. As $O_K = O_A \cup O_B$ and $\alpha|_{O_K} \neq \beta|_{O_K}$, it holds that $beh(\sigma_A) \neq beh(\tau_A)$ or $beh(\sigma_B) \neq beh(\tau_B)$. Without loss of generality, assume $beh(\sigma_A) \neq beh(\tau_A)$. Then, $\alpha|_{I_K} = \beta|_{I_K}$ implies $beh(\sigma_A)|_{I_A} = beh(\tau_A)|_{I_A}$ since $I_A \cap O_B = \emptyset$ and thus $I_A \subseteq I_K$ by definition of composition. This contradicts that A is unambiguously specified because $beh(\sigma_A), beh(\tau_A) \in behs(A)$ and $beh(\sigma_A)|_{I_A} = beh(\tau_A)|_{I_A}$ and $beh(\sigma_A) \neq beh(\tau_A)$. □

The TSPA obtained from hiding an unambiguously specified TSPA's output channel is again an unambiguously specified TSPA. Hiding an input channel does usually not preserve the unambiguously specified property.

Theorem 4 *Let A be an unambiguously specified TSPA and let $o \in O_A$ be an output channel of A. Then, $A \upharpoonright (C_A \backslash \{o\})$ is unambiguously specified.*

Proof Let A and o be given as above. Let $B = A \upharpoonright (C_A \backslash \{o\})$. Suppose B is not unambiguously specified. Then there exist executions $\sigma = s_0, \theta_0, s_1, \theta_1 \dots$ and $\tau = s'_0, \kappa_0, s'_1, \kappa_1 \dots$ of B such that $beh(\sigma)|_{I_B} = beh(\tau)|_{I_B}$ and $beh(\sigma) \neq beh(\tau)$. By definition of TSPA restriction, this implies there exist executions $\sigma' = s_0, \theta'_0, s_1, \theta'_1 \dots$ and $\tau' = s'_0, \kappa'_0, s'_1, \kappa'_1 \dots$ of A such that $\theta_i = \theta'_i|_B$ and $\kappa_i = \kappa'_i|_B$ for all $i \in \mathbb{N}$. This contradicts that A is unambiguously specified because $beh(\sigma')|_{I_A} = beh(\sigma)|_{I_B} = beh(\tau)|_{I_B} = beh(\tau')|_{I_A}$ and $beh(\sigma') \neq beh(\tau')$ since $beh(\sigma')|_{C_B} = beh(\sigma) \neq beh(\tau) = beh(\tau')|_{C_B}$. □

4.2 An influence relation between channels of components

A component's input channel influences an output channel if the messages sent via the latter depend on the messages received via the former.

Definition 2 (*Channel Influence Relation*) Let $A = (I, O, S, \iota, \delta)$ be an unambiguously specified TSPA, let $i \in I$ be an input channel of A, and let $o \in O$ be an output channel of A. The channel i influences the channel o in A (denoted $i \rightsquigarrow_A o$) iff

$$\exists \alpha, \beta \in behs(A) \, : \, \alpha|_{I\backslash\{i\}} = \beta|_{I\backslash\{i\}} \wedge \alpha|_{\{o\}} \neq \beta|_{\{o\}}.$$

The above definition requires that there exist two behaviors α, β with the same messages on all input channels except i such that the behaviors are different on the output channel o. As the inputs are equal on all channels except i, the values received on i are responsible for the differences regarding the possible outputs on o.

The other way around, the channel i does not influence the channel o in A iff for any two possible inputs that are equal on all channels except i, the automaton A always produces the same outputs on o when processing the inputs. More formally, negating the definition we obtain: a channel i does not influence a channel o in A (denoted $i \not\rightsquigarrow_A o$) iff $\forall \alpha, \beta \in behs(A) \, : \, \alpha|_{I\backslash\{i\}} = \beta|_{I\backslash\{i\}} \Rightarrow \alpha|_{\{o\}} = \beta|_{\{o\}}$. Hiding an input channel does not always preserve the unambiguously specified property (cf. Sect. 4.1). However, if an input channel i does not influence an output channel o in an unambiguously specified TSPA A, then hiding the input channel i and all output channels except o results again in an unambiguously specified TSPA:

Theorem 5 *Let A be an unambiguously specified TSPA, let $i \in I_A$ be an input channel of A, and let $o \in O_A$ be an output channel of A. If $i \not\rightsquigarrow_A o$, then $A \upharpoonright (\{o\} \cup I\backslash\{i\})$ is unambiguously specified.*

Proof Let A, i, and o be given as above. Let $B = A \upharpoonright (\{o\} \cup I\backslash\{i\})$. We need to show that $|\{\alpha \in behs(B) \mid \alpha|_{I_B} = h\}| = 1$ for all $h \in I_B^{\Omega}$.

(1) We first show that $|\{\alpha \in behs(B) \mid \alpha|_{I_B} = h\}| > 0$ for all $h \in I_B^{\Omega}$: Let $h \in I_B^{\Omega}$. As A is unambiguously specified and $I_B \subseteq I_A$, there exists a behavior $b \in behs(A)$ such that $b|_{I_B} = h$. Let $\sigma = s_0, \theta_0, s_1, \theta_1 \dots$ be an execution of A such that $beh(\sigma) = b$. Then, by definition of execution $s_0 = \iota_A$ and $(s_j, \theta_j, s_{j+1}) \in \delta_A$ for all $j \in \mathbb{N}$. By definition of restriction, we have that $s_0 = \iota_A = \iota_B$ and $(s_j, \theta_j|_{C_B}, s_{j+1}) \in \delta_B$ for all $j \in \mathbb{N}$. Hence, $\tau = s_0, \theta_0|_{C_B}, s_1, \theta_1|_{C_B} \dots$ is an execution of B with $\tau|_{I_B} = h$.

(2) We now show that $|\{\alpha \in behs(B) \mid \alpha|_{I_B} = h\}| < 2$ for all $h \in I_B^{\Omega}$: Suppose towards a contradiction there exist $h \in I^{\Omega}$ and $\alpha, \beta \in behs(B)$ such that $\alpha|_{I_B} = \beta|_{I_B}$ and $\alpha \neq \beta$. Thus, $\alpha|_{O_B} \neq \beta|_{O_B}$. Let $\sigma = s_0, \alpha_0, s_1, \alpha_1 \dots$ and $\tau = s'_0, \beta_0, s'_1, \beta_1 \dots$ be executions of B such that $beh(\sigma) = \alpha$ and $beh(\tau) = \beta$. As σ and τ are executions of B, we have $s_0 = s'_0 = \iota_B$ and $(s_j, \alpha_j, s_{j+1}), (s'_j, \beta_j, s'_{j+1}) \in \delta_B$ for all $j \in \mathbb{N}$. By definition of TSPA restriction, this implies $s_0 = s'_0 = \iota_B = \iota_A$ and for all $j \in \mathbb{N}$, there exist $\theta_j, \kappa_j \in C_A^{\rightarrow}$ such that $(s_j, \theta_j, s_{j+1}) \in \delta_A$ and $\theta_j|_{C_B} = \alpha_j$ and $(s'_j, \kappa_j, s'_{j+1}) \in \delta_A$ and $\kappa_j|_{C_B} = \beta_j$. Hence, $\sigma' = s_0, \theta_0, s_1, \theta_1 \dots$ and $\tau' = s'_0, \kappa_0, s'_1, \kappa_1 \dots$ are executions of A. This contradicts that $i \not\rightsquigarrow_A o$ because

$beh(\sigma')|_{I\setminus\{i\}} = beh(\sigma')|_{I_B} = beh(\sigma)|_{I_B} = \alpha|_{I_B} = \beta|_{I_B} = beh(\tau')|_{I\setminus\{i\}}$ and $beh(\sigma')|_{\{o\}} = \alpha|_{\{o\}} \neq \beta|_{\{o\}} = beh(\tau')|_{\{o\}}$. □

If there exists a pair of an input and an output channel of an unambiguously specified component such that the input channel does not influence the output channel, it is possible to split the component into a semantically equivalent architecture of two components. This architecture models a new component that is functionally better separated as the original component. This does not only improve the architecture's design but also increases understandability of the architecture and enables independent functional testing. Further, dividing the component also facilitates compositional architecture verification: A property might be independent of the behaviors of one of a composed component's subcomponents. Thus, verifying the property is possible without considering the subcomponent not influencing the property's satisfaction.

Section 4.3 shows that the channel influence relation of every finite unambiguously specified TSPA is decidable. Subsequently, Sect. 4.4 introduces the automated decomposition procedure based on the channel influence relation.

4.3 Deciding influence in unambiguously specified TSPAs

This section shows that it is decidable whether one channel influences another channel in an unambiguously specified finite TSPA. The decision relies on the construction of finite Büchi automata (BA) accepting infinite words (Büchi 1962; Farwer 2002; Safra 1988). BAs are well-known and studied in the automata theory domain. The next section fixes our notation for BAs and recaps decidability properties of BAs used in this paper before Sect. 4.3.1 presents the decision procedure.

A Büchi automaton (BA) is a tuple $(\Sigma, Q, I, F, \delta)$ where

- Σ is a finite alphabet,
- Q is a finite set of states,
- $I \subseteq Q$ is a set of initial states,
- $F \subseteq Q$ is a set of accepting states, and
- $\delta \subseteq Q \times \Sigma \times Q$ is the transition relation.

Let $\mathcal{A} = (\Sigma, Q, I, F, \delta)$ be a BA. A run of $\mathcal{A}$ on a word $w = \sigma_1, \sigma_2 \cdots \in \Sigma^\infty$ starting in a state $q_0 \in Q$ is an infinite sequence $q_0, q_1 \ldots$ such that $(q_{j-1}, \sigma_j, q_j) \in \delta$ for all $j \in \mathbb{N}$ with $j > 0$. The run $q_0, q_1 \ldots$ is accepting if $q_0 \in I$ and $q_i \in F$ for infinitely many $i \in \mathbb{N}$. The accepted language of $\mathcal{A}$ is defined as $\mathcal{L}(\mathcal{A}) = \{w \in \Pi^\infty \mid \text{there exists an accepting run of } \mathcal{A} \text{ on } w\}$. The emptiness problem, asking whether $\mathcal{L}(\mathcal{A}) = \emptyset$ for a BA $\mathcal{A}$ is decidable (Büchi 1962; Farwer 2002). The language of BAs is further closed under intersection (Büchi 1962): For all BAs $\mathcal{A}$ and $\mathcal{B}$, there exist an algorithm for constructing a BA $\mathcal{C}$ such that $\mathcal{L}(\mathcal{C}) = \mathcal{L}(\mathcal{A}) \cap \mathcal{L}(\mathcal{B})$. The languages accepted by BAs are closed under complement: For every BA $\mathcal{A} = (\Sigma, Q, I, F, \delta)$, there is an algorithm for computing a BA $\mathcal{B}$ such that $L(\mathcal{B}) = \Sigma^\infty \setminus \mathcal{L}(\mathcal{A})$ (Safra 1988). We denote the BA accepting the complement of the language accepted by a BA $\mathcal{A}$ with $\overline{\mathcal{A}}$.

4.3.1 Deciding influence

In the remainder of this section, let A be a finite unambiguously specified TSPA, let $i \in I_A$ be an input channel of A and let $o \in O_A$ be an output channel of A. The procedure for checking whether i influences o in A relies on constructing three Büchi automata $\mathcal{A}$, $\mathcal{I}$, and $\mathcal{O}$.

- The automaton $\mathcal{A}$ encodes all tuples of behaviors of A.
- The automaton $\mathcal{I}$ models the set of all tuples of behaviors in C_A^Ω that are equal on all input channels in $I_A \backslash \{i\}$.
- The automaton $\mathcal{O}$ encodes the set of all tuples of behaviors in C_A^Ω that are equal on the output channel o.

Thus, the automaton accepting $\mathcal{L}(\mathcal{A}) \cap \mathcal{L}(\mathcal{I}) \cap \mathcal{L}(\overline{\mathcal{O}})$ accepts all tuples of behaviors of A that are equal on the input channels in $I_A \backslash \{i\}$ and not equal on the output channel o. We show that $\mathcal{L}(\mathcal{A}) \cap \mathcal{L}(\mathcal{I}) \cap \mathcal{L}(\overline{\mathcal{O}}) = \emptyset$ if, and only if, i does not influence o in A.

The Büchi automaton $\mathcal{A}$ that encodes all tuples of behaviors of A is constructed as follows:

$\mathcal{A} = (C_A^{\rightarrow} \times C_A^{\rightarrow}, S_A \times S_A, \{(\iota_A, \iota_A)\}, S_A \times S_A, \delta)$, where

$\delta = \{((s,u),(a,b),(t,v)) \mid (s,a,t),(u,b,v) \in \delta_A\}$

As A is finite, $C_A^{\rightarrow}$ and S_A are finite. Hence, $C_A^{\rightarrow} \times C_A^{\rightarrow}$ and $S_A \times S_A$ are finite. This implies that δ is finite. Therefore, $\mathcal{A}$ is a well-defined BA.

Theorem 6 *For all $\alpha, \beta \in C_A^\Omega$, it holds that*

$$\alpha, \beta \in \mathit{behs}(A) \Leftrightarrow (\alpha.0, \beta.0),(\alpha.1,\beta.1) \cdots \in \mathcal{L}(\mathcal{A}).$$

Proof Let $\alpha, \beta \in C_A^\Omega$.

"$\Rightarrow$": Assume it holds that $\alpha, \beta \in \mathit{behs}(A)$. Then, there exist two executions $\sigma = s_0, \theta_0, s_1, \theta_1 \cdots \in \mathit{execs}(A)$ and $\tau = s'_0, \kappa_0, s'_1, \kappa_1 \cdots \in \mathit{execs}(A)$ such that $\mathit{beh}(\sigma) = \alpha$ and $\mathit{beh}(\tau) = \beta$. By definition of execution we have that $s_0 = s'_0 = \iota_A$ and $(s_t, \theta_t, s_{t+1}), (s'_t, \kappa_t, s'_{t+1}) \in \delta_A$ for all $t \in \mathbb{N}$. By definition of the transition relation δ of the BA $\mathcal{A}$, this implies $((s_t, s'_t),(\theta_t,\kappa_t),(s_{t+1},s'_{t+1})) \in \delta$ for all $t \in \mathbb{N}$. Hence, $(s_0,s'_0),(s_1,s'_1)\ldots$ is a run of $\mathcal{A}$ on the word $(\theta_0,\kappa_0),(\theta_1,\kappa_1)\ldots$. As all states in $\mathcal{A}$ are accepting, all states on the run are accepting. As further $(s_0,s'_0) = (\iota_A,\iota_A)$, we have that the run is accepting. Thus, it holds that $(\theta_0,\kappa_0),(\theta_1,\kappa_1)\ldots$ is a word accepted by $\mathcal{A}$. Observing that $\theta_t = \alpha.t$ and $\kappa_t = \beta.t$ for all $t \in \mathbb{N}$, we can conclude $(\alpha.0,\beta.0),(\alpha.1,\beta.1) \cdots \in \mathcal{L}(\mathcal{A})$.

"$\Leftarrow$": Assume $(\alpha.0,\beta.0),(\alpha.1,\beta.1) \cdots \in \mathcal{L}(\mathcal{A})$. This implies there exists an accepting run $\sigma = (s_0,s'_0),(s_1,s'_1)\ldots$ on the word $(\alpha.0,\beta.0),(\alpha.1,\beta.1)\ldots$ in $\mathcal{A}$. Thus, we have $(s_0,s'_0) = (\iota_A,\iota_A)$ and $((s_t,s'_t),(\alpha.t,\beta.t),(s_{t+1},s'_{t+1})) \in \delta$ for all $t \in \mathbb{N}$ where δ is the transition relation of $\mathcal{A}$. Using the definition of the transition relation δ of $\mathcal{A}$, the above implies $(s_t,\alpha.t,s_{t+1}),(s'_t,\beta.t,s'_{t+1}) \in \delta_A$ for all $t \in \mathbb{N}$. Hence, by definition of execution $\sigma = s_0,\alpha.0,s_1,\alpha.1 \cdots \in \mathit{execs}(A)$ and $\tau = s'_0,\beta.0,s'_1,\beta.1 \cdots \in \mathit{execs}(A)$.

From observing that $beh(\sigma) = \alpha$ and $beh(\tau) = \beta$, we can conclude that $\alpha, \beta \in behs(A)$. $\square$

The constructions of the BAs $\mathcal{I}$ and $\mathcal{O}$ are analogous to each other. We thus first present a more general construction before defining $\mathcal{I}$ and $\mathcal{O}$. Let $B \subseteq C_A$ be a set of channels of A. The BA $\mathcal{E}(B)$ encoding all pairs of behaviors in C_A^Ω that are equal on the channels in B is constructed as follows:

$$\mathcal{E}(B) = (C_A^{\rightarrow} \times C_A^{\rightarrow}, \{\top\}, \{\top\}, \{\top\}, \delta) \text{ where } \delta = \{(\top, (a_1, a_2), \top) \mid a_1|_B = a_2|_B\}.$$

As A is finite, $C_A^{\rightarrow}$ is finite. Thus, $C_A^{\rightarrow} \times C_A^{\rightarrow}$ is finite. Further, $\mathcal{E}(B)$ has exactly one state. Hence, δ is finite and $\mathcal{E}(B)$ is well-defined.

Theorem 7 *Let $B \subseteq C_A$. For all behaviors $\alpha, \beta \in C_A^\Omega$ it holds that* $\alpha|_B = \beta|_B \Leftrightarrow (\alpha.0, \beta.0), (\alpha.1, \beta.1) \cdots \in \mathcal{L}(\mathcal{E}(B))$.

Proof Let $B \subseteq C_A$ and let $\alpha, \beta \in C_A^\Omega$.

"$\Rightarrow$": Assume $\alpha|_B = \beta|_B$. This implies $\alpha.t|_B = \beta.t|_B$ for all $t \in \mathbb{N}$. Thus, by definition of the transition relation of $\mathcal{E}(B)$, we have that $(\top, (\alpha.t, \beta.t), \top) \in \delta$ for all $t \in \mathbb{N}$ where δ is the transition relation of $\mathcal{E}(B)$. Using the definition of accepting run, we have that $\top, \top, \top \ldots$ is an accepting run on the word $(\alpha.0, \beta.0), (\alpha.1, \beta.1) \ldots$ in $\mathcal{E}(B)$. Thus, $(\alpha.0, \beta.0), (\alpha.1, \beta.1), (\alpha.2, \beta.2) \cdots \in \mathcal{L}(\mathcal{E}(B))$.

"$\Leftarrow$": Assume $\gamma = (\alpha.0, \beta.0), (\alpha.1, \beta.1) \cdots \in \mathcal{L}(\mathcal{E}(B))$. Then, there exists an accepting run σ of $\mathcal{E}(B)$ on the word γ. As $\top$ is the only state of $\mathcal{E}(B)$, we have that $\sigma.t = \top$ for all $t \in \mathbb{N}$. As σ is a run of $\mathcal{E}(B)$, we have $(\sigma.t, (\alpha.t, \beta.t), \sigma.(t+1)) = (\top, (\alpha.t, \beta.t), \top) \in \delta$ for all $t \in \mathbb{N}$ where δ is the transition relation of $\mathcal{E}(B)$. By definition of the transition relation, this implies $\alpha.t|_B = \beta.t|_B$ for all $t \in \mathbb{N}$. This is equivalent to $\alpha|_B = \beta|_B$. $\square$

The Büchi automata $\mathcal{I}$ and $\mathcal{O}$ are defined as $\mathcal{I} = \mathcal{E}(I_A \backslash \{i\})$ and $\mathcal{O} = \mathcal{E}(\{o\})$.

Theorem 8 *It holds $i \rightsquigarrow_A o$ iff $\mathcal{L}(\mathcal{A}) \cap \mathcal{L}(\mathcal{I}) \cap \mathcal{L}(\overline{\mathcal{O}}) \neq \emptyset$.*

Proof Using Theorems 6 and 7, we have for all behaviors $\alpha, \beta \in C_A^\Omega$:

$$\alpha, \beta \in behs(A) \Leftrightarrow (\alpha.0, \beta.0), (\alpha.1, \beta.1) \cdots \in \mathcal{L}(\mathcal{A}) \text{ and}$$
$$\alpha|_{I_A \backslash \{i\}} = \beta|_{I_A \backslash \{i\}} \Leftrightarrow (\alpha.0, \beta.0), (\alpha.1, \beta.1) \cdots \in \mathcal{L}(\mathcal{I}) \text{ and}$$
$$\alpha|_{\{o\}} \neq \beta|_{\{o\}} \Leftrightarrow (\alpha.0, \beta.0), (\alpha.1, \beta.1) \cdots \in \mathcal{L}(\overline{\mathcal{O}}).$$

Combining the three equivalences, we obtain for all behaviors $\alpha, \beta \in C_A^\Omega$:

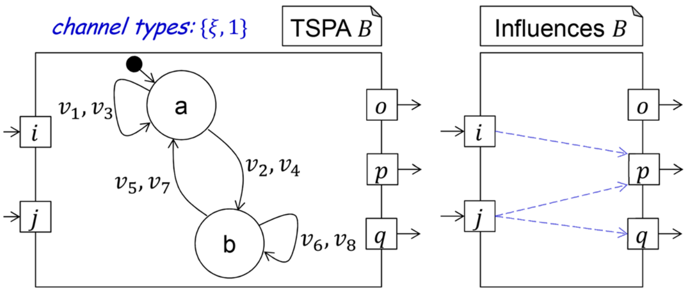
$$(\alpha, \beta \in behs(A) \wedge \alpha|_{I_A \backslash \{i\}} = \beta|_{I_A \backslash \{i\}} \wedge \alpha|_{\{o\}} \neq \beta|_{\{o\}}) \Leftrightarrow$$
$$(\alpha.0, \beta.0), (\alpha.1, \beta.1) \cdots \in \mathcal{L}(\mathcal{A}) \cap \mathcal{L}(\mathcal{I}) \cap \mathcal{L}(\overline{\mathcal{O}}).$$

"⇒": Assume $i \rightsquigarrow_A o$. Then, there exist behaviors $\alpha, \beta \in behs(A) : \alpha|_{I\setminus\{i\}} = \beta|_{I\setminus\{i\}} \wedge \alpha|_{\{o\}} \neq \beta|_{\{o\}}$. Using the above, this implies $(\alpha.0, \beta.0), (\alpha.1, \beta.1) \cdots \in \mathcal{L}(\mathcal{A}) \cap \mathcal{L}(\mathcal{I}) \cap \mathcal{L}(\overline{\mathcal{O}})$. Thus, $\mathcal{L}(\mathcal{A}) \cap \mathcal{L}(\mathcal{I}) \cap \mathcal{L}(\overline{\mathcal{O}}) \neq \emptyset$.

"⇐": Assume $\mathcal{L}(\mathcal{A}) \cap \mathcal{L}(\mathcal{I}) \cap \mathcal{L}(\overline{\mathcal{O}}) \neq \emptyset$. This implies that there exists a word $(\alpha.0, \beta.0), (\alpha.1, \beta.1) \cdots \in \mathcal{L}(\mathcal{A}) \cap \mathcal{L}(\mathcal{I}) \cap \mathcal{L}(\overline{\mathcal{O}})$. Let α, β be two behaviors defined by: $\alpha = \alpha.0, \alpha.1 \cdots \in C_A^{\Omega}$ and $\beta = \beta.0, \beta.1 \cdots \in C_A^{\Omega}$. Using the above, we obtain $\alpha, \beta \in behs(A) \wedge \alpha|_{I_A\setminus\{i\}} = \beta|_{I_A\setminus\{i\}} \wedge \alpha|_{\{o\}} \neq \beta|_{\{o\}}$. This implies $i \rightsquigarrow_A o$. □

For example, Fig. 8 depicts the TSPA B. The TSPA has the two input channels i and j and the three output channels o, p, and q. Each channel has the type $\{\xi, 1\}$. The graphical representation of the TSPA uses eight transition labels that are defined in the table, which is depicted at the bottom of Fig. 8. The top-right part of Fig. 8 sketches the influence relation between the input and output channels of the TSPA B. For instance, the channel i influences the channel p, but the channel i does not influence the channel q. From the graphical representation, the channel influence relation in the TSPA B is not obvious. Using the procedure presented in this section, the channel influence relation can be computed fully automatically.

The following example demonstrates the construction to show that the input channel i influences the output channel p in the TSPA B. In the following, we construct the three BAs A, I, and $\overline{O}$ for determining whether the input channel i influences the output channel p. From these BAs, we construct the BA $A \times I \times \overline{O}$ that recognizes



Label	i	j	o	p	q
v_1	ξ	ξ	1	ξ	ξ
v_2	ξ	1	1	1	ξ
v_3	1	ξ	1	1	ξ
v_4	1	1	1	ξ	ξ
v_5	ξ	ξ	1	1	1
v_6	ξ	1	1	ξ	1
v_7	1	ξ	1	ξ	1
v_8	1	1	1	1	1

Fig. 8 TSPA where one output channel is not influenced by any input channel, one output channel is influenced by one input channel, and one output channel is influenced by two input channels

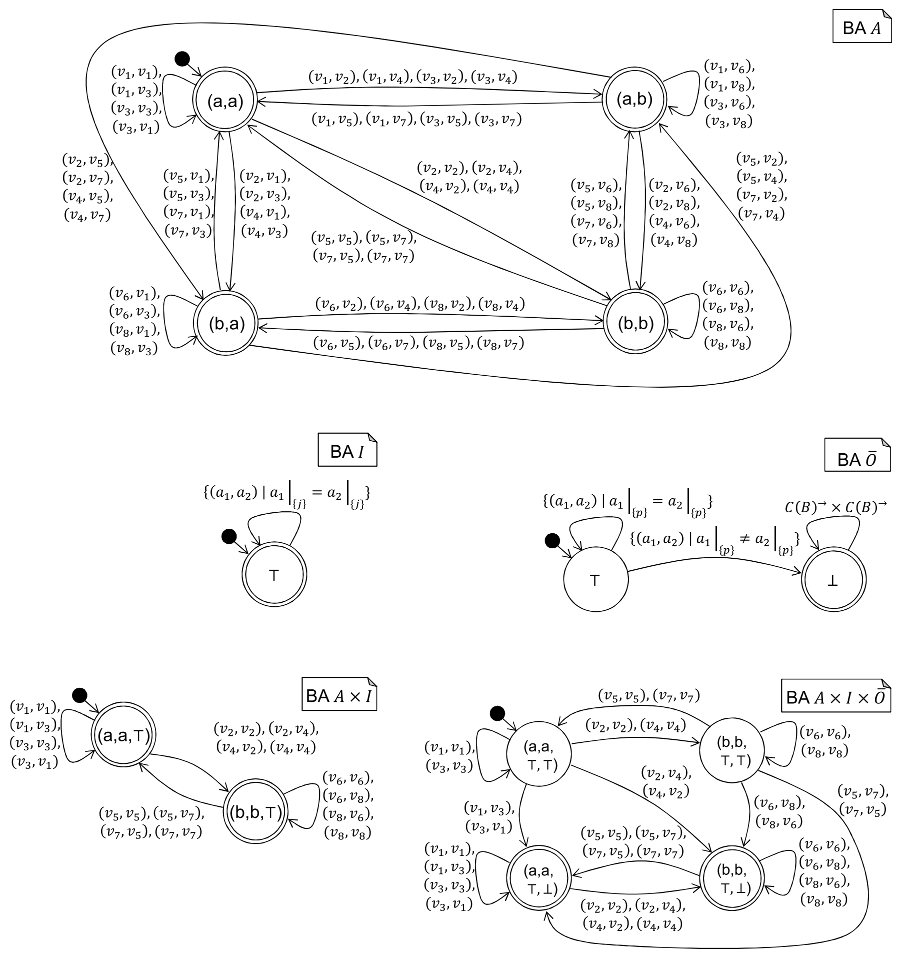

BA A
(a,a)
(a,b)
(b,a)
(b,b)
(v_1,v_1), (v_1,v_3), (v_3,v_3), (v_3,v_1)
(v_1,v_2), (v_1,v_4), (v_3,v_2), (v_3,v_4)
(v_1,v_5), (v_1,v_7), (v_3,v_5), (v_3,v_7)
(v_1,v_6), (v_1,v_8), (v_3,v_6), (v_3,v_8)
(v_2,v_5), (v_2,v_7), (v_4,v_5), (v_4,v_7)
(v_5,v_1), (v_5,v_3), (v_7,v_1), (v_7,v_3)
(v_2,v_1), (v_2,v_3), (v_4,v_1), (v_4,v_3)
(v_2,v_2), (v_2,v_4), (v_4,v_2), (v_4,v_4)
(v_5,v_5), (v_5,v_7), (v_7,v_5), (v_7,v_7)
(v_5,v_6), (v_5,v_8), (v_7,v_6), (v_7,v_8)
(v_2,v_6), (v_2,v_8), (v_4,v_6), (v_4,v_8)
(v_5,v_2), (v_5,v_4), (v_7,v_2), (v_7,v_4)
(v_6,v_1), (v_6,v_3), (v_8,v_1), (v_8,v_3)
(v_6,v_2), (v_6,v_4), (v_8,v_2), (v_8,v_4)
(v_6,v_5), (v_6,v_7), (v_8,v_5), (v_8,v_7)
(v_6,v_6), (v_6,v_8), (v_8,v_6), (v_8,v_8)
BA I
$\{(a_1,a_2) \mid a_1|_{\{j\}} = a_2|_{\{j\}}\}$
⊤
BA $\overline{O}$
$\{(a_1,a_2) \mid a_1|_{\{p\}} = a_2|_{\{p\}}\}$
$\{(a_1,a_2) \mid a_1|_{\{p\}} \neq a_2|_{\{p\}}\}$
$C(B)^{\rightarrow} \times C(B)^{\rightarrow}$
⊤
⊥
BA $A \times I$
(a,a,⊤)
(b,b,⊤)
(v_1,v_1), (v_1,v_3), (v_3,v_3), (v_3,v_1)
(v_2,v_2), (v_2,v_4), (v_4,v_2), (v_4,v_4)
(v_5,v_5), (v_5,v_7), (v_7,v_5), (v_7,v_7)
(v_6,v_6), (v_6,v_8), (v_8,v_6), (v_8,v_8)
BA $A \times I \times \overline{O}$
(a,a, ⊤,⊤)
(b,b, ⊤,⊤)
(a,a, ⊤,⊥)
(b,b, ⊤,⊥)
(v_5,v_5), (v_7,v_7)
(v_2,v_2), (v_4,v_4)
(v_1,v_1), (v_3,v_3)
(v_6,v_6), (v_8,v_8)
(v_2,v_4), (v_4,v_2)
(v_1,v_3), (v_3,v_1)
(v_6,v_8), (v_8,v_6)
(v_5,v_7), (v_7,v_5)
(v_5,v_5), (v_5,v_7), (v_7,v_5), (v_7,v_7)
(v_1,v_1), (v_1,v_3), (v_3,v_3), (v_3,v_1)
(v_6,v_6), (v_6,v_8), (v_8,v_6), (v_8,v_8)
(v_2,v_2), (v_2,v_4), (v_4,v_2), (v_4,v_4)

Fig. 9 The BA A models all tuples of behaviors of the TSPA B, which is depicted in Fig. 8. The BA I models the set of all tuples of behaviors in $C(B)^{\Omega}$ that are equal on the input channels in $I_B \backslash \{i\}$. The BA $\overline{O}$ models the set of all tuples of behaviors in $C(B)$ that are not equal on the output channel p. The reachable part of the BA $A \times I$ models all tuples of behaviors of A that are equal on all input channels in $I_B \backslash \{i\} = \{j\}$. The reachable part of the BA $A \times I \times \overline{O}$ models all tuples of behaviors of A that are equal on the input channel j and not equal on the output channel p

the language $\mathcal{L}(A) \cap \mathcal{L}(I) \cap \mathcal{L}(\overline{O})$. Using Theorem 8, the language recognized by $A \times I \times \overline{O}$ is not empty iff the channel i influences the channel p in the TSPA B. The BA A modeling all tuples of behaviors of the TSPA B is graphically illustrated in the top of Fig. 9. This BA uses the same transition labels as the TSPA B, which are defined in Fig. 8. The BAs I and $\overline{O}$ are depicted in the middle of Fig. 9. The BA I models the set of all tuples of behaviors in $C(B)^{\Omega}$ that are equal on all input channels in $I_B \backslash \{i\} = \{j\}$. The BA $\overline{O}$ represents the set of all behaviors in $C(B)^{\Omega}$ that are not equal on the output channel p. The bottom left of Fig. 9 depicts the reachable part of the BA $A \times I$ that accepts the intersection of the languages accepted by the BAs

A and I. Thus, the BA $A \times I$ models the set of all tuples of behaviors of A that are equal on the input channel j. The bottom right of Fig. 9 depicts the reachable part of the BA $A \times I \times \overline{O}$. This BA models all tuples of behaviors of A that are equal on the input channel j and not equal on the output channel p. As the language accepted by this BA is not empty, the channel i influences the channel p. For example, a word accepted by this BA is given by $w = (v_2, v_4) \cdot ((v_5, v_5), (v_2, v_2))^\infty$. The word w represents the behaviors $\alpha = v_2 \cdot (v_5, v_2)^\infty$ and $\beta = v_4 \cdot (v_5, v_2)^\infty$ where $\alpha|_{I_B \setminus \{i\}} = \beta|_{I_B \setminus \{i\}}$ and $\alpha|_p \neq \beta|_p$. Thus, the word w encodes a concrete proof in the form of two behaviors proving that the channel i influences the channel p.

The following example demonstrates the construction to show that the input channel i does not influence the output channel q in the TSPA B. To this effect, we first construct the BA $\overline{O'}$. This BA models all behaviors in $C(B)^\Omega$ that are not equal on the output channel q. Afterwards, we construct the BA $A \times I \times \overline{O'}$, which models all behaviors of A that are equal on all channels in $I_B \setminus \{i\} = \{j\}$ and not equal on the output channel q. The reachable part of the BA $\overline{O'}$ is depicted in the left part of Fig. 10. The right part of Fig. 10 depicts the reachable part of the BA $A \times I \times \overline{O'}$. The language of this BA is empty. Thus, with Theorem 8, the input channel i does not influence the output channel q in the TSPA B: For every input, the output on the channel q does not depend on the input on channel i.

4.4 Decomposing components along influencers

Composing the TSPAs obtained from decomposing a TSPA into two compatible TSPAs in parallel, such that the composition contains exactly the channels of the original, always results in a TSPA that generalizes the behavior of the original. This holds because hiding an input channel from a TSPA removes information that restricts the TSPA's behaviors:

Theorem 9 *Let A be a TSPA and let $D, E \subseteq C_A$ such that $D \cap E \cap O_A = \emptyset$ and $D \cup E = C_A$. Then, $behs(A) \subseteq behs(A \upharpoonright D \otimes A \upharpoonright E)$.*

Proof Let A, D, and E be given as above. Let $X = A \upharpoonright D$ and let $Y = A \upharpoonright E$. X and Y are compatible because $D \cap E \cap O_A = \emptyset$ implies $O_X \cap O_Y = \emptyset$. Let $\sigma = s_0, \theta_0, s_1, \theta_1, \ldots$ be an execution of A. By definition of execution it holds that $s_0 = \iota_A$ and $(s_i, \theta_i, s_{i+1}) \in \delta_A$ for all $i \in \mathbb{N}$. Hence, using the definition of restriction we have that $(s_i, \theta_i|_{C_X}, s_{i+1}) \in \delta_X$ and $(s_i, \theta_i|_{C_Y}, s_{i+1}) \in \delta_Y$ for all $i \in \mathbb{N}$. Thus,

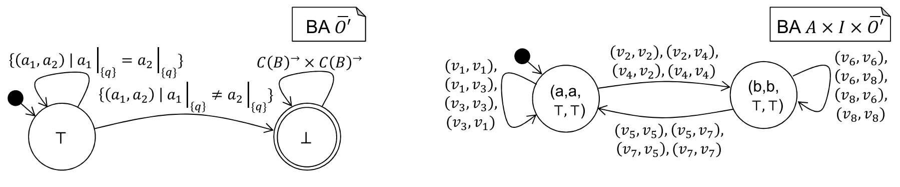

BA $\overline{O'}$
$\{(a_1, a_2) \mid a_1|_{\{q\}} = a_2|_{\{q\}}\}$
$\{(a_1, a_2) \mid a_1|_{\{q\}} \neq a_2|_{\{q\}}\}$
$C(B)^\rightarrow \times C(B)^\rightarrow$
$\top$
$\bot$
BA $A \times I \times \overline{O'}$
$(v_1, v_1), (v_1, v_3), (v_3, v_3), (v_3, v_1)$
(a,a, T, T)
$(v_2, v_2), (v_2, v_4), (v_4, v_2), (v_4, v_4)$
(b,b, T, T)
$(v_5, v_5), (v_5, v_7), (v_7, v_5), (v_7, v_7)$
$(v_6, v_6), (v_6, v_8), (v_8, v_6), (v_8, v_8)$

Fig. 10 The BA $\overline{O'}$ and the reachable part of the BA $A \times I \times \overline{O'}$ that models all tuples of behaviors of A that are equal on the input channel j and not equal on the output channel q

as by assumption $C_X \cup C_Y = C_A$, by definition of TSPA composition, this implies $((s_i, s_i), \theta_i, (s_{i+1}, s_{i+1})) \in \delta_{X \otimes Y}$. Observing that $(s_0, s_0) = (\iota_A, \iota_A)$ is the initial state of $X \otimes Y$, we can conclude that $\kappa = (s_0, s_0), \theta_0, (s_1, s_1), \theta_1, \dots$ is an execution of $X \otimes Y$. Thus, $beh(\kappa) = beh(\sigma) \in behs(X \otimes Y)$. To conclude: for each execution of A, there exists an execution of $X \otimes Y$ such that the executions have the same behaviors. This implies that each behavior of A is also a behavior of $X \otimes Y$. Thus, $behs(A) \subseteq behs(X \otimes Y)$. □.

As hiding may remove information that restrict a TSPA's behaviors, the other direction does not necessarily hold. Thus, the composition of two TSPAs resulting from a decomposition may have behaviors that are not present in the original TSPA. However, if the decomposition is performed along channels that do not influence each other, then the composition of two TSPAs resulting from the decomposition has exactly the same behaviors as the original:

Theorem 10 *Let A be an unambiguously specified TSPA, let $i \in I_A$, and let $o \in O_A$. If $i \not\rightsquigarrow_A o$, then $behs(A \upharpoonright (\{o\} \cup I_A \backslash \{i\}) \otimes A \upharpoonright (C_A \backslash \{o\}) = behs(A)$.*

Proof Let A, i and o be given as above. Let $D = A \upharpoonright (\{o\} \cup I_A \backslash \{i\})$ and let $E = A \upharpoonright (C_A \backslash \{o\})$. As A is unambiguously specified, Theorem 4 guarantees that E is unambiguously specified. As $i \not\rightsquigarrow_A o$, Theorem 5 guarantees that D is unambiguously specified. As D and E are unambiguously specified and $O_D \cap I_E = \{o\} \cap I_A = \emptyset = (O_A \backslash \{o\}) \cap (I_A \backslash \{i\}) = O_E \cap I_D$, using Theorem 3, we have that $D \otimes E$ is unambiguously specified. By definition of D and E, we have $O_D \cap O_E = \{o\} \cap (O_A \backslash \{o\}) = \emptyset$ and $C_D \cup C_E = (\{o\} \cup I_A \backslash \{i\}) \cup (C_A \backslash \{o\}) = C_A$. Hence, with Theorem 9, we have $behs(A) \subseteq behs(D \otimes E)$. Therefore, as A and $D \otimes E$ are unambiguously specified and $O_A = \{o\} \cup (O_A \backslash \{o\}) = O_D \cup O_E = O_{D \otimes E}$ and $behs(A) \subseteq behs(D \otimes E)$, using Theorem 2 we can conclude $behs(A) = behs(D \otimes E)$. □

This enables decomposing components based on channel pairs that do not influence each other. Algorithm 1 is a procedure for iteratively determining a maximal decomposition with respect to the influence relation between channels in a TSPA. The basic operations are TSPA restriction and checking whether there exist channels that influence each other in a TSPA. A procedure for determining whether an input channel influences an output channel is detailed in the previous Sect. 4.3.

Algorithm 1 An algorithm for the parallel decomposition of a TSPA A based on the influences-in-A relation.

```
1:  function DECOMPOSE(TSPA $A$)
2:      if $\exists i \in I_A : \exists o \in O_A : i \not\rightsquigarrow_A o$ then
3:          let $(i, o) \in I_A \times O_A$ such that $i \not\rightsquigarrow_A o$
4:          $D \leftarrow A{\restriction}(\{o\} \cup I_A \setminus \{i\})$
5:          $E \leftarrow A{\restriction}(C_A \setminus \{o\})$
6:          return DECOMPOSE($D$) $\cup$ DECOMPOSE($E$)
7:      else
8:          return $\{A\}$
9:      end if
10: end function
```

For example, decomposing the TSPA B of Fig. 8 with Algorithm 1 yields the decomposition represented by the set $\{B \restriction \{j\}, B \restriction \{o\}, B \restriction \{j, q\}, B \restriction \{i, j, p\}\}$.

5 Elevator control system example revisited

Section 2 presented the software component for an elevator control system (ECS) as inspired by Butting et al. (2017b), Strobl et al. (1999) and Ringert et al. (2016). At some point, the engineers developed a monolithic `ECS` component as depicted in Fig. 1. The `ECS` component is a finite state system (Butting et al. 2017b; Strobl et al. 1999; Ringert et al. 2016) that can be transformed to a finite TSPA (Butting et al. 2017b). The component's implementation has already been shipped but is still available. Due to changed requirements for the elevator's successor version, the team needs to adjust the component's behavior concerning the control of the floor lights in response to the elevator's cabin position. The floor lights are controlled with messages sent via the channels `li1`, `li2`, and `li3`. The elevator's position is indicated by messages received via the channels `at1`, `at2`, and `at3`. Changing the implementation is error-prone as the architecture is monolithic, i.e., changing the implementation may change the component's behavior on channels that are not impacted by the changed requirement. For instance, as the component is not adequately decomposed, changing the component's implementation may result in a change of its behavior on the channels `up` and `down` for steering the elevator cabin, although the behavior on these channels does not need to be adjusted to satisfy the changed requirement. The engineering team is also uncertain which input channels influence which output channels, i.e., whether there are hidden influence dependencies between channels. The team thus uses our method for the automated decomposition of components.

Figure 11 depicts three `ECS` architectures that are obtained as intermediate results during the decomposition of the initial `ECS` implementation. The initial implementation is illustrated in the top-left of Fig. 11.

The decomposition procedure initially detects that the input channel `btn2` does not influence the output channel `li1` (cf. Algorithm 1, l. 2). An automatic procedure for checking whether an input channel influences an output channel is detailedly

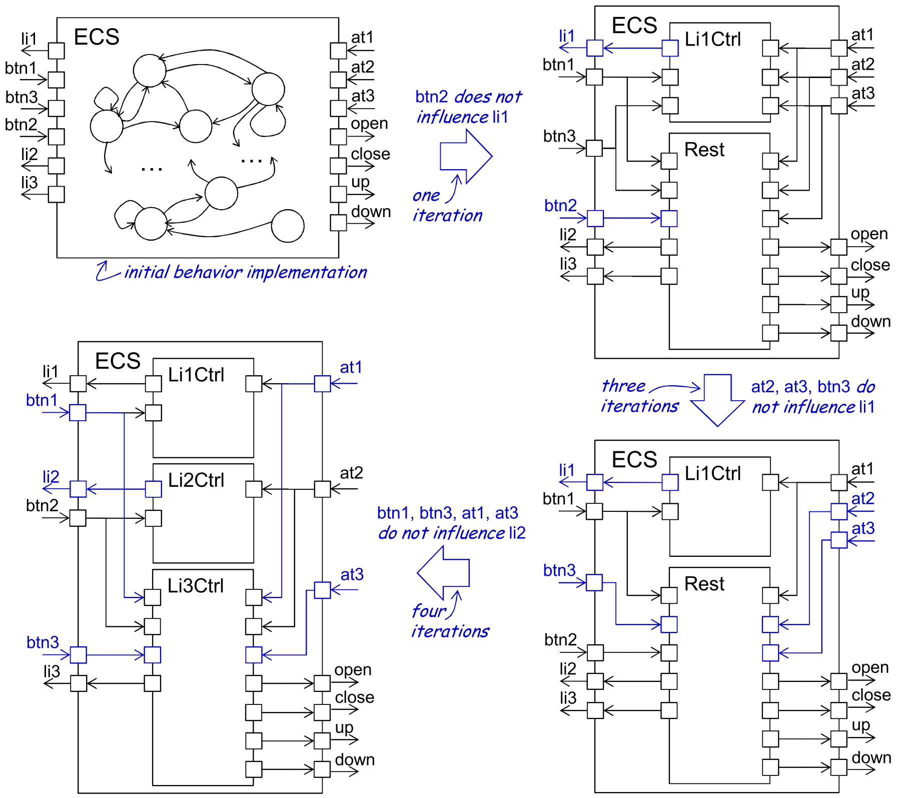

ECS
li1
btn1
btn3
btn2
li2
li3
at1
at2
at3
open
close
up
down
initial behavior implementation
btn2 does not influence li1
one iteration
Li1Ctrl
Rest
three iterations
at2, at3, btn3 do not influence li1
btn1, btn3, at1, at3 do not influence li2
four iterations
Li2Ctrl
Li3Ctrl

Fig. 11 Representation of different intermediate architectures obtained during the automatic decomposition. Top-left describes the initial behavior representation as presented in Fig. 1. The architectures represented clockwise describe intermediate results after various iterations

described in Sect. 4.3.1. The algorithm splits the `ECS` implementation into the two components `Li1Ctrl` and `Rest` (called *D* and *E* in Algorithm 1, ll. 4–5). The resulting architecture is depicted in the top-right of Fig. 11. The component `Li1C-trl` has the single output channel `li1` and the five input channels `btn1`, `btn3`, `at1`, `at2`, `at3`. As the input channel `btn2` does not influence the output channel `li1` in the component `ECS`, the channel `btn2` is no input channel of the component `Li1Ctrl`. At this stage during the decomposition procedure, it is not clear whether other input channels do not influence the output channel `li1`, either. Similarly, at this stage, it has not been detected which channels do not influence the other output channels. Therefore, all input channels of the initial `ECS` component are also input channels of the component `Rest` and all output channels of the initial `ECS` component except the channel `li1` are the output channels of `Rest`.

In the next three iterations of the decomposition, Algorithm 1 detects that the channels `at2` , `at3`, and `btn3` do not influence the channel `li1` in `Li1Ctrl`, either. Therefore, Algorithm 1 decomposes the component `Li1Ctrl` accordingly: The input channels `at2`, `at3`, and `btn3` are removed from the component

Li1Ctrl. As byproducts from the decomposition, the algorithm produces components without output channels. As these components do not sent messages to their environments, they can be safely removed without changing the semantics of the architecture and are not depicted above. The resulting architecture after the decomposition and the removal of the components is depicted in the bottom-right of Fig. 11.

Similarly, in the next four iterations of the decomposition procedure, the algorithm detects that the input channels btn1, btn3, at1, and at3 do not influence the channel li2 in Rest and decomposes the component Rest accordingly. The resulting architecture after removing the components without output channels is depicted in the bottom-left of Fig. 11.

Analogously, the input channels btn1, btn2, at1, and at2 do not influence the channel li3 in Rest. Therefore, the algorithm decomposes the component Rest accordingly. The resulting architecture after removing all components without output channels is depicted Fig. 12. In this architecture, every input channel of every component influences every output channel of the component. Therefore, the decomposition procedure terminates.

By the decomposition procedure's properties, the decomposed component (cf. Fig. 12) is semantically equivalent to the original and clearly better separated regarding the influence relation between channels. From reviewing the new architecture, the engineers now understand that messages emitted via a channel for controlling a floor light only depend on the corresponding elevator cabin position sensor and whether the corresponding request button has been pressed. The implementation of a light controller can now be changed without the threat of accidentally changing the behavior on other channels. They also understand that all input channels influence the channels open, close, up, and down. Thus, the messages the component sends via these channels depend on the messages received via all input

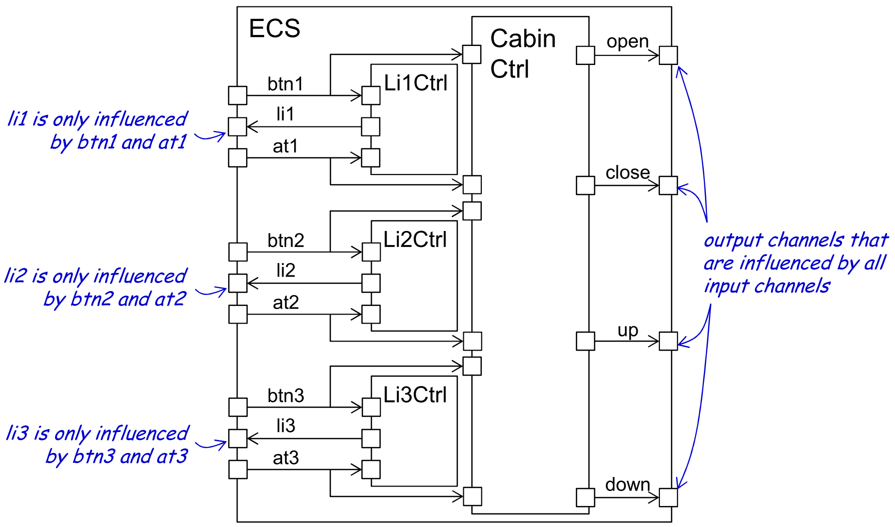

ECS
Cabin Ctrl
Li1Ctrl
Li2Ctrl
Li3Ctrl
btn1
li1
at1
btn2
li2
at2
btn3
li3
at3
open
close
up
down
li1 is only influenced by btn1 and at1
li2 is only influenced by btn2 and at2
li3 is only influenced by btn3 and at3
output channels that are influenced by all input channels

Fig. 12 Semantically equivalent decomposed variant of the ECS

channels. The behavior of the floor lights controlling components and the cabin controlling component can now be unit tested and formally verified individually. As the decomposition is a refactoring, the satisfactions of preexisting symbolic system tests and formally specified requirements for the `ECS` component are preserved.

6 Discussion

Currently, our approach applies only to unambiguously specified and deterministic component implementations. This prevents automated decomposition of component specifications, which usually are underspecified (e.g., by non-determinism). Also, our influence-based decomposition is limited to time-synchronous systems. While these are ubiquitous in embedded and cyber-physical systems, other domains, such as cloud computing, usually rely on event-based message passing. Although FOCUS supports both, non-deterministic specification and untimed communication, applying the notion of channel influencing requires additional research. We consider this as interesting future work.

As our notion of influencing channels establishes relations from input channels to output channels, the resulting decomposition always is parallel, i.e., produces subcomponents connecting a subset of the input channels to a subset of the output channels. Prescribing intermediate channels for more detailed decomposition might be additionally helpful. This also is subject to future research.

Algorithm 2 Parallel decomposition of a TSPA A based on the influences in A relation while respecting pairs of channels that must not be separated.

```
1:  function DECOMPOSE(TSPA $A$, $I \subseteq I_A \times O_A$)
2:      if $\exists i \in I_A : \exists o \in O_A : (i, o) \notin I \wedge i \not\rightsquigarrow_A o$ then
3:          let $(i, o) \in I_A \times O_A$ such that $(i, o) \notin I \wedge i \not\rightsquigarrow_A o$
4:          $D \leftarrow A\upharpoonright(\{o\} \cup I_A \setminus \{i\})$
5:          $E \leftarrow A\upharpoonright(C_A \setminus \{o\})$
6:          return DECOMPOSE($D$) $\cup$ DECOMPOSE($E$)
7:      else
8:          return $\{A\}$
9:      end if
10: end function
```

The algorithm for the decomposition of components as presented in Sect. 4 always computes a maximal decomposition: It decomposes the input component (respectively the intermediate decomposition results) as long as there exists at least one input/output channel pair where the input channel does not influence the output channel. A user might consider an input channel to be associated with an output channel, although the input channel does not influence the output channel. This might be the case, for instance, because the channels are functionally related. In the ECS example (cf. Fig. 12), for instance, a user might consider each button-channel (`btn1`, `btn2`, `btn3`) to be associated with each light-channel (`li1`, `li2`, `li3`). This might be the case, because the channels are functionally related in the sense that they are all used for steering different floor lights. In such cases, the user might be not interested in a maximal decomposition of the system. Instead, she might be interested in a decomposition procedure that does definitely not decompose predefined pairs of input and output channels, disregarding whether the input channel of a pair influences the output channel of the pair. Algorithm 2 is an adjusted version of Algorithm 1 for accomplishing this task. The algorithm additionally takes a set I (for inseparable) of pairs of input and output channels of the TSPA as input. The algorithm separates an input channel from an output channel iff the input channel does not influence the output channel and the input/output channel pair is no element of the set I containing the pairs of inseparable channels. Thus, the adjusted algorithm computes a maximal decomposition while respecting pairs of channels that should not be separated from each other.

Focus operates on component instances, i.e., the information about component types is implicit only. Consequently, our approach produces component instances also. If these, as illustrated by components `Li1Ctrl`, `Li2Ctrl`, and `Li3Ctrl` of Fig. 12, are equivalent, we could deduce type information and synthesize new component types for patterns identified through decomposition accordingly. This might facilitate component reuse. In this case, the decomposition would derive a new component type `LightCtrl` and instantiate it three times accordingly.

This paper presents the theoretical foundations of automated decomposition along pairs of channels that influence each other. The automated decomposition rests on the assumption that the systems largely comprise components that are free of side effects, i.e., “pure”, Focus components. Where components yield side effects, checking whether system functions or capabilities have changed demands additional measures, such as sufficient test coverage or manual analysis. Another challenge in using our method for automated decomposition is its scaling-up. For instance, the `ECS` sketched in Fig. 1 and based on Butting et al. (2019) will be translated into a TSPA with a large number of transitions, which might be too large for human comprehension and reproduction in this paper. However, usually, the models that engineers start with are specified manually and, from our experience, thus, small and comprehensible.

Our approach for automated decomposition is limited to Focus-compatible architectures, which belong to a group of more formal modeling techniques that might not yet be state-of-practice. For modelers operating within less well-defined or incompatible technological spaces, we consider our contribution towards the automated evolution of software architecture models a relevant case in point for at least investigating the benefits of more formal modeling techniques in practice. Whether the results from the decomposition are useful for engineers needs further evaluation including real systems and engineers. We consider this interesting future work.

7 Related work

While agile architecting has been under investigation lately, e.g., driven by change impact analysis (Díaz et al. 2013), cost-and-risk analysis (Poort 2014), or for specific domains (Díaz et al. 2014), there are only a few approaches towards agile architecting with semantically well-defined ADLs and these usually rest on Focus or the π-calculus (Milner 1999).

7.1 Automata decomposition

The decomposition of automata has been subject to research for several decades. For instance, our contribution also relates to parallel decomposition of automata (Gerace and Gestri 1967). While it also aims at a practical decomposition (Nozaki 1978), i.e., the resulting components yield fewer states than the component they were decomposed from, in contrast to more current related work (Uygur and Sattler 2013), it operates specifically on time-synchronous port automata. Similarly, while port automata generally can be decomposed into compositions consisting of FIFOs and XORs only (Koehler and Clarke 2009), this resulting granularity does not produce automata accessible for constructive systems engineering. Related decomposition approaches also exist for probabilistic automata (Carlsson and Yu 2015) or linear automata (Plotkin and Plotkin 2015), none of which consider automated decomposition in the presence of influencing channels.

There also are related approaches in the parallel decomposition of processes (Jongmans et al. 2016). Here, the decomposition leverages the underlying Reo (Razavi and Sirjani 2006) process algebraic semantics (Kokash et al. 2010). With Reo, communication is untimed in the Focus (Broy and Stølen 2001; Broy 2010) sense and the decomposition follows process actions instead of shared channels. How the parallel decomposition of Reo processes can be translated to untimed Focus systems is subject to ongoing research.

7.2 Agile architecting

Industry and research have produced over 120 ADLs (Malavolta et al. 2013). Most of these feature the composition of components into larger architectures and some of these also feature the denotational semantics necessary to support agile architecting through automated decomposition. This section discusses related ADLs and their support for automated decomposition.

AutoFocus 3 (Hölzl and Feilkas 2010) and MontiArc (Butting et al. 2017a) are ADLs featuring tool chains for developing architectures of reactive software systems that are grounded in Focus (Broy and Stølen 2001). This paper's system model describes the formal foundations of both ADLs. AutoFocus 3 supports model checking the behavior of architectures against LTL and CTL properties (Campetelli et al. 2011). MontiArc supports semantic differencing of components (Butting et al. 2017b). However, both currently lack fully automated component decomposition methods. Hence, even if employed in agile processes, the challenge of manually decomposing monolithic architectures remains. Our approach can directly be integrated into the tool chains of both ADLs.

The π-ADL supports model checking for verifying software architectures against DynBLTL properties (Cavalcante et al. 2016). Therefore, a statistical model of finite system executions is created and the probability of satisfying a property within confidential bounds is calculated. However, we are unaware of any agile architecting methods based on the π-calculus. As our approach is based on Focus and not on the π-calculus, it cannot be directly integrated into the tool chains of ADLs that are based on the π-calculus. Developing an influence relation and a decomposition procedure for systems based on the π-calculus is interesting future work.

7.3 Applicability to other automata models

Other automata models, such as I/O automata (Lynch and Tuttle 1989), Interface automata (de Alfaro and Henzinger 2005), team automata (ter Beek et al. 2003), and component-interaction automata (Brim et al. 2006), do not include the notion of channel. Instead, they distinguish between input, internal, and output actions. Composition operators compose different automata according to their actions. As these automata models do not explicitly incorporate the notion of channel, it is not possible to define an influence relation between the channels of these automata. However, it could be interesting to define a notion of influence between input and output actions of the automata. The relation could be defined such that it identifies whether the receipt of a specific input action influences the output of a specific output action. Transferring this idea to the automata model used in this paper, the above corresponds to the question whether a specific message on a specific input channel influences the output of a specific message on a specific output channel. We consider the definition of such a relation and the development of automated tool support as interesting future work. This would enable a more fine-grained analysis as presented in this paper. Whether this analysis or the analysis presented in this paper is more appropriate depends on the use case and intention by the developer.

For other automata models that include the notion of channel, such as port automata (Grosu and Rumpe 1995), time-synchronous channel automata (Butting et al. 2019), and MAA_{ts} automata (Ringert 2014), it is possible to transfer the notion of influence between channels. However, some of these automata models use a different semantics as the automaton model used in this paper. The method for detecting whether one channel influences another channel needs to be adjusted depending on the semantics of the respective automaton model. Consequently, the decomposition method also needs to be adjusted depending on the composition operator of the respective automaton model.

8 Summary

We have presented a method to automatically decompose a monolithic deterministic component into an architecture consisting of multiple subcomponents that are composed in parallel. This supports agile architecting by reducing the effort for analyzing and implementing system behavior along subcomponents and facilitates refinement and refactoring of architectures. To this end, we have conceived a notion of influence between channels and formalized it in the Focus (Broy and Stølen 2001) theory. We have proven that this decomposition is an actual refactoring, i.e., the resulting systems are semantically equivalent to the original systems. Hence, this decomposition can be applied to stepwise refinement and ultimately facilitates architecture modeling.

Acknowledgements This research has partly received funding from the German Federal Ministry for Education and Research under Grant No. 01IS16043P. The responsibility for the content of this publication is with the authors.

References

Béal, M., Carton, O.: Determinization of transducers over infinite words. In: ICALP, Springer, Lecture Notes in Computer Science, vol. 1853, pp. 561–570 (2000)

Béal, M.P., Carton, O.: Determinization of transducers over finite and infinite words. Theor. Comput. Sci. **289**(1), 225–251 (2002)

Brim, L., Černá, I., Vařeková, P., Zimmerova, B.: Component-interaction automata as a verification-oriented component-based system specification. SIGSOFT Softw. Eng. Notes **31**, 4-es (2006)

Broy, M.: A logical basis for component-oriented software and systems engineering. Comput. J. **53**(10), 1758–1782 (2010)

Broy, M., Stølen, K.: Specification and Development of Interactive Systems: Focus on Streams, Interfaces and Refinement. Springer, Heidelberg (2001)

Büchi, J.R.: On a decision method in restricted second order arithmetic. In: International Congress on Logic, Methodology and Philosophy of Science, pp. 1–11 (1962)

Butting, A., Haber, A., Hermerschmidt, L., Kautz, O., Rumpe, B., Wortmann, A.: Systematic language extension mechanisms for the MontiArc Architecture Description Language. In: Modelling Foundations and Applications (ECMFA'17), Held as Part of STAF 2017, Springer International Publishing, pp. 53–70 (2017)

Butting, A., Kautz, O., Rumpe, B., Wortmann, A.: Semantic differencing for message-driven component & connector architectures. In: International Conference on Software Architecture (ICSA'17), IEEE, pp. 145–154 (2017)

Butting, A., Kautz, O., Rumpe, B., Wortmann, A.: Continuously analyzing finite, message-driven, time-synchronous component & connector systems during architecture evolution. J. Syst. Softw. **149**, 437–461 (2019)

Campetelli, A., Hölzl, F., Neubeck, P.: User-friendly model checking integration in model-based development. In: International Conference on Computer Applications in Industry and Engineering (2011)

Carlsson, G., Yu, J.: A prime decomposition of probabilistic automata (2015). arXiv preprint arXiv:150301502

Cavalcante, E., Quilbeuf, J., Traonouez, L.M., Oquendo, F., Batista, T., Legay, A.: Statistical model checking of dynamic software architectures. In: European Conference on Software Architecture (2016)

de Alfaro, L., Henzinger, T.A.: Interface-based design. In: Engineering Theories of Software Intensive Systems (2005)

Debruyne, V., Simonot-Lion, F., Trinquet, Y.: EAST-ADL—an architecture description language. In: Architecture Description Languages, pp. 181–195. Springer (2005)

Díaz, J., Pérez, J., Garbajosa, J., Yagüe, A.: Change-impact driven agile architecting. In: 2013 46th Hawaii International Conference on System Sciences, pp. 4780–4789 (2013)

Díaz, J., Pérez, J., Garbajosa, J.: Agile product-line architecting in practice: a case study in smart grids. Inf. Softw. Technol. **56**(7), 727–748 (2014)

Farwer, B.: ω-Automata. In: Grädel, E., Thomas, W., Wilke, T. (eds.) Automata Logics and Infinite Games: A Guide to Current Research, pp. 3–21. Springer, Berlin (2002)

Feiler, P.H., Gluch, D.P.: Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language. Addison-Wesley, Boston, MA (2012)

France, R., Rumpe, B.: Model-driven development of complex software: a research roadmap. In: Future of Software Engineering 2007 at ICSE (2007)

Friedenthal, S., Moore, A., Steiner, R.: A Practical Guide to SysML: The Systems Modeling Language. The MK/OMG Press. Elsevier Science, Amsterdam (2011)

Gerace, G., Gestri, G.: Decomposition of synchronous sequential machines into synchronous and asynchronous submachines. Inf. Control **11**(5), 568–591 (1967)

Grosu, R., Rumpe, B.: Concurrent timed port automata. Technical Report TUM-I9533, TU Munich (1995)

Hoare, C.A.R.: Communicating sequential processes. Commun. ACM **21**(8), 666–677 (1978)

Hölzl, F., Feilkas, M.: AutoFocus 3—a scientific tool prototype for model-based development of component-based, reactive, distributed systems. In: Giese, H., Karsai, G., Lee, E., Rumpe, B., Schätz, B. (eds.) Model-Based Engineering of Embedded Real-Time Systems. Springer, Berlin (2010)

Jongmans, S.S., Clarke, D., Proença, J.: A procedure for splitting data-aware processes and its application to coordination. Sci. Comput. Program. **115**, 47–78 (2016)

Koehler, C., Clarke, D.: Decomposing port automata. In: Proceedings of the 2009 ACM symposium on Applied Computing, pp. 1369–1373. ACM (2009)

Kokash, N., Krause, C., de Vink, E.P.: Data-aware design and verification of service compositions with Reo and mCRL2. In: Proceedings of the 2010 ACM Symposium on Applied Computing. ACM (2010)

Lynch, N.A., Tuttle, M.R.: An introduction to input/output automata. CWI Q. **2**(3), 219–246 (1989)

Malavolta, I., Lago, P., Muccini, H., Pelliccione, P., Tang, A.: What industry needs from architectural languages: a survey. IEEE Trans. Softw. Eng. **39**, 869–891 (2013)

Medvidovic, N., Taylor, R.N.: A classification and comparison framework for software architecture description languages. IEEE Trans. Softw. Eng. **26**, 70–93 (2000)

Milner, R.: Communicating and Mobile Systems: The π-Calculus. Cambridge University Press, New York, NY (1999)

Naur, P., Randell, B. (eds.): Software engineering: report of a conference sponsored by the NATO Science Committee, Garmisch, Germany, 7–11 Oct 1968, Brussels, Scientific Affairs Division, NATO (1969)

Nozaki, A.: Practical decomposition of automata. Inf. Control **36**(3), 275–291 (1978)

Plotkin, B., Plotkin, T.: Decompositions and complexity of linear automata (2015). arXiv preprint arXiv:150606017

Poort, E.R.: Driving agile architecting with cost and risk. IEEE Softw. **31**(5), 20–23 (2014)

Razavi, N., Sirjani, M.: Using Reo for formal specification and verification of system designs. In: Proceedings of the Fourth ACM and IEEE International Conference on Formal Methods and Models for Co-design, 2006. MEMOCODE'06. Proceedings. IEEE Computer Society, pp. 113–122 (2006)

Ringert, J.O.: Analysis and Synthesis of Interactive Component and Connector Systems. Aachener Informatik-Berichte, Software Engineering, Band 19. Shaker Verlag, Herzogenrath (2014)

Ringert, J.O., Rumpe, B.: A little synopsis on streams, stream processing functions, and state-based stream processing. Int. J. Softw. Inform. **5**(1–2), 29–53 (2011)

Ringert, J.O., Rumpe, B., Wortmann, A.: Model-based specification of component behavior with controlled underspecification. In: Modellbasierte Entwicklung eingebetteter Systeme (MBEES'16) (2016)

Safra, S.: On the complexity of σ-automata. In: Proceedings of the 29th Annual Symposium on Foundations of Computer Science, pp. 319–327. IEEE Computer Society (1988)

Schlegel, C., Steck, A., Lotz, A.: Model-driven software development in robotics: communication patterns as key for a robotics component model. In: Chugo, D., Yokota, S. (eds.) Introduction to Modern Robotics. iConcept Press, Hong Kong (2011)

Strobl, F., Wisspeintner, A., Marz, A.: Specification of an elevator control system. Technical report, TU Munich (1999)

ter Beek, M.H., Ellis, C.A., Kleijn, J., Rozenberg, G.: Synchronizations in team automata for groupware systems. Comput. Support. Coop. Work (CSCW) **12**, 21–69 (2003)

Uygur, G., Sattler, S.M.: Parallel decomposition for safety-critical systems. In: 2013 3rd International Electric Drives Production Conference (EDPC), pp. 1–8 (2013)

Van Ommering, R., Van Der Linden, F., Kramer, J., Magee, J.: The Koala component model for consumer electronics software. Computer **33**(3), 78–85 (2000)

Völter, M., Stahl, T., Bettin, J., Haase, A., Helsen, S., Czarnecki, K.: Model-Driven Software Development: Technology, Engineering, Management. Wiley Software Patterns Series. Wiley, Hoboken (2013)

Weber, A.: Transforming a single-valued transducer into a mealy machine. J. Comput. Syst. Sci. **56**(1), 46–59 (1998)

Weber, A., Klemm, R.: Economy of description for single-valued transducers. Inf. Comput. **118**(2), 327–340 (1995)