[BKRW18a] A. Butting, O. Kautz, B. Rumpe, A. Wortmann:
Continuously Analyzing Finite, Message-Driven, Time-Synchronous Component & Connector Systems During Architecture Evolution.
In: Journal of Systems and Software, 2018.
www.se-rwth.de/publications/

Continuously Analyzing Finite, Message-Driven, Time-Synchronous Component & Connector Systems During Architecture Evolution

Arvid Butting[a], Oliver Kautz[a,*], Bernhard Rumpe[a], Andreas Wortmann[a]

[a]*Software Engineering, RWTH Aachen University, Aachen, Germany, www.se-rwth.de*

Abstract

Understanding the semantic differences of continuously evolving system architectures by semantic analyses facilitates engineers during evolution analysis in understanding the impact of the syntactical changes between two architecture versions. To enable effective semantic differencing usable in practice, this requires means to fully automatically check whether one version of a system admits behaviors that are not possible in another version. Previous work produced very general system models for message-driven time-synchronous (MDTS) systems that impede fully automated semantic differencing but very adequately describe such systems from a black-box viewpoint abstracting from hidden internal component behavior. This paper presents a system model for MDTS systems from a white-box viewpoint (assuming component implementation availability) and presents a sound and complete method for semantic differencing of finite MDTS system architectures. This method relies on representing (sub-)architectures as channel automata and a reduction from the semantic differencing problem for such automata to the language inclusion problem for Büchi automata. The system model perfectly captures the logical basics of MDTS systems from a white-box viewpoint and the method enables to fully automatically calculate semantic differences between two finite MDTS systems on push-button basis, yields witnesses, and ultimately facilitates semantic evolution analysis of such systems.

Keywords: Component Software Engineering, Semantics, Automata, Refinement, Semantic Differencing, Evolution Analysis

1. Introduction

Component-based software engineering [30] promises improving software development through reuse of independently developed and validated off-the-shelf building blocks with stable interfaces. These building blocks usually are implemented in general-purpose programming languages (GPLs), Hence, they are subject to the conceptual gap between the problem domains and solution domains of discourse, which arises from addressing problem domain challenges with programming language complexities [14].

Model-driven development (MDD) [44] aims at reducing this gap by lifting domain-specific, abstract, models to primary development artifacts. Such models can leverage domain-specific vocabulary to be better comprehensible as well as more abstract and hence are better suited towards analysis and transformation than GPL programs. Software engineering also applies MDD to itself to facilitate addressing its challenges. Consequently, modeling languages for various challenges in software engineering, such as database manipulation languages, build process description languages, and architecture description languages have been developed.

Architecture description languages (ADLs) [29] leverage the potential of model-driven development [44] for the description of software architectures. In many domains, knowing the precise semantics of models is crucial due to safety concerns, but current architecture modeling processes, such as MDA [31] do not take these into account. Stepwise refinement [5, 6] is 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. To this effect, checking whether successor component versions *refine* their predecessors in terms of observable input/output behavior is crucial.

Similar to UML [32], the specific semantics of many ADL details are encoded in their infrastructures and tools only. Where fully detailed denotational or operational semantics are available, such as with Focus [7], these are usually too complex for fully automated refinement checking and typically require to (partially) manually prove refinement between two component versions. This impedes stepwise refinement so severely that it becomes a "highly idealistic" [5] idea. However, enabling automatic stepwise refinement for software architecture models would greatly facilitate development in domains where component adherence to certain properties is crucial. With automated methods, manual proofs become redundant. This enables users who are no experts in formal methods to prove or disproof refinement between architecture versions. As programmers are rarely experts in formal methods, this opens the possibility to apply stepwise refinement methodologies to a broader user range. In case an architecture is no refinement of another,

*Corresponding author
Email addresses: butting@se-rwth.de (Arvid Butting), kautz@se-rwth.de (Oliver Kautz), rumpe@se-rwth.de (Bernhard Rumpe), wortmann@se-rwth.de (Andreas Wortmann)

the method presented in this paper fully automatically calculates a behavior that is possible in the one architecture but not in the other. This behavior serves as witness and is a concrete disproof for refinement. Software engineers can use the witness as evidence for efficiently identifying the syntactic elements in the architecture's implementation that cause non-refinement.

In [9], we identified a subset of the Focus [7] semantics for time-synchronous, distributed, interactive systems that is powerful enough to model complex and realistic systems and is adaptable to enable fully automated refinement checking between components. Based on this, [9] describes an approach to transform software component models into a variant of port automata [16], compose these syntactically, and translate the results into Büchi automata, where their refinement can be checked through language inclusion [23]. This approach is realized with the MontiArcAutomaton component & connector ADL [35, 37] and the RABIT [2, 3] tool for fully automated language inclusion checking between Büchi automata. It enables modeling software architectures with powerful ADLs and checking refinement on a push-button basis. To this effect, the contributions of [9] are:

- A formulation of the semantics domain of time-synchronous [7] stream processing functions (TSSPFs) inspired by the notion of stream processing function [36].
- A variant of port automata: time-synchronous port automata (TSPA) [16] with operational semantics based on execution traces and denotational semantics based on sets of TSSPFs.
- A semantically compositional syntactic composition operator for TSPAs: The semantics of the syntactic composition of two TSPAs is equal to the composition of the semantics of the individual TSPAs.
- A transformation from finite TSPAs to Büchi automata.
- A proof showing the operational semantics of a finite TSPA and the language accepted by the Büchi automaton resulting from such a transformation coincide.
- The result that refinement checking and disproof generation in form of semantic difference witnesses for software architectures where components can be mapped to finite TSPAs can be reduced to language inclusion checking and counterexample generation for Büchi automata.
- An implementation based on the MontiArcAutomaton component & connector ADL [35, 37] and RABIT [2, 3].

In this paper, we enhance and extend the previous approach to achieve practical efficiency improvements and technical enhancements of the underlying formal system model. To this effect, this paper's additional contributions are:

- Time-synchronous channel automata (TSCAs): an improved variant of TSPAs that enables defining an associative and commutative syntactic composition operator, while retaining previous results regarding the relation between the system models and compositionality.
- The previous composition operator for TSPAs (cf. [9]) is neither associative nor commutative. Using the commutativity and associativity of the TSCA composition operator enables to define an intuitive notion of system architecture, which is not possible with the TSPA composition operator.
- A method for trimming finite TSCAs to reduce complexity of analyses.
- A method for composing finite TSCAs such that the compound does not contain any unproductive states to mitigate state explosions.
- The identification of a subclass of finite non-deterministic TSCAs, which is a proper superset of deterministic TSPAs, where semantic differencing is possible in polynomial time.
- The insight that the Büchi automata resulting from transforming TSCAs are always "weak" and therefore enable the application of efficient algorithms enabling, for instance, easy complementation or minimization.
- A notion of system architecture based on a white-box viewpoint on message-driven time-synchronous (MDTS) systems and the previously developed theory. The associativity and commutativity of the composition operator for TSCAs is important for the notion of system architecture to be well defined. The system architecture definition as introduced in this paper is not possible with TSPAs as introduced in [9] because TSPAs do not have a commutative and associative composition operator.
- A method for mitigating the state explosion problem during semantic differencing of finite system architectures, which is especially useful during continuous architecting when it comes to understanding the semantic differences between two successor versions. The method not only relies on trimming but also on iteratively applying refinement checking to smaller sub-architectures.
- An extended evaluation including an additional example and an improved composition method that combines composition with trimming.

This paper further contains many additional examples that increase comprehensibility and illustrate this paper's approach. The resulting fully automatic analysis technique for comparisons of TSCAs greatly supports continuously evolving projects where the overall architecture changes frequently. It also greatly facilitates analyzing the semantic differences between products of a product line architecture where the individual products are syntactically only slightly different.

1.1. Paper Structure and Overview

Section 2 sketches the idea of stepwise refinement. To this effect, it presents two architecture models, the elevator control system presented and evaluated in [9] as well as a more compact architecture serving as running example throughout this paper.

Subsequently, Section 3 presents the Focus subset used as semantics domain from a black-box viewpoint (as functions). This paper's approach is applicable to finite systems where it is possible to describe the system's semantics with the system model described in this section. It is argued that the system model is adequate for describing architectures while abstracting from hidden internal details, but hiding internal details hampers automated analyses.

This motivates Section 4, which describes a new system model that represents components from a white-box perspective (as automata). The automata model is compatible to the function model of the previous section and explicitly captures internal component details.

Afterwards, Section 5 presents automated semantic differencing based on the latter system model (automata). We obtain a more efficient semantic differencing method as described in previous work. The compatibility of the system models ensures the results equally apply to both semantic domains. However, this paper's approach is only applicable if component implementations are available and can be transformed to the automata introduced in Section 4.

Section 6 presents the implementation of our approach with MontiArcAutomaton and RABIT and evaluates its applicability. Section 7 discusses observations and Section 8 highlights related work before Section 9 concludes. The appendix describes examples used throughout the paper in more detail.

2. Examples

This section presents two example architectures for stepwise refinement. The first example illustrates the benefits of our approach on an elevator control system (Section 2.1) as presented in [9]. The second example describes a distributed Modulo-8-Counter (Section 2.2), which is used as running example throughout the remainder of this paper. While the former is suited to comprehending the benefits of stepwise refinement intuitively, the latter is compact enough to be discussed in details in the remainder.

2.1. An Elevator Control System

Consider the model-driven development of an elevator control system (ECS) as presented in [42]. The `ECS` depicted in Figure 1 comprises two hierarchically composed components representing the three floors the elevator serves (component `Floors`) and the elevator cabin (component `Elevator`). Whenever a button on a floor (indicated, for example, by a message on the incoming port `btn1`) is pressed, the `ECS` should activate the light (by sending a message via outgoing port `led1`) on the corresponding floor and instruct the elevator cabin to visit that floor. The control logic of the elevator is modeled via a statechart variant embedded into the `Elevator`'s subcomponent `Control`. This component receives messages upon arriving at a specific floor (*e.g.,* via incoming port `at1`) and sends messages to `Door` and `Motor` to operate its door and to move between the floors. The latter two embed models of compact action languages to describe their respective behavior.

For this version of the `ECS`, the software architects have proven that certain properties hold (*e.g.,* that it cannot produce blocking situations). Now they aim to replace the `Elevator` component with a smarter version that reacts only to elevator requests on a floor if there is no such request yet. To this effect, the company employs stepwise refinement to avoid proving the properties of `Elevator` again for its successor version `SmartElevator`. Therefore, the behavior descriptions of all subcomponents are translated into TSCAs. For composed components, the behavior descriptions of their subcomponents are translated also and merged iteratively. This ultimately eliminates all hierarchy levels but the last. The result of this transformation is depicted in Figure 2, where the behavior descriptions of all three subcomponents have been transformed accordingly and merged into a single TSCA. The same is performed for the improved `SmartElevator` component before both are transformed into weak non-deterministic Büchi automata as presented in Section 6.

Using this transformation reduces semantic component refinement to language inclusion on Büchi automata and can be solved automatically, for instance, by using the tool RABIT. Hence, with this infrastructure in place, the company now can fully automated ensure whether the `SmartElevator`, and its potential successors, actually refine their predecessors or require further adjustment. Where refinement is refuted, difference witnessing input/output pairs are produced. This automation of stepwise refinement can increase the pace of each refinement step and, hence, overall development efficiency.

2.2. A Modulo-8 Counter

This example presents a modulo-8 counter inspired by the model presented in [15] as demonstration of stepwise refinement along the depth of composition layers. The modulo-8 counter outputs the binary representation of a number n between 0 and 7, which can be incremented ($(n + 1)$ % 8) or reseted ($n = 0$). The initial value of n is 0. The modulo-8 counter is modeled as the MontiArcAutomaton component `Mod8Counter` depicted in Figure 3 (a). The component has two incoming ports and three outgoing ports of the data type Boolean. In the initial definition, only the behavior of the outermost component `Mod8Counter` is specified. The valuations of the outgoing ports $\mathtt{x}_2$, $\mathtt{x}_1$, and $\mathtt{x}_0$ are equal to the Boolean representations of the variables in the binary representation of n (*i.e.,* $n = x_2 \cdot 2^2 + x_1 \cdot 2^1 + x_0 \cdot 2^0$). Upon receiving *true* via the incoming port `inc`, the value of n is increased if the value on port `res` is not equal to *true*, and on receipt of *true* via the port `res`, the value of n is set to 0, regardless of the value received on port `inc`.

To decouple parts of the functionality of the modulo-8 counter, *e.g.,* for individual testing, the behavior of the `Mod8Counter` is structurally refined by introducing the two subcomponents `Controller` and `Counter`, as depicted in Figure 3 (b). The controller component is responsible to delegate a reset of the counted value to the counter. This reset is triggered either after receiving a message *true* on its incoming port `rIn` or if the current counted value is 7 and the value should be further increased. The counter component realizes the counting

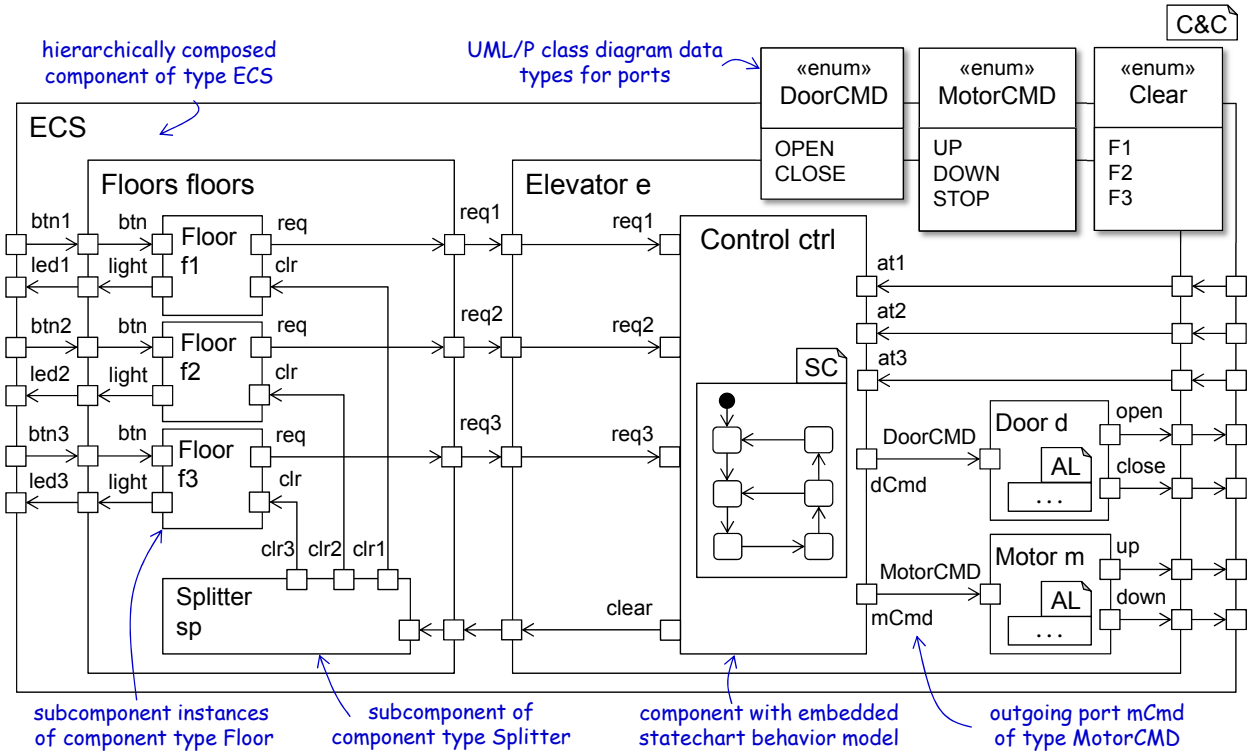

Figure 1: The elevator control system ECS comprises subcomponents to manage serving elevation requests on up to three floors.

functionality, but is unable to reset a counted value from 7 to 0 after increasing. Using the method for refinement checking presented in this paper, it is possible to fully automatically check whether the original version (atomic Mod8Counter) is equivalent to its successor version (composed Mod8Counter).

Later, the behavior of the counter is refined in a further structural refinement step (*cf.* Figure 3 (c)) by introducing subcomponents to the component Counter. The company reuses these subcomponents from a different project. The behavior of the component Counter is then defined by three counter bit components pos0, pos1, and pos2, which all have the same component behavior - denoted in MontiArcAutomaton by the fact that they are of the same component type CBC. Each of these can count a single bit component only. The MontiArcAutomaton component CBC with an embedded automaton realizing the component behavior is depicted in Figure 4. The bit value can be increased (modulo 2) via a message *true* on the incoming port i and reseted to *false* via a message *true* on the incoming port r. The current value of the bit is output via the outgoing port v, and the value of q is *true* iff, after increasing, the bit value changes from *true* to *false*. Otherwise, it emits *false*. Using our method, checking whether the new architecture is semantically equivalent to any of the other two architectures is possible within milliseconds.

At this point, another modeling expert notices that the design of the mod-8 counter is too complex and can be simplified, as the behavior of each CBC components already realizes the overflow of the modulo. Therefore, the expert proposes to model the behavior as depicted in Figure 5. As it is not obvious if the behaviors of Figure 3 (c) and Figure 5 are equivalent, the refinement check presented in this paper is employed and yields sound and complete results within milliseconds.

3. A Semantics Domain for Components

This section introduces the semantics domain for components based on the Focus framework [5, 7, 16, 36, 39] and recapitulates the most important results from [9, 16], which underlie the approach presented in this paper.

We interpret software architectures as networks of autonomously acting components communicating in a time-synchronous manner via directed, typed channels connecting the components' interfaces. A time-synchronous architecture can be interpreted as a system where component computations are performed concurrently and controlled by a global clock that splits runtime into discrete and equidistant time units. In every time unit, each component receives finitely many input messages via its interfaces and outputs finitely many messages to its environment. The computations of each component in every time unit must terminate. To this end, components partition time slices into sequences of operations (*e.g.,* assessing the guard of an embedded automaton's transition or assigning values according to its actions). Although these sequences of operations are untimed in the Focus sense, they are causally related. The semantics of component behavior thus happens logically in superdense time [28], which, following [26], distinguishes between the discrete "time continuum" (global Focus time) and "untimed causally-related actions" (a component behavior's actions within the component's time slice).

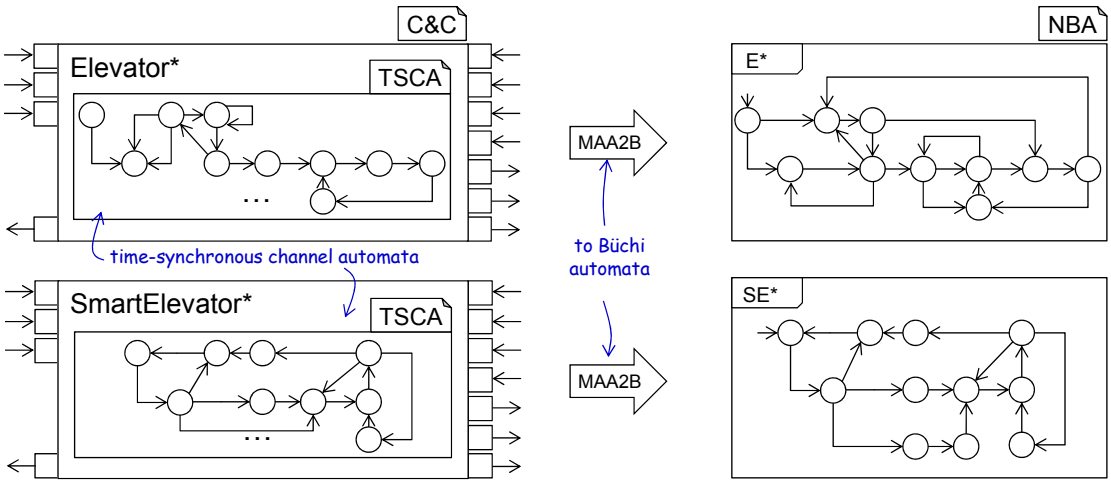

Figure 2: The composed components `Elevator` and `SmartElevator` each are transformed into flat components with a single port automaton prior to being transformed into Büchi automata and checked for language inclusion.

In the remainder, 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 $Z \subseteq X$, the restriction of f to Z is the function $f|_Z \in [Z \to Y]$ that satisfies $f|_Z(x) = f(x)$ for all $x \in Z$. Given two functions $f \in [X \to A]$ and $g \in [Y \to B]$, the overriding union of f with g is the function $f + g \in [(X \cup Y) \to (A \cup B)]$ that satisfies $(f + g)(x) = g(x)$ if $x \in Y$ and $(f + g)(x) = f(x)$ if $x \in X \setminus Y$ for all $x \in X \cup Y$.

3.1. Streams, Messages, Types, and Communication Histories

The history of messages a component receives or sends via an interface (*e.g.,* channel) is formally described as a stream that contains messages in order of their transmission. Let M be an arbitrary alphabet. A stream over the set M is a finite or infinite sequence of elements from M. Following [7, 39], we denote by

- M^* the set of all finite streams over M,
- M^∞ the set of all infinite streams over M,
- $\langle\rangle$ the empty stream, which is an element of M^*,
- $s^\frown t$ the concatenation of two streams s and t such that $((M^* \cup M^\infty), ^\frown, \langle\rangle)$ is a monoid. If $s \in M^\infty$ then $s^\frown t = s$.
- $\sqsubseteq$ the prefix relation over streams, which is a partial order defined by: $\forall s, t \in (M^* \cup M^\infty) : s \sqsubseteq t \Leftrightarrow \exists u : s^\frown u = t$,
- $s.t$ the $(t + 1)$-st element of a stream $s \in (M^* \cup M^\infty)$,
- $s{\downarrow_t}$ the prefix of a stream $s \in M^\infty$ of length $t \in \mathbb{N}$.

Example 1. *The finite sequence $fib_7 = 0, 1, 1, 2, 3, 5, 8 \in \mathbb{N}^*$ is a finite stream of natural numbers. It contains the first seven Fibonacci numbers. The infinite stream of all Fibonacci numbers $fib \in \mathbb{N}^\infty$ is defined by $fib.0 = 0 \wedge fib.1 = 1 \wedge \forall t \in \mathbb{N} : t \geq 2 \Rightarrow fib.t = fib.(t-2) + fib.(t-1)$. By definition, we have $fib^\frown fib_7 = fib$. Further, $fib_7 \sqsubseteq fib$ because the prefix of length 7 of fib is equal to fib_7, i.e., $fib{\downarrow_7} = fib_7$. Thus, the first seven elements of fib_7 and fib are equal, e.g., $fib_7.0 = fib.0 = 0$ and $fib_7.3 = fib.3 = 2$.*

In the remainder, let M denote an arbitrary but fixed set of data elements, such as messages, and let $Type$ be a set of data types such that each $t \in Type$ satisfies $t \subseteq M$. Types facilitate restricting the set of messages a component may emit or receive via an interface. We assume a discrete model of time where component computation is divided into discrete time units of equal and finite duration. In each time unit each component receives at most one message via each incoming interface, may perform finitely many state changes and emits at most one message via each outgoing interface. We use the special symbol $\varepsilon \in M$ to denote the absence of a message during a time unit and require $\varepsilon \in t$ for each $t \in Type$.

A *channel* is an identifier for a communication link between interface elements of components. In the following, we denote by C a set of typed channel names. The function $type \in [C \to Type]$ maps each channel in the set C to its type. Let $B \subseteq C$ be an arbitrary set of channel names. We model the history of messages emitted via the channels in the set B as a *communication history* $h \in B^\Omega$, which is an element of the set B^Ω defined as follows: $B^\Omega \stackrel{\text{def}}{=} \{h \in [B \to M^\infty] \mid \forall b \in B : h(b) \in type(b)^\infty\}$. Let $h \in B^\Omega$ be a communication history, $H \subseteq B^\Omega$ a set of communication histories, and $t \in \mathbb{N}$ a natural number. We lift the operator $\downarrow$ to communication histories and sets of communication histories in a point-wise manner, *i.e.,* $b{\downarrow_t} \in [B \to M^*]$ denotes the function that satisfies $b{\downarrow_t}(i) = b(i){\downarrow_t}$ for all $i \in B$ and $H{\downarrow_t} \stackrel{\text{def}}{=} \bigcup_{h \in H} h{\downarrow_t}$ denotes the set resulting from applying the operator to each element in H.

Example 2. *Let $c \in C$ be a channel of natural numbers. Then, in each time unit, the channel c can be either assigned a natural number or the empty message. Thus, $type(c) = \mathbb{N} \cup \{\varepsilon\} \in Type \subseteq M$. The communication history that assigns the channel c the sequence of Fibonacci numbers is given by $h = \{c \mapsto fib\} \in c^\Omega$ where fib is defined as in Example 1. The stream containing all negative integers neg defined by $\forall t \in \mathbb{N} : neg.t = -t$ is no valid assignment to channel c because there exists a time unit $t \in \mathbb{N}$ such that $neg.t \notin type(c) = \mathbb{N} \cup \{\varepsilon\}$, e.g., we have $neg{\downarrow_2} = -1, -2$.*

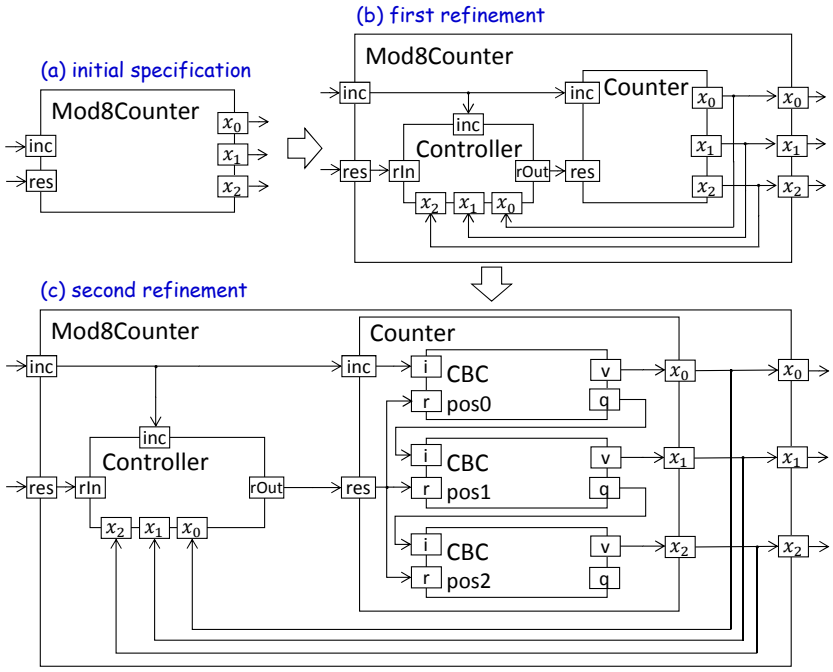

Figure 3: Graphical representation of the component `Mod8Counter` in MontiArcAutomaton syntax (a) in its initial specification and (b) after a first and (c) a second structural refinement step. All ports are of data type Boolean.

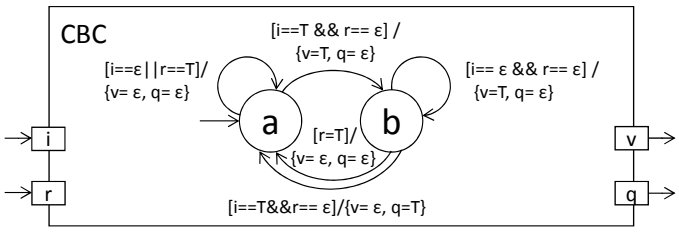

Figure 4: Automaton model realizing the component behavior of `CBC`.

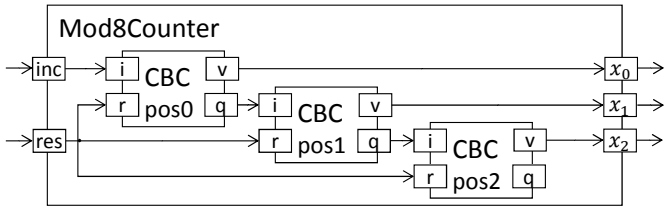

Figure 5: Alternative model of the mod-8 counter, with the behavioral equivalence to the model in Figure 3 (c) in question.

Thus, $\{c \mapsto neg\} \notin a^{\Omega}$ *is no communication history. The function mapping the channel c to its first* 7 *elements is given by* $h{\downarrow_7} = \{c \mapsto fib_7\}$ *where* fib_7 *is defined as in Example 1. Let* $empty \in \{c\}^{\Omega}$ *be defined by* $\forall t \in \mathbb{N} : empty(c).t = \varepsilon$ *denote the communication history that always assigns the channel c to the empty message. Then,* $\{h, empty\}{\downarrow_7} = \{h{\downarrow_7}, empty{\downarrow_7}\} = \{\{c \mapsto fib_7\}, \{c \mapsto \varepsilon, \varepsilon, \varepsilon, \varepsilon, \varepsilon, \varepsilon, \varepsilon\}\}$.

3.2. Time-Synchronous Stream Processing Functions

We model the semantics of distributed interactive systems as sets of time-synchronous stream processing functions (TSSPFs) [9]. The notion of TSSPFs is inspired by the notion of timed SPFs [7, 16, 36, 39]. The major and crucial difference between the two notions is that TSSPFs process exactly one message per channel per time unit, whereas SPFs process a stream of messages per channel per time unit. The key idea is to treat components as black-boxes having an observable behavior characterized by the interactions on channels between systems and subsystems while hiding internal implementation details. A component is mapped to a set of functions describing the component's possible behaviors. Such a function maps communication histories over the set of input channels of a component to communication histories over the set of the component's output channels. Thus, each function in the semantics of a component with input channels $I \subseteq C$ and output channels $O \subseteq C$ is of the form $f \in [I^{\Omega} \to O^{\Omega}]$. However, such functions are not always realizable in the sense that they can be implemented [7, 34]. Intuitively, the characterizing properties for realizability are captured by the notion of weak-causality: a component cannot change messages it received or sent in the past and cannot react to messages it receives in the future [7, 34, 36, 39]. Thus, the output of a behavior describing function until time t must be completely determined by its input until time t:

Definition 1 (Time-Synchronous Stream Processing Function)**.** *Let* $I, O \subseteq C$ *be two disjoint sets of input and output channels. A function* $f \in [I^{\Omega} \to O^{\Omega}]$ *is called (weakly causal) time-synchronous stream processing function iff*

$\forall i, i' \in I^{\Omega} : \forall t \in \mathbb{N} : i{\downarrow_t} = i'{\downarrow_t} \Rightarrow f(i){\downarrow_t} = f(i'){\downarrow_t}$.

We denote by $[I^{\Omega} \xrightarrow{wc} O^{\Omega}]$ the set of all (weakly causal) TSSPFs mapping input histories in I^{Ω} to output histories in O^{Ω}.

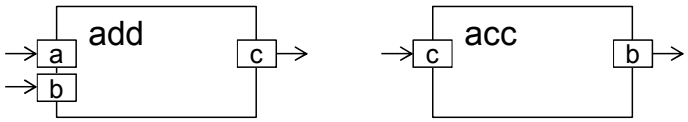

Figure 6: Graphical representation of the TSSPFs *add* and *acc*.

Example 3. *This example defines the stream processing function add that specifies the behavior of a component for adding natural numbers. The interface of the TSSPF is graphically illustrated on the left hand side of Figure 6. The input channels are* $I = \{a, b\}$ *and the set of output channels is* $O = \{c\}$*. The type of all channels is the type of natural numbers, i.e.,* $type(a) = type(b) = type(c) = \mathbb{N} \cup \{\varepsilon\} \in Type \subseteq M$*. If the function add receives natural numbers on both channels a and b in a time unit t, then the function outputs the sum of the received messages via the channel c in time unit t. Otherwise, if the function receives the empty message* ε *on any of the input channels in time unit t, then the function outputs* ε *in time unit t. The function* $add \in [I^\Omega \to O^\Omega]$ *is formally defined by* $\forall i \in I^\Omega : \forall t \in \mathbb{N} : (add(i))(c).t = . \begin{cases} i(a).t + i(b).t, & \text{if } i(a).t, i(b).t \in \mathbb{N} \\ \varepsilon, & \text{otherwise} \end{cases}$

The function add is weakly causal because its output in each time unit is fully specified by its inputs in the same time unit, i.e., in each time unit, the function's output does not depend on future input and the function does not change previously processed messages. This is verifiable with a short proof by induction over the lengths of prefixes of communication histories.

The following example illustrates that the weak causality requirement on TSSPFs is necessary.

Example 4. *This example defines the function u (unrealizable) over communication histories that is not weakly causal. We define the function over Boolean messages. The function's input channel set is given by* $I = \{in\}$ *and its output channel set is given by* $O = \{out\}$*. The types of in and out are* $type(in) = type(out) = \{\top, \bot, \varepsilon\} \in Type \subseteq M$ *where* $\top$ *represents the value true and* $\bot$ *represents the value false. In each time unit t, the function* $u \in [I^\Omega \to O^\Omega]$ *outputs the value it receives in time unit* $t + 1$*. It is formally defined by* $\forall i \in I^\Omega : \forall t \in \mathbb{N} : u(out).t = i(in).(t + 1)$*. Obviously, the functionality described by the function u cannot be implemented by a component: A component implementing the function would be able to predict its future input to determine its present output. This is formally captured by weak-causality. To proof that the function u is not weakly causal, we need to find two input channel histories* $i, i' \in I^\Omega$ *and a time unit* $t \in \mathbb{N}$ *such that* $i{\downarrow_t} = i'{\downarrow_t} \wedge u(i){\downarrow_t} \neq u(i'){\downarrow_t}$*. We choose i and i′ such that* $\forall t \in \mathbb{N} : i(in).t = \bot$ *and* $i'(in).0 = \bot \wedge \forall t \in \mathbb{N} : t \geq 1 \Rightarrow i'(in).t = \top$*. Then,* $i{\downarrow_1} = \{in \mapsto \bot\} = i'{\downarrow_1}$ *and* $u(i){\downarrow_1} = \{out \mapsto \bot\}$ *and* $u(i'){\downarrow_1} = \{out \mapsto \top\}$*. Thus,* $i{\downarrow_1} = i'{\downarrow_1} \wedge u(i){\downarrow_1} \neq u(i'){\downarrow_1}$*.*

A single TSSPF is well-suited to model the semantics of a deterministic component. However, as a TSSPF maps each input to exactly one output, TSSPFs are not general enough to model the semantics of underspecified components where the exact output to an input is not fully specified. We thus model the semantics components as sets of TSSPFs:

Definition 2 (Component Semantics Describing)**.** *Let* $I, O \subseteq C$ *be two disjoint sets of channels. A set of TSSPFs* $F \subseteq [I^\Omega \xrightarrow{wc} O^\Omega]$ *is called component semantics describing iff it satisfies* $\forall g \in [I^\Omega \xrightarrow{wc} O^\Omega] : ((\forall i \in I^\Omega : \exists f \in F : g(i) = f(i)) \Rightarrow g \in F)$*.*

The definition above makes the semantics domain of components fully abstract [16, 17] in the sense of [19] and allows to handle unbounded non-determinism [16]. Full abstraction is achieved by the closeness property, which requires that each TSSPF resulting from a combination of TSSPFs included in the set F is also included in F. The closeness property is also important to make component semantics as little distinguishing as possible. This is illustrated by the fact that two different arbitrary sets of TSSPFs may encode the same component behaviors. The reason for this is that one may find a TSSPF $g \notin F$ that is not included in a set of TSSPFs F, which can be interpreted as a combination of different TSSPFs contained in F. It thus does not induce a new behavior not already covered by a TSSPF in F but, for instance, induces a semantic difference between a component with semantics described by F and a component with semantics described by $F \cup \{g\}$. As a result the semantics of two components that have the exact same observable behaviors may be considered unequal. Consequently, full abstraction is not achieved. Thereby, the closeness property is necessary. However, each arbitrary set of TSSPFs $F \subseteq [I^\Omega \xrightarrow{wc} O^\Omega]$ can be lifted to a component semantics describing set of TSSPFs $\overline{F} \stackrel{\text{def}}{=} \{g \in [I^\Omega \xrightarrow{wc} O^\Omega] \mid \forall i \in I^\Omega : \exists f \in F : g(i) = f(i)\}$ that satisfies $F \subseteq \overline{F}$ and $\overline{\overline{F}} = \overline{F}$.

3.2.1. Composition of TSSPFs

Composition is an important concept to achieve modularity. Composing the semantics of the individual components of a system leads to the semantics of the whole system. Composing arbitrary sets of TSSPFs can lead to realizability problems in delay-free feedback loops where the input of a component in time unit t depends on another component's output in time unit t and vice versa. Thus, composition is only defined for TSSPFs where causality between inputs and outputs on channels connected via a feedback loop is ensured. This is the case if one of the TSSPFs participating in a composition is strongly causal with respect to its channels connected by the composition. Intuitively, a TSSPFs f is strongly causal modulo the input channels J and output channels P, if the function's outputs on the channels in P until time unit $t + 1$ is not influenced by the function's inputs received on the channels in J after time unit t. Other than with weak causality, this especially includes that the outputs do not rely on the inputs received in the *same* time unit.

Definition 3 (Strongly Causal Modulo)**.** *Let* $f \in [I^\Omega \xrightarrow{wc} O^\Omega]$ *be a TSSPF and let* $J \subseteq I$ *and* $P \subseteq O$ *be two subsets of input and output channels names. The TSSPF f is called strongly causal modulo* (J, P) *iff* $\forall i, i' \in I^\Omega : \forall t \in \mathbb{N} :$

$((i|_J){\downarrow_t} = (i'|_J){\downarrow_t} \wedge i|_{I \setminus J} = i'|_{I \setminus J}) \Rightarrow f(i)|_P{\downarrow_{t+1}} = f(i')|_P{\downarrow_{t+1}}$.

The following example illustrates that there are weakly causal TSSPFs that are not strongly causal.

Example 5. *The function* $add \in [I^\Omega \xrightarrow{wc} O^\Omega]$ *as defined in Example 3 and depicted in Figure 6 is not strongly causal modulo* (I, O)*. This holds because the function's output in a time unit always depends on its present input. To formally show that add is not strongly causal modulo* (I, O)*, we need to find two inputs* $i, i' \in I^\Omega$ *and a time unit* $t \in \mathbb{N}$ *such that* $i|_I\downarrow_t = i'|_I\downarrow_t$ *and* $add(i)|_O\downarrow_{t+1} \neq add(i')|_O\downarrow_{t+1}$*. We chose* i *and* i' *such that* $\forall t \in \mathbb{N} : i(a).t = i(b).t = 1$ *and* $\forall t \in \mathbb{N} : i'(a).t = 2 \wedge i'(b).t = 1$*. Then,* $i|_I\downarrow_0 = \{c \mapsto \langle\rangle\} = i'|_I\downarrow_0$ *and* $add(i)|_O\downarrow_1 = \{c \mapsto 2\}$ *and* $add(i')|_O\downarrow_1 = \{c \mapsto 3\}$*. Thus,* $i|_I\downarrow_t = i'|_I\downarrow_t$ *and* $add(i)|_O\downarrow_{t+1} \neq add(i')|_O\downarrow_{t+1}$*, which we needed to show. Using the same counterexample, it is possible to show that add is not strongly causal with respect to* $(\{a\}, O)$*, either. Analogously, it can be shown that add is not strongly causal modulo* $(\{b\}, O)$*.*

The following example describes a strongly causal TSSPF:

Example 6 (Strongly Causal TSSPF)**.** *Consider the strongly causal TSSPF* $acc \in [I^\Omega \xrightarrow{wc} O^\Omega]$ *where* $I = \{c\}$ *and* $O = \{b\}$ *and* $type(c) = type(b) = \mathbb{N} \cup \{\varepsilon\}$*. The interface of the TSSPF is graphically illustrated on the right hand side of Figure 6. The TSSPF acc specifies the behavior of an accumulator component. In each time unit, the component outputs the sum of the values it received in the past. The component initially outputs the message* 0*, which reflects that it has not received positive integers, yet. When the component receives a positive integer in a time unit, it outputs the sum of the received integer and the value emitted in the current time unit in the next time unit. When the accumulator receives the empty message* ε*, the accumulated value remains unchanged. Thus, in the next time unit, the component again emits the value that it emits in the current time unit. Thus, the output of the function acc at time unit* $t + 1$ *only depends on its input up to and including time unit t. We formally define the TSSPF acc by the following equation:*

$$\forall i \in I^\Omega : \forall t \in \mathbb{N} : acc(i)(b).t = \begin{cases} 0 & \text{if } t = 0 \\ acc(i)(b).(t-1) + i(c).(t-1) & \text{if } t \geq 1 \wedge i(c).(t-1) \in \mathbb{N} \\ acc(i)(b).(t-1) & \text{if } t \geq 1 \wedge i(c).(t-1) = \varepsilon \end{cases}$$

The function acc is strongly causal modulo (I, O) *by definition. This can be formally proven by induction over the length of prefixes of input communication histories:*

$t = 0$*: The property is satisfied because the TSSPF add always outputs the same initial value in time unit* $t = 0$*, independent of its inputs in time unit* $t = 0$*.*

Let $n \in \mathbb{N}$*. Assume for all* $t \leq n$ *and all* $i, i' \in I^\Omega$*, it holds that* $i|_I\downarrow_t = i'|_I\downarrow_t \Rightarrow acc(i)|_O\downarrow_{t+1} = acc(i')|_O\downarrow_{t+1}$*.*

Let $t = n + 1$*.*

Let $i, i' \in I^\Omega$ *such that* $i|_I\downarrow_t = i'|_I\downarrow_t$*.*

We need to show $acc(i)|_O\downarrow_{t+1} = acc(i')|_O\downarrow_{t+1}$*.*

Using the induction hypothesis: $acc(i)|_O\downarrow_t = acc(i')|_O\downarrow_t$*.*

Therefore, especially $acc(i)(b).(t-1) = acc(i')(b).(t-1)$*.*

By assumption $i|_I\downarrow_t = i'|_I\downarrow_t$ *and thus* $i(c).(t-1) = i'(c).(t-1)$*.*

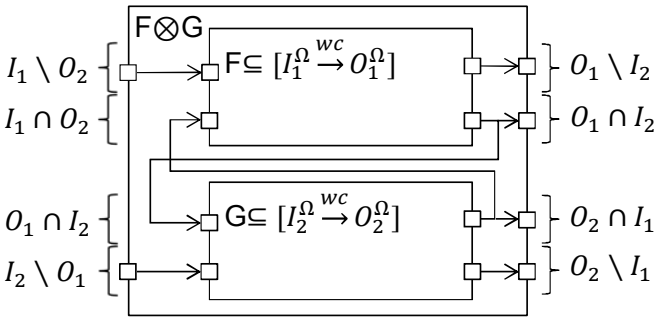

Figure 7: Graphical representation of the composition of two sets of TSSPFs.

As $t = n + 1$*, we have that* $t \geq 1$*.*
If $i(c).(t-1) = i'(c).(t-1) \in \mathbb{N}$*, then the above implies*
$acc(i)(b).t = acc(i)(b).(t-1) + i(c).(t-1) =$
$acc(i')(b).(t-1) + i'(c).(t-1)\} = acc(i')(b).t$
Similarly, if $i(c).(t-1) = i'(c).(t-1) = \varepsilon$*, then*
$acc(i)(b).t = acc(i)(b).(t-1) = acc(i')(b).(t-1) = acc(i')(b).t$
We can conclude that $acc(i)|_O\downarrow_{t+1} = acc(i')|_O\downarrow_{t+1}$*.*

A set of TSSPFs F is called strongly causal with respect to (J, P) iff there exists a function $f \in F$ that is strongly causal with respect to (J, P). With this, the set of TSSPFs F is required to exhibit at least one realization that is strongly causal with respect to (J, P). The causality complication is avoided, if causality between the inputs and outputs on the connected channels of at least one composition participant is guaranteed:

Definition 4 (Composable)**.** *Two sets of TSSPFs* $F_1 \subseteq [I_1^\Omega \xrightarrow{wc} O_1^\Omega]$ *and* $F_2 \subseteq [I_2^\Omega \xrightarrow{wc} O_2^\Omega]$ *are called composable iff* F_1 *is strongly causal with respect to* $(I_1 \cap O_2, I_2 \cap O_1)$ *or* F_2 *is strongly causal modulo* $(I_2 \cap O_1, I_1 \cap O_2)$*.*

Example 7 (Composability)**.** *The TSSPFs add and acc are described and formally defined in Example 3 and Example 6. The interfaces of the TSSPFs are graphically presented in Figure 6. Let* $Add = \{add\}$ *and* $Acc = \{acc\}$ *denote the singleton sets containing the TSSPFs add and acc. The two sets of TSSPFs are composable because, as shown in Example 6, the TSSPF* $acc \in Acc$ *is strongly causal modulo* $(\{c\}, \{b\}) = (I_{acc} \cap O_{add}, O_{acc} \cap I_{add})$*.*

Components communicate with each other via unidirected, typed channels established by connectors connecting component interfaces. Multiple components may read from the same channel, whereas only one component is permitted to write messages on a channel. This ensures that no merging of messages emitted from different components via the same channel is necessary. Thus the output channels of the functions of two sets of TSSPFs need to be disjoint to enable composition. The output channels of the composition result are the output channels of both composition's participants. The compound's input channels are exactly the input channels of both components that are no output channels of any of the two components.

The composition of two sets of TSSPFs F and G is graphically illustrated in Figure 7. The input channels of $F \otimes G$ are

the input channels $I_1 \setminus O_2$ of F that are no output channels of G and the input channels $I_2 \setminus O_1$ of G that are no output channels of F. The output channels of $F \otimes G$ are all output channels of F and G. The composition of two sets of TSSPFs yields a set of TSSPFs:

Definition 5 (Composition)**.** *Let $F_1 \subseteq [I_1^\Omega \xrightarrow{wc} O_1^\Omega]$ and $F_2 \subseteq [I_2^\Omega \xrightarrow{wc} O_2^\Omega]$ be two component semantics describing and composable sets of TSSPFs with disjoint output channel sets $O_1 \cap O_2 = \emptyset$. Let $I = (I_1 \setminus O_2) \cup (I_2 \setminus O_1)$ and $O = O_1 \cup O_2$. The composition $F_1 \otimes F_2 \subseteq [I^\Omega \xrightarrow{wc} O^\Omega]$ of F_1 and F_2 is defined by*

$$F_1 \otimes F_2 \stackrel{\text{def}}{=} \{f \mid \forall i \in I^\Omega : \exists f_1 \in F_1 : \exists f_2 \in F_2 : f(i) = o + p, \\ \text{where } o = f_1((i+p)|_{I_1}), p = f_2((i+o)|_{I_2})\}$$

The composition operator is defined similar as in [16, 17, 39] with the difference that we consider the time-synchronous system model instead of the more general timed system model [7].

Example 8. *The following demonstrates the composition of sets of TSSPFs by example. Let $Add = \{add\}$ and $Acc = \{acc\}$ be sets of TSSPFs as defined in Example 7. The sets Add and Acc are composable (cf. Example 7). As both sets contain a single TSSPF, the sets are component semantics describing (cf. Definition 2). Further, the sets of output channels of the sets' TSSPFs are disjoint. Thus, the composition operator $\otimes$ is applicable. Figure 8 graphically illustrates the result from composing the two sets Add and Acc. The set of TSSPFs $Add \otimes Acc$ contains the single TSSPF $f \in [\{a\}^\Omega \xrightarrow{wc} \{b,c\}^\Omega]$ that satisfies $\forall i \in \{a\}^\Omega : f(i) = o + p$ where $o = add((i+p)|_{I_{add}})$ and $p = acc((i+o)|_{I_{acc}})$. Given an input $i \in \{a\}^\Omega$, iteratively computing the values of o, p, c, and b is possible because the TSSPF acc is strongly casual. For instance, let $i = \{a \mapsto 1, 1, ...\} \in \{a\}^\Omega$ denote the communication history that always assigns channel a to 1, i.e., $\forall t \in \mathbb{N} : i(a).t = 1$. The first output of acc via channel b is by definition always 0 (cf. Example 6). With this, we can compute that add outputs $1 = 0 + 1$ via channel c in time unit 0. This determines the output 1 of acc at time unit 1. This again enables to determine that add outputs $2 = 1 + 1$ via channel c in time unit 1. This determines that acc outputs value 3 via channel b in time unit 2. Thus, add outputs 4 via channel c in time unit 2. We can approximate the value of the TSSPF f up to every fixed time unit $t \in \mathbb{N}$ for every fixed input $i \in \{a\}^\Omega$ by using the method sketched above.*

The composition is well defined and results in a component semantics describing set of TSSPFs. This is a consequence of the requirement that one set of TSSPFs must be strongly causal modulo its connected channels.

Theorem 1. *If F_1 and F_2 are two component semantics describing and composable sets of TSSPFs with disjoint output channel sets, then $F_1 \otimes F_2$ is also component semantics describing.*

Proof. Analogous to proof of Theorem 9 in [16] by replacing the set the function f is chosen from with $[I^\Omega \xrightarrow{wc} O^\Omega]$. □

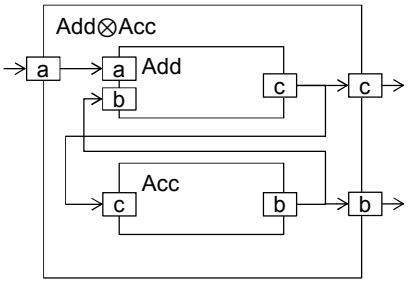

Figure 8: Graphical representation of the composition of *Add* and *Acc*.

4. Time-Synchronous Channel Automata

A TSCA specifies the behavior (of parts) of an interactive system and represents a component semantics describing set of TSSPFs that is given by its semantics. We later use TSCAs to model components. TSCAs as introduced in this paper are based on our previous work on TSPAs [9] and are strongly inspired by port automata [16], I/O* automata [36, 39], and MAA_{ts} automata [34]. Port automata and I/O* automata consume and produce time slices of arbitrary but finitely many input messages in every transition step. In contrast, TSCAs, TSPAs, and MAA_{ts} automata consume and produce at most one message per input channel in each time slice. Given the set of states and the channel types of an automaton are finite, MAA_{ts} automata, TSPAs, and TSCAs are guaranteed to have finitely many transitions. This is not the case for I/O* and port automata since both have to define a transition for each state and each possible input communication history. Even if the type of a channel is finite, the number of communication histories (streams) of the channel's type is infinite. I/O* automata and MAA_{ts} automata enforce causality between input and output histories by requiring initial outputs on all channels. In contrast, TSPAs and TSCAs do not require initial outputs. While the syntax of MAA_{ts} and TSCAs models variables explicitly, in TSPAs [9] variables have to be represented implicitly in the state space. While MAA_{ts} automata distinguish between data and control states (*i.e.,* variables and (control) states), TSCAs consist of data states (variables) only. This eliminates unnecessary complexity and notational clutter as control states can be easily represented as data states. Some proofs of some theorems presented in the following are analog to proofs that have already been carried out in [9, 16]. In case we are stating an analogous theorem, we refer to the appropriate corresponding proof in [9, 16].

A TSCA consists of a set of states, an interface of input and output channels, and transitions defining the TSCA's behavior. The interface is encoded by a channel signature.

Definition 6 (Channel Signature)**.** *Let $I, O \subseteq C$ be two disjoint sets of channel names. A channel signature is a tuple $\Sigma = (I, O)$. We denote by $C(\Sigma) \stackrel{\text{def}}{=} I \cup O$ the set of all channels in Σ. A channel signature Σ is called finite iff $C(\Sigma)$ and $type(c)$ for all $c \in C(\Sigma)$ are finite.*

A channel assignment maps channels to messages of the

channels' types. Let $B \subseteq C$. A *channel assignment* is an element of the set $B^{\rightarrow}$ defined as $B^{\rightarrow} \stackrel{\text{def}}{=} \{a \in [B \rightarrow M] \mid \forall b \in B : a(b) \in type(b)\}$. Channel assignments are used as TSCA transition labels.

Definition 7 (TSCA)**.** *A time-synchronous channel automaton is a tuple $A = (\Sigma, X, S, \iota, \delta)$ where:*

- $\Sigma = (I, O)$ *is a channel signature,*
- $X \subseteq C$ *is a set of variable symbols (internal channels),*
- $S \subseteq X^{\rightarrow}$ *is a set of states,*
- $\iota \in S$ *is the initial state,*
- $\delta \subseteq S \times C(\Sigma)^{\rightarrow} \times S$ *is the transition relation.*

For convenience, we sometimes write $s \xrightarrow{\theta}_{\delta} t$ instead of $(s, \theta, t) \in \delta$ and simply $s \xrightarrow{\theta} t$ if δ is clear from the context. To avoid notational clutter, we often denote the components of a TSCA $A = (\Sigma, X, S, \iota, \delta)$ with $\Sigma = (I, O)$ by $\Sigma_A \stackrel{\text{def}}{=} \Sigma$, $X_A \stackrel{\text{def}}{=} X$, $S_A \stackrel{\text{def}}{=} S$, $\iota_A \stackrel{\text{def}}{=} \iota$, $\delta_A \stackrel{\text{def}}{=} \delta$, $I_A \stackrel{\text{def}}{=} I$, and $O_A \stackrel{\text{def}}{=} O$. We also omit the subscripts if they are clear from the context.

Example 9 (TSCA of the component `CBC`)**.** *The TSCA of the component `CBC` is similar to the behavior automaton of the `CBC` component, which is graphically depicted in Figure 4. The channel signature comprises input and output channels. States and transitions reflect states and transitions in the behavior automaton, and the internal channel state reflects the current state of the behavior automaton. We interpret the absence of a message ("ε") equal to the Boolean value "$false$" and, again, denote "$\top$" as the Boolean value "true". The TSCA of the component `CBC` then can be formulated as $TSCA_{CBC} = (\Sigma_{CBC}, X_{CBC}, S_{CBC}, \iota_{CBC}, \delta_{CBC})$ with*

- *channel signature $\Sigma_{CBC} = (I_{CBC}, O_{CBC}) = (\{i, r\}, \{v, q\})$,*
- *channel data types: $type(i) = type(r) = type(v) = type(q) = \{\top, \varepsilon\}$,*
- *internal channel $X_{CBC} = \{state\}$ with $type(state) = \{0, 1\}$,*
- *states $S_{CBC} = X^{\rightarrow}_{CBC} = \{a, b\}$ with $a = \{state \mapsto 0\}$ and $b = \{state \mapsto 1\}$,*
- *initial state $\iota_{CBC} = a$,*
- *and transition relation $\delta_{CBC} =$ {*
  $\{(a, \theta, a) \mid (\theta(i) = \varepsilon \vee \theta(r) = \top) \wedge \theta(v) = \varepsilon \wedge \theta(q) = \varepsilon\}$
  $\cup \{(a, \theta, b) \mid \theta(i) = \top \wedge \theta(r) = \varepsilon \wedge \theta(v) = \top \wedge \theta(q) = \varepsilon\}$
  $\cup \{(b, \theta, b) \mid \theta(i) = \varepsilon \wedge \theta(r) = \varepsilon \wedge \theta(v) = \top \wedge \theta(q) = \varepsilon\}$
  $\cup \{(b, \theta, a) \mid \theta(r) = \top \wedge \theta(v) = \varepsilon \wedge \theta(q) = \varepsilon\}$
  $\cup \{(b, \theta, a) \mid \theta(i) = \top \wedge \theta(r) = \varepsilon \wedge \theta(v) = \varepsilon \wedge \theta(q) = \top\}$.*

The reactions of a TSCA are defined by its transitions. In each time unit, a TSCA performs one state change by executing one transition enabled by its input and outputs one message on each output channel. Let A be a TSCA. A is said to be *reactive* iff $\forall s \in S : \forall i \in I^{\rightarrow} : \exists t \in S : \exists \theta \in C(\Sigma)^{\rightarrow} : (s, \theta, t) \in \delta \wedge \theta|_I = i$. A reactive TSCA is called *component*. Components must not block their environments and must be able to react to any possible well-typed input in any time unit. Therefore, a component must define a reaction to every possible input for each of its states. A is called *finite* iff Σ and S are finite. The *size* of A, denoted $|A|$, is defined as the sum of the number of its states and transitions, *i.e.,* $|A| = |S| + |\delta|$. A is called *deterministic* iff $\forall s \in S : \forall i \in I^{\rightarrow} : |\{t \in S \mid \exists \theta \in C(\Sigma)^{\rightarrow} : \theta|_I = i \wedge (s, \theta, t) \in \delta\}| = 1$. A is called *I/O-deterministic* iff $\forall s \in S : \forall \theta \in C(\Sigma)^{\rightarrow} : |\{t \in S \mid (s, \theta, t) \in \delta\}| \leq 1$. Reactive TSCAs are adequate models for components as they specify at least one output for each input. The size of TSCAs is used for measuring algorithmic complexities. Intuitively, if A is deterministic, then for each state and each input, A only has at most one choice for switching the state when processing the input. It thus acts as a system part in a deterministic implementation. If A is reactive and deterministic, then it has exactly one choice for switching its state. In contrast, if A is I/O-deterministic, for each state, the state A switches to when it reads an input and produces an output can be uniquely identified by the input/output pair. As shown in Section 5, semantic differencing of I/O-deterministic TSCAs is possible in polynomial time in the sizes of the automata.

Example 10 ($TSCA_{CBC}$ is reactive and deterministic)**.** *$TSCA_{CBC}$ (cf. Example 9) is reactive because in both of its states, there is an outgoing transition with a channel assignment that has all input channels in its domain. In other words, it defines an output for each possible state/input combination and therefore it describes a component. $TSCA_{CBC}$ is finite, because $|S|$ and Σ are finite: The set of states S is finite since $|S| = 2$. The channel signature Σ is finite because $|C(\Sigma)| = 4$ and $\forall c \in C(\Sigma) : |type(c)| = |\{\top, \varepsilon\}| = 2$. $TSCA_{CBC}$ is reactive and deterministic because in both states and for each possible input, there is exactly one state that the TSCA may change to.*

The following theorem shows that determinism implies I/O-determinism. The other direction, however, does not hold.

Theorem 2. *Any deterministic TSCA is I/O-deterministic.*

Proof. Let $A = (\Sigma, X, S, \iota, \delta)$ with $\Sigma = (I, O)$ be a deterministic TSCA. Suppose towards a contradiction there exists a state s and a channel valuation $\theta \in C(\Sigma)^{\rightarrow}$ such that $|\{t \in S \mid s \xrightarrow{\theta} t\}| > 1$. Thus, there exist $u, v \in S$ such that $u \neq v$ and $s \xrightarrow{\theta} u$ and $s \xrightarrow{\theta} v$. Let $i = \theta|_I$. Then, as $u \neq v$ and $s \xrightarrow{\theta} u$, it holds that $u, v \in \{t \in S \mid \exists \theta \in C(\Sigma)^{\rightarrow} : \theta|_I = i \wedge s \xrightarrow{\theta} t\}$. Thus, $|\{t \in S \mid \exists \theta \in C(\Sigma)^{\rightarrow} : \theta|_I = i \wedge s \xrightarrow{\theta} t\}| \geq 2$, which contradicts that A is deterministic. □

With this, problems that are efficiently solvable for I/O-deterministic TSCAs are at least as efficiently solvable for deterministic TSCAs.

4.1. Execution and Behavior Semantics of TSCAs

This section formalizes the intuitive descriptions of a TSCA's behavior. Further analyses on TSCAs will be based on the operational semantics introduced in this section.

Definition 8 (Execution)**.** *Let $A = (\Sigma, X, S, \iota, \delta)$ be a TSCA. An execution σ of A is an infinite, alternating sequence of states and channel assignments starting with the initial state ι:*

$\sigma = s_0, \theta_0, s_1, \theta_1, ...$ such that $s_0 = \iota$ and $\forall i \in \mathbb{N} : s_i \xrightarrow{\theta_i} s_{i+1}$. The set of all executions of A is denoted by $execs(A)$.

Executions comprise the state changes and interactions performed by a TPSA. Abstracting from state changes allows to treat TSCAs as black boxes with hidden internal details. This requires explicating the *behavior* of a TSCA.

Definition 9 (Behavior)**.** *Let $A = (\Sigma, X, S, \iota, \delta)$ be a TSCA with channel signature $\Sigma = (I, O)$. The behavior of an execution $\sigma = s_0, \theta_0, s_1, \theta_1, ...$ of A is defined as the sequence $beh(\sigma) \stackrel{\text{def}}{=} \theta_0, \theta_1, ...$ containing only channel assignments. For $P \subseteq C(\Sigma)$, the restriction of $beh(\sigma)$ to P is defined as $beh(\sigma)|_P \stackrel{\text{def}}{=} \theta_0|_P, \theta_1|_P, ...$ We denote by $behs(A) \stackrel{\text{def}}{=} \bigcup_{\sigma \in execs(A)} beh(\sigma)$ the set of all behaviors of all executions of A. The named communication history h_α induced by a behavior $\alpha \in behs(A)$ with $\alpha = e_0, e_1, ...$ is defined as the function $h_\alpha \in (I \cup O)^\Omega$ that satisfies $h_\alpha(x).t = e_t(x)$ for all $x \in I \cup O$ and $t \in \mathbb{N}$.*

Given a TSCA A with $\Sigma_A = (I, O)$ and an input history $i \in I^\Omega$, we denote the set of communication histories induced A with input i by $A[i] \stackrel{\text{def}}{=} \{o \in O^\Omega \mid \exists \alpha \in behs(A) : o = h_\alpha|_O \wedge h_\alpha|_I = i\}$.

Example 11 (Execution and behavior of $TSCA_{CBC}$)**.** *An execution of a TSCA is an infinite sequence in general. Let a, b, θ_{ab}, θ_{ba}, θ_{res}, and θ_{nop} be given as follows:*

- $a = \{state \mapsto 0\}$, $b = \{state \mapsto 1\}$,
- $\theta_{ab} = \{i \to \top, r \mapsto \varepsilon, v \mapsto \top, q \mapsto \varepsilon\}$,
- $\theta_{ba} = \{i \mapsto \top, r \mapsto \varepsilon, v \mapsto \top, q \mapsto \top\}$,
- $\theta_{res} = \{i \mapsto \top, r \mapsto \top, v \mapsto \varepsilon, q \mapsto \varepsilon\}$, *and*
- $\theta_{nop} = \{i \mapsto \varepsilon, r \mapsto \varepsilon, v \mapsto \varepsilon, q \mapsto \varepsilon\}$.

An execution of the $TSCA_{CBC}$, for instance, is $e = a, \theta_{ab}, b, \theta_{ba}, a, \theta_{ab}, b, \theta_{res}, a(, \theta_{nop}, a)^\infty$. Accordingly, the behavior of this execution is given by $beh(e) = \theta_{ab}, \theta_{ba}, \theta_{ab}, \theta_{res}(, \theta_{nop})^\infty$. This behavior can be restricted to include only a subset of the involved channels, which is done by restricting the individual channel assignments. For instance, the restriction $e|_{\{q\}}$ of e to $\{q\}$ is given by $e|_{\{q\}} = \theta_{ab}|_{\{q\}}, \theta_{ba}|_{\{q\}}, \theta_{ab}|_{\{q\}}, \theta_{res}|_{\{q\}}(, \theta_{nop}|_{\{q\}})^\infty = \{q \mapsto \varepsilon\}, \{q \mapsto \top\}, \{q \mapsto \varepsilon\}, \{q \mapsto \varepsilon\}, \{q \mapsto \varepsilon\}^\infty$. The communication history h_e induced by the behavior e maps the channel q, for instance, to the stream $h_e(q) = \varepsilon, \top, \varepsilon, \varepsilon, \varepsilon^\infty$.

If a state is not visited by any of the TSCA's executions, then it is not productive in the sense that it does not influence any behavior. Thus, when analyzing the set of behaviors of a TSCA it suffices to analyze only the TSCA's reachable part that only consists of states visited by at least one execution. A state is reachable in a TSCA if there is an execution that visits it.

Definition 10 (Reachable)**.** *Let $A = (\Sigma, X, S, \iota, \delta)$ be a TSCA with channel signature $\Sigma = (I, O)$. A state $s \in S$ is called reachable in A if there exists a finite alternating sequence of states $s_0, \theta_1, s_1, \theta_2, ..., \theta_n, s_n$ starting in the initial state $s_0 = \iota$ and ending in state $s = s_n$ such that $s_i \xrightarrow{\theta_{i+1}} s_{i+1}$ for all $0 \leq i < n$. The set of all reachable states in A is denoted by $reach(A)$.*

Non-reachable states are redundant in the sense that they do not affect a TSCA's behavior.

Example 12 (Reachable states in $TSCA_{CBC}$)**.** *In $TSCA_{CBC}$, both states are reachable because $reach(TSCA_{CBC}) = \{a, b\}$. The execution e depicted in Example 11 reaches both states of the TSCA. To this effect, any prefix of e ending in state a and any prefix of e ending in state b are valid finite alternating sequences of states and channel assignments. This shows that both states are reachable.*

Removing the unreachable states from a TSCA results in a TSCA with exactly the same behaviors.

Theorem 3. *Let $A = (\Sigma, X, S, \iota, \delta)$ be a TSCA with channel signature $\Sigma = (I, O)$ and let $R = reach(A)$ denote the reachable states of A. Then, $B \stackrel{\text{def}}{=} (\Sigma, R, \iota, \delta \cap R \times C(\Sigma)^\to \times R)$ is a TSCA that satisfies $behs(A) = behs(B)$.*

Proof. Let A and B be given as above and let $\Delta = \delta \cap R \times C(\Sigma)^\to \times R$ denote the transitions of B.

$behs(A) \subseteq behs(B)$: Let $\sigma = s_0, \theta_1, s_1, \theta_2, s_2 ...$ be an execution of A. Then, it holds that $s_0 = \iota$ and $\forall i \in \mathbb{N} : s_i \xrightarrow{\theta_{i+1}}_\delta s_{i+1}$. Now, let $j \in \mathbb{N}$. As $\forall i \in \mathbb{N} : s_i \xrightarrow{\theta_{i+1}}_\delta s_{i+1}$ is satisfied, it especially holds that $s_i \xrightarrow{\theta_{i+1}}_\delta s_{i+1}$ for each $0 \leq i < j$. Thus, the finite sequence $s_0, \theta_1, s_1, \theta_2, s_2, ..., \theta_j, s_j$ satisfies $s_i \xrightarrow{\theta_{i+1}}_\delta s_{i+1}$ for all $0 \leq i < j$. From this, we can conclude that each state s_j where $j \in \mathbb{N}$ is reachable in A. As $\forall i \in \mathbb{N} : s_i \in R$, we have that $\forall i \in \mathbb{N} : (s_i, \theta_{i+1}, s_{i+1}) \in R \times C(\Sigma)^\to \times R$. From this and $\forall i \in (s_i \theta_{i+1}, s_{i+1}) \in \delta$, we can conclude $(s_i \theta_{i+1}, s_{i+1}) \in \Delta = \delta \cap R \times C(\Sigma)^\to \times R$, *i.e.,* $\forall i \in \mathbb{N} : s_i \xrightarrow{\theta_{i+1}} s_{i+1}$. From the above we can conclude $\sigma \in execs(B)$. All in all, we obtain $execs(A) \subseteq execs(B)$ and therefore $behs(A) \subseteq behs(B)$.

$behs(B) \subseteq behs(A)$: Let $\sigma = s_0, \theta_0, s_1, \theta_1, ...$ be an execution of A. Then, it holds that $s_0 = \iota$ and $\forall i \in \mathbb{N} : s_i \xrightarrow{\theta_i} s_{i+1}$. As $R \subseteq S$ and $\Delta \subseteq \delta$, we obtain $\forall s, t \in R : \forall \theta \in C(\Sigma)^\to : s \xrightarrow{\theta}_\Delta \Rightarrow s \xrightarrow{\theta}_\delta t$. Therefore, $\forall i \in \mathbb{N} : s_i \xrightarrow{\theta_i}_\Delta s_{i+1}$ implies $\forall i \in \mathbb{N} : s_i \xrightarrow{\theta_i}_\delta s_{i+1}$. Thus, it holds that $\sigma \in execs(A)$. We can conclude $execs(B) \subseteq execs(A)$ and therefore $behs(B) \subseteq behs(A)$. □

Algorithm 1 shows a procedure for removing the unreachable states from any finite TSCA. The algorithm performs a depth-first traversal on the input TSCA to only retain the input TSCA's states that are reachable from its initial state. While traversing the automaton, the algorithm also adds the transitions originating from any reachable state to the resulting automaton. As any state that is the target of any transition with a reachable source state is also reachable, the transitions added in Algorithm 1 always connect reachable states. The operations *push*, *pop*, and

top denote the standard stack operations and the symbol $\bot$ denotes the empty stack. The algorithm terminates because the input TSCA is required to be finite and every state is visited at most once.

Algorithm 1 Trimming a finite TSCA.

Input: Finite TSCA $A = (\Sigma, X, S, \iota, \delta)$
Output: TSCA containing only reachable parts of A

```
define R ← {ι} as set            /* reachable, visited states */
define U ← ⊥ as empty stack      /* states to visit */
define δ' ← ∅ as set
push(ι, U)
while U ≠ ⊥ do
  s ← top(U)
  δ' ← δ' ∪ {t ∈ δ | ∃θ ∈ C(Σ)^→ : ∃r ∈ S : t = (s, θ, r)}
  if {r ∈ S | ∃θ ∈ C(Σ)^→ : (s, θ, r) ∈ δ} ⊆ R then
    pop(U)
  else
    let s' ∈ {r ∈ S | ∃θ ∈ C(Σ)^→ : (s, θ, r) ∈ δ} \ R be
    arbitrary
    push(s', U)
    R ← R ∪ {s'}
  end if
end while
return (Σ, R, ι, δ')
```

Removing the unreachable states from a component again results in a component. Thus, the reactivity property is not lost by removing unreachable states.

Theorem 4. *Let $A = (\Sigma, X, S, \iota, \delta)$ be a component with channel signature $\Sigma = (I, O)$ and let $R = reach(A)$ denote the reachable states of A. Then, $B \stackrel{\text{def}}{=} (\Sigma, R, \iota, \delta \cap R \times C(\Sigma)^{\rightarrow} \times R)$ is a component.*

Proof. Let A and B be given as above and let $\Delta = \delta \cap R \times C(\Sigma)^{\rightarrow} \times R$ denote the transitions of B.

We need to show that B is reactive: Let $r \in R$ be a state of B. As $r \in R$ is a reachable state in A, it clearly holds that each target state of any transition in A with source state r is also an element of R, *i.e.,* $\forall u \in S : (\exists \theta \in C(\Sigma)^{\rightarrow} : s \xrightarrow{\theta} u) \Rightarrow u \in R$. Thus, we have that $\{(s, \theta, t) \in \delta \mid s = r\} \subseteq R \times C(\Sigma)^{\rightarrow} \times R$. As further $\{(s, \theta, t) \in \delta \mid s = r\} \subseteq \delta$, it holds that $\{(s, \theta, t) \in \delta \mid s = r\} \subseteq \delta \cap R \times C(\Sigma)^{\rightarrow} \times R$. This is equivalent to $\forall t \in S : r \xrightarrow{\theta}_{\delta} t \Rightarrow r \xrightarrow{\theta}_{\Delta} t$. As A is reactive and each transition of A starting from a reachable state $r \in R$ is also a transition of B, we obtain that B is also reactive. □

Therefore, the resulting from trimming a component is again an equivalent component that uses less space than the original. This eases analyses of the original component's behaviors.

4.2. Composition of TSCAs

As for TSSPFs, causality expresses the dependency between the inputs and outputs of a TSCA. A TSCA's output in time t must be completely determined by its input until time t. Thus it cannot change messages sent in the past and cannot predict messages it receives in the future (*cf.* pulse-drivenness in [16]):

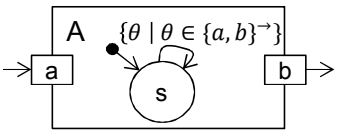

Figure 9: A strongly causal TSCA A that permits every possible output in reaction to every possible input.

Definition 11 (Weakly Causal TSCA)**.** *A TSCA A with $\Sigma_A = (I, O)$ is called weakly causal iff*

$\forall i, i' \in I^{\Omega} : \forall t \in \mathbb{N} : i{\downarrow_t} = i'{\downarrow_t} \Rightarrow A[i]{\downarrow_t} = A[i']{\downarrow_t}$.

Weak causality states that for every two inputs i, i' having a common prefix of length t and for every behavior $\alpha \in A[i]$ there is a behavior $\beta \in A[i']$ having a common prefix of length t with α. Similar as for TSSPFs, weak causality can lead to composition complications, which are avoidable analogously.

Definition 12 (Strongly Causal Modulo)**.** *Let A be a TSCA with channel signature $\Sigma_A = (I, O)$ and let $J \subseteq I$ and $P \subseteq O$ be two sets of input and output channels of A. The TSCA A is called strongly causal modulo (J, P) iff*

$\forall i, i' \in I^{\Omega} : \forall t \in \mathbb{N} :$

$((i|_J){\downarrow_t} = (i'|_J){\downarrow_t} \wedge i|_{I \setminus J} = i'|_{I \setminus J}) \Rightarrow (A[i]|_P){\downarrow_{t+1}} = (A[i']|_P){\downarrow_{t+1}}$.

Intuitively, a TSCA is strongly causal with respect to (J, P), if its outputs on the channels in P until time $t+1$ are not influenced by its inputs on the channels in J after time t.

Example 13 (Strongly Causal Modulo: $TSCA_{CBC}$)**.** *The TSCA $TSCA_{CBC}$, for instance, is not strongly causal with respect to $(\{r\}, \{v\})$. This is simple to show by contradiction: Let $in = \{r \mapsto \top^{\infty}, i \mapsto \top^{\infty}\} \in I^{\Omega}_{CBC}$ and $in' = \{r \mapsto \varepsilon^{\infty}, i \mapsto \top^{\infty}\} \in I^{\Omega}_{CBC}$ be two input histories. As $(in|_{\{r\}}){\downarrow_0} = (in'|_{\{r\}}){\downarrow_0} = \{r \mapsto \langle\rangle\}$ and $in|_{\{i\}} = in'|_{\{i\}} = \{i \mapsto \top^{\infty}\}$, the premises of the implication in Definition 12 hold for the chosen input histories and time $t = 0$. But as $(TSCA_{CBC}[in]|_{\{v\}}){\downarrow_1} = \langle\{v \mapsto \varepsilon\}\rangle$ and $(TSCA_{CBC}[in']|_{\{v\}}){\downarrow_1} = \langle\{v \mapsto \top\}\rangle$, the conclusion is not satisfied. Thus, the property stated in Definition 12 does not hold and $TSCA_{CBC}$ is not strongly causal modulo $(\{r\}, \{v\})$.*

At first sight, it might seem that a TSCA is strongly causal if, and only if, it always delays it's outputs. However, delaying of outputs is only a sufficient, not a necessary condition for a TSCA to be strongly causal. This holds because a TSCA A might simultaneously model a realization that is not strongly causal and another realization that is strongly causal, *i.e.,* a deterministic strongly causal component that only exhibits behaviors that are also possible in A. An example TSCA modeling arbitrary behavior illustrates this situation:

Example 14 (Arbitrary Behavior is Strongly Causal)**.** *Let $a, b \in C$ be two channels over Boolean values, i.e., $type(a) = type(b) = \{\varepsilon, \bot, \top\}$. Further, let $e \in C$ be a channel that only permits the empty message, i.e., $type(e) = \{\varepsilon\}$. We define the reactive TSCA A as illustrated in Figure 9 that is able to react with every possible output to every possible input as follows: $\Sigma_A = (\{a\}, \{b\})$, $X_A = e$, $S_A = \{s\}$, $\iota_A = s$,*

$\delta_A = \{(s, \theta, s) \mid \theta \in \{a, b\}^{\rightarrow}\}$ *where* $s = \{e \mapsto \varepsilon\}$. *It is easy to proof by induction that A is strongly causal modulo* (I_A, O_A) *because A permits every possible output in reaction to every possible input. Intuitively, this holds because when interpreting A as specification, we can find a strongly causal implementation I (a reactive deterministic component) that implements A, i.e.,* $behs(I) \subseteq behs(A)$. *An example for I is a TSCA that always outputs* ε *via channel b, independent of the input on channel a.*

TSCAs communicate with each other via their input and output channels. Multiple automata may read from the same channel, whereas only one automaton is permitted to write messages on a channel. Thus, no merging of messages on channels emitted by different automata is necessary.

Definition 13 (Compatible Channel Signatures)**.** *Two channel signatures* $\Sigma_A = (I_A, O_A)$ *and* $\Sigma_B = (I_B, O_B)$ *are called compatible iff* $O_A \cap O_B = \emptyset$.

By composing two TSCAs, the output channels of one automaton are connected to the input channels with the same name of the other automaton. The connected input channels are hidden implicitly. The set of output channels of the new automaton is the union of the sets of the output channels of the two original TSCAs. The input channels of the new automaton are the input channels of the two automata that do not share a common name with the output channels of the other automaton.

Definition 14 (Composition of Signatures)**.** *The composition of two channel signatures* $\Sigma_A = (I_A, O_A)$ *and* $\Sigma_B = (I_B, O_B)$ *is defined as* $\Sigma_A \otimes \Sigma_B \stackrel{\text{def}}{=} (I, O)$ *where* $I = (I_A \setminus O_B) \cup (I_B \setminus O_A)$ *and* $O = (O_A \cup O_B)$.

The composition of two TSCAs should be a TSCA that represents the behaviors of the TSCAs when they run in parallel. Therefore, we require the TSCAs participating in a composition must not share any internal variables (states).

Definition 15 (Compatible TSCAs)**.** *Two TSCAs A and B are called compatible iff* Σ_A *and* Σ_B *are compatible and* $X_A \cap X_B = \emptyset$.

Figure 10 illustrates the composition of two TSCAs A and B. The input channels of the compound $A \otimes B$ is the union of the input channels of A and B minus the union of the output channels of both TSCAs. The output channels of $A \otimes B$ are exactly the output channels of A and B. The composition of the TSCAs' states and transitions reflect the parallel execution of both TSPAs. The following formally defines the composition operator for TSCAs.

Definition 16 (Composition of TSCA)**.** *Let A and B be two compatible TSCAs. The composition of A and B is defined as the TSCA* $A \otimes B \stackrel{\text{def}}{=} (\Sigma, X, S, \iota, \delta)$ *where*

- $\Sigma = \Sigma_A \otimes \Sigma_B$,
- $X = X_A \cup X_B$,
- $S = \{s_A \cup s_B \mid s_A \in S_A \wedge s_B \in S_B\}$
- $\iota = \iota_A \cup \iota_B$

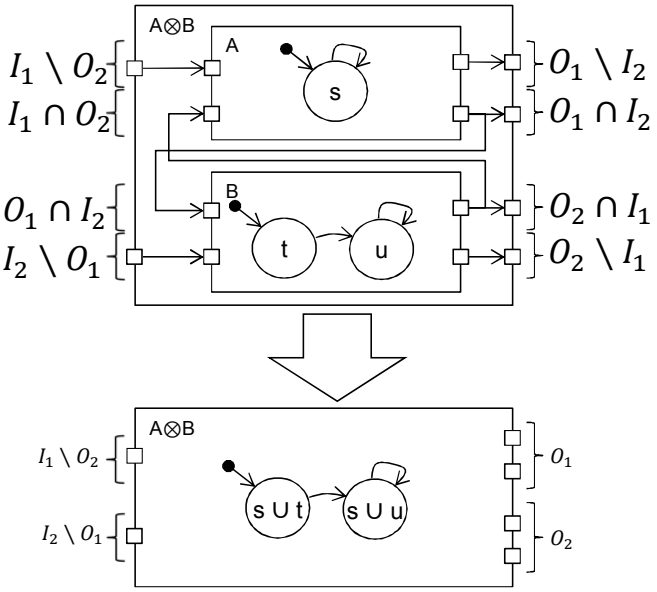

Figure 10: Composition of two compatible TSCAs.

- $\delta = \{(s, \theta, t) \in S \times C(\Sigma)^{\rightarrow} \times S \mid$
  $(s|_{S_A}, \theta|_{C(\Sigma_A)}, t|_{S_A}) \in \delta_A \wedge (s|_{S_B}, \theta|_{C(\Sigma_B)}, t|_{S_B}) \in \delta_B\}$

The union of the functions of S_A and S_B used in the definition of S (cf. Definition 16) is well defined since the the functions' domains X_A and X_B are disjoint.

Example 15 (Composition of two instances of $TSCA_{CBC}$)**.** *This example describes the composition of the TSCAs of the components* `pos0` *and* `pos1` *as depicted in Figure 3 (c). In MontiArcAutomaton, port names of different components may be equal and connectors establish channels between connected ports. In contrast, TSCAs communicate via shared channels. With this, a connector between two MontiArcAutomaton components describes a channel in the TSCA that formally describes the composed component's behaviors. Thus, the port names of the MontiArcAutomaton components have to be adjusted to achieve compatibility on TSCA level. We denote the TSCA of* `pos0` *by* CBC_0 *and the TSCA of* `pos1` *by* CBC_1. *The two TSCAs as well as their compound are depicted in Figure 11.*

They are defined by $CBC_0 = ((I_0, O_0), S_0, X_0, \iota_0, \delta_0)$ *and* $CBC_1 = ((I_1, O_1), S_1, X_1, \iota_1, \delta_1)$ *with*

- *input channels* $I_0 = \{i, r\}$ *and* $I_1 = \{q_0, r\}$ *where* $type(c) = \{\varepsilon, \top\}$ *for all* $c \in I_0 \cup I_1$,
- *output channels* $O_0 = \{x_0, q_0\}$ *and* $O_1 = \{q_1, x_1\}$ *where* $type(c) = \{\varepsilon, \top\}$ *for all* $c \in O_0 \cup O_1$,
- *internal channels* $X_0 = \{state_0\}$ *and* $X_1 = \{state_1\}$ *where* $type(state_0) = type(state_1) = \{0, 1\}$,
- *states* $S_0 = \{s_0, s_1\}$ *and* $S_1 = \{t_0, t_1\}$ *where* $s_i = \{state_0 \mapsto i\}$ *and* $t_i = \{state_1 \mapsto i\}$ *for all* $i \in \{0, 1\}$,
- *initial states* $\iota_0 = s_0$ *and* $\iota_1 = t_0$,

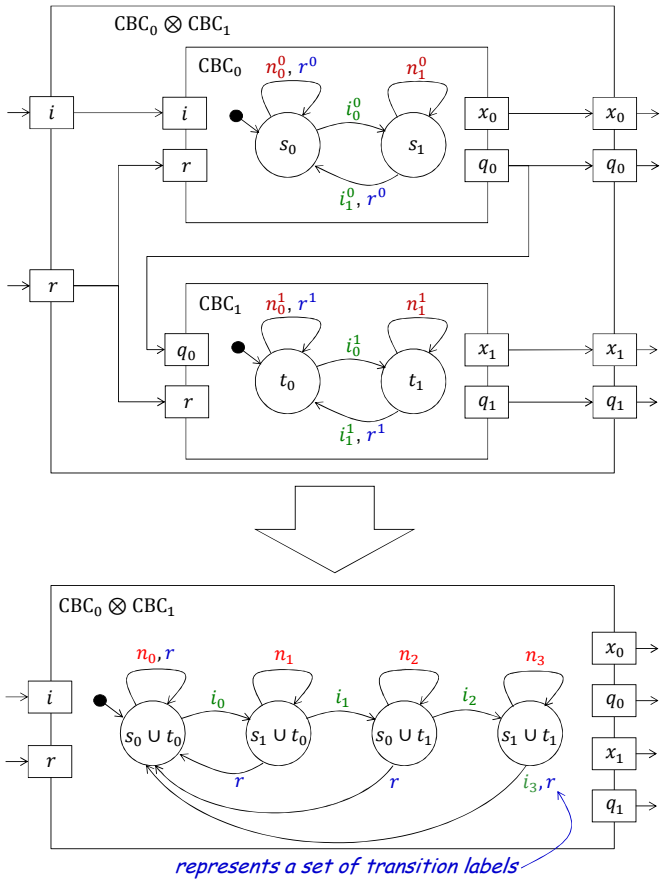

Figure 11: Composition of two CBC instances.

- *transition relations as depicted in the top part of Figure 11 where the transition labels of CBC_0 are defined as:*

  $n_0^0 = \{i \mapsto \varepsilon, r \mapsto \varepsilon, x_0 \mapsto \varepsilon, q_0 \mapsto \varepsilon\}$,

  $n_1^0 = \{i \mapsto \varepsilon, r \mapsto \varepsilon, x_0 \mapsto \top, q_0 \mapsto \varepsilon\}$,

  $i_0^0 = \{i \mapsto \top, r \mapsto \varepsilon, x_0 \mapsto \top, q_0 \mapsto \varepsilon\}$,

  $i_1^0 = \{i \mapsto \top, r \mapsto \varepsilon, x_0 \mapsto \varepsilon, q_0 \mapsto \top\}$,

  $r^0 = \{\theta \in (I_0 \cup O_0)^{\rightarrow} \mid \theta(r) = \top \wedge \theta(x_0) = \varepsilon \wedge \theta(q_0) = \varepsilon\}$,

  and the transition labels of CBC_1 are defined as:

  $n_0^1 = \{q_0 \mapsto \varepsilon, r \mapsto \varepsilon, x_1 \mapsto \varepsilon, q_1 \mapsto \varepsilon\}$,

  $n_1^1 = \{q_0 \mapsto \varepsilon, r \mapsto \varepsilon, x_1 \mapsto \top, q_1 \mapsto \varepsilon\}$,

  $i_0^1 = \{q_0 \mapsto \top, r \mapsto \varepsilon, x_1 \mapsto \top, q_1 \mapsto \varepsilon\}$,

  $i_1^1 = \{q_0 \mapsto \top, r \mapsto \varepsilon, x_1 \mapsto \varepsilon, q_1 \mapsto \top\}$,

  $r^1 = \{\theta \in (I_0 \cup O_0)^{\rightarrow} \mid \theta(r) = \top \wedge \theta(x_1) = \varepsilon \wedge \theta(q_1) = \varepsilon\}$.

The TSCAs CBC_0 and CBC_1 are compatible because the channel signatures are compatible ($O_0 \cap O_1 = \emptyset$) and the internal channels are pairwise disjoint $X_0 \cap X_1 = \{state_0\} \cap \{state_1\} = \emptyset$.

The composed TSCA $CBC_0 \otimes CBC_1$ is depicted in the lower part of Figure 11 and is formally given by $CBC_0 \otimes CBC_1 = (\Sigma, X, S, \iota, \delta)$ with

- *the channel signature $\Sigma = \Sigma_0 \otimes \Sigma_1 = (\{i, r\}, \{q_0, x_0, q_1, x_1\})$,*
- *internal channels $X = \{\{state_0\}, \{state_1\}\}$,*
- *states $S = \{s_{00}, s_{01}, s_{10}, s_{11}\}$, where*
  $s_{00} = \{\{state_0 \mapsto 0\}, \{state_1 \mapsto 0\}\}$,
  $s_{01} = \{\{state_0 \mapsto 0\}, \{state_1 \mapsto 1\}\}$,
  $s_{10} = \{\{state_0 \mapsto 1\}, \{state_1 \mapsto 0\}\}$, *and*
  $s_{11} = \{\{state_0 \mapsto 1\}, \{state_1 \mapsto 1\}\}$
- *the initial state $\iota = s_{00}$, and*
- *the transition relation as depicted in the bottom part of Figure 11 where the transition labels of $CBC_0 \otimes CBC_1$ are defined as:*

  $i_0 = \{i \mapsto \top, r \mapsto \varepsilon, x_0 \mapsto \top, q_0 \mapsto \varepsilon, x_1 \mapsto \varepsilon, q_1 \mapsto \varepsilon\}$,

  $i_1 = \{i \mapsto \top, r \mapsto \varepsilon, x_0 \mapsto \varepsilon, q_0 \mapsto \top, x_1 \mapsto \top, q_1 \mapsto \varepsilon\}$,

  $i_2 = \{i \mapsto \top, r \mapsto \varepsilon, x_0 \mapsto \top, q_0 \mapsto \varepsilon, x_1 \mapsto \top, q_1 \mapsto \varepsilon\}$,

  $i_3 = \{i \mapsto \top, r \mapsto \varepsilon, x_0 \mapsto \varepsilon, q_0 \mapsto \top, x_1 \mapsto \varepsilon, q_1 \mapsto \top\}$,

  $n_0 = \{i \mapsto \varepsilon, r \mapsto \varepsilon, x_0 \mapsto \varepsilon, q_0 \mapsto \varepsilon, x_1 \mapsto \varepsilon, q_1 \mapsto \varepsilon\}$,

  $n_1 = \{i \mapsto \varepsilon, r \mapsto \varepsilon, x_0 \mapsto \top, q_0 \mapsto \varepsilon, x_1 \mapsto \varepsilon, q_1 \mapsto \varepsilon\}$,

  $n_2 = \{i \mapsto \varepsilon, r \mapsto \varepsilon, x_0 \mapsto \varepsilon, q_0 \mapsto \varepsilon, x_1 \mapsto \top, q_1 \mapsto \varepsilon\}$,

  $n_3 = \{i \mapsto \varepsilon, r \mapsto \varepsilon, x_0 \mapsto \top, q_0 \mapsto \varepsilon, x_1 \mapsto \top, q_1 \mapsto \varepsilon\}$,

  $r = \{\theta \mid \theta(r) = \top \wedge \theta(q_0) = \theta(q_1) = \theta(x_0) = \theta(x_1) = \varepsilon\}$.

The result of this composition of two CBC components, i.e., two mod-2 counters, is a mod-4 counter. In this composed TSCA, all four states are reachable.

Components can block each other if they simultaneously require an input emitted by the other component to produce the next output. Composing such components results in a TSCA that is not reactive and therefore no component. However, there is a sufficient condition ensuring the resulting transition relation is reactive and the compound is a component.

Definition 17 (Composability of TSCAs)**.** *Two components A and B are called composable iff*

- *A and B are compatible and*
- *A is strongly causal with respect to $(I_A \cap O_B, I_B \cap O_A)$ or B is strongly causal with respect to $(I_B \cap O_A, I_A \cap O_B)$.*

Example 16 (Composability of $TSCA_{CBC}$)**.** *As shown in Example 15, the $TSCA_0$ of `pos0` and the $TSCA_1$ of `pos1` are compatible. To show composability between these, it is to show that $TSCA_0$ is strongly causal modulo $(I_0 \cap O_1, I_1 \cap O_0)$. This holds because $I_0 \cap O_1 = \emptyset$ and $I_1 \cap O_0 = \{q_0\}$: It is not possible that the messages emitted via an output channel of CBC_1 influence the behavior of CBC_0 because no output channel of CBC_1 is an input channel of CBC_0.*

The following theorem states that composing two composable components always results in a well-formed component.

Theorem 5. *If A and B are composable components, then the reachable part of $A \otimes B$ is a component.*

Proof. Analogous to proof of Theorem 3 in [16] by replacing the set the function i is chosen from with $I^{\rightarrow}$. □

Example 17 (The composition of two $TSCA_{CBC}$ is a component)**.** *Example 16 shows that the TSCAs CBC_0 of `pos0` and the CBC_1 of `pos1` are composable. Further, Example 10 proves that CBC_0 and CBC_1 are reactive, i.e., describe components. With Theorem 5, the composition of $TSCA_0$ and $TSCA_1$ is a component as it can be seen in Example 15.*

The composition operator further is commutative and associative. This guarantees the component resulting from composing several components is independent of the order in which the components are composed. Section 5.4 defines a notion of system architecture, which is well-defined because of the associativity and commutativity of the TSCA composition operator.

Theorem 6. *If A, B, and C are three pairwise compatible TSCAs, then the following holds:*

1. *$A \otimes B$ and C are compatible,*
2. *$A \otimes B = B \otimes A$, and*
3. *$(A \otimes B) \otimes C = A \otimes (B \otimes C)$.*

Proof. Let A, B, and C be three pairwise compatible TPSAs.

$A \otimes B$ and C are compatible: As A, B, and C are pairwise compatible, it holds that $X_A \cap X_B = X_A \cap X_C = X_B \cap X_C = \emptyset$. Thus, $X_{A\otimes B} \cap X_C = (X_A \cup X_B) \cap X_C = (X_A \cap X_C) \cup (X_B \cap X_C) = \emptyset$. As A, B, and C are pairwise compatible, it holds that Σ_A, Σ_B, and Σ_C are pairwise compatible and therefore $O_A \cap O_B = O_A \cap O_C = O_B \cap O_C = \emptyset$. Thus, it holds that $O_{A\otimes B} \cap O_C = (O_A \cup O_B) \cap O_C = (O_A \cap O_C) \cup (O_B \cap O_C) = \emptyset$. As $X_{A\otimes B} \cap X_C = O_{A\otimes B} \cap O_C = \emptyset$, $A \otimes B$ and C are compatible.

$A \otimes B = B \otimes A$: The set operations used in the definitions are all commutative. Commutativity for each part of the tuple follows directly by applying the sets' definitions.

$(A \otimes B) \otimes C = A \otimes (B \otimes C)$: Let $D = (A \otimes B) \otimes C$ and let $E = A \otimes (B \otimes C)$. As A, B, and C are all pairwise compatible, it holds by (1.) that $A \otimes B$ and C as well as A and $B \otimes C$ are compatible. The composition operator is therefore applicable for constructing D and E. Applying the operator, we obtain:

- $\Sigma_D = \Sigma_{A\otimes B} \otimes \Sigma_C$
  $= ((I_A \setminus O_B) \cup (I_B \setminus O_A), O_A \cup_B) \otimes \Sigma_C$
  $= (((I_A \setminus O_B) \cup (I_B \setminus O_A)) \setminus O_C \cup I_C \setminus (O_A \cup O_B), O_A \cup O_B \cup O_C)$
  $= (I_A \setminus (O_B \cup O_C) \cup I_B \setminus (O_A \cup O_C) \cup I_C \setminus (O_A \cup O_B), O_A \cup O_B \cup O_C)$
  $= (I_A \setminus (O_B \cup O_C) \cup (I_B \setminus O_C \cup I_C \setminus O_B) \setminus O_A, O_A \cup O_B \cup O_C)$
  $= \Sigma_A \otimes ((I_B \setminus O_C) \cup (I_C \setminus O_B), O_B \cup O_C) = \Sigma_A \otimes (\Sigma_B \otimes \Sigma_C)$
  $= \Sigma_E$,
- $X_D = X_A \cup X_B \cup X_C = X_E$,
- $S_D = \{s_A \cup s_B \cup s_C \mid s_A \in S_A \wedge s_B \in S_B \wedge s_C \in S_C\} = S_E$,
- $\iota_D = \iota_A \cup \iota_B \cup \iota_C = \iota_E$,
- $\delta_D = \{(s, \theta, t) \mid (s|_{S_{A\otimes B}}, \theta|_{C(\Sigma_{A\otimes B})}, t|_{S_{A\otimes B}}) \in \delta_{A\otimes B} \wedge (s|_{S_C}, \theta|_{C(\Sigma_C)}, t_{S_C}) \in \delta_C\}$
  $= \{(s, \theta, t) \mid (s|_{S_A}, \theta|_{C(\Sigma_A)}, t_{S_A}) \in \delta_A \wedge (s|_{S_B}, \theta|_{C(\Sigma_B)}, t_{S_B}) \in \delta_B \wedge (s|_{S_C}, \theta|_{C(\Sigma_C)}, t_{S_C}) \in \delta_C\}$
  $= \{(s, \theta, t) \mid (s|_{S_{B\otimes C}}, \theta|_{C(\Sigma_{B\otimes C})}, t|_{S_{B\otimes C}}) \in \delta_{B\otimes C} \wedge (s|_{S_A}, \theta|_{C(\Sigma_A)}, t_{S_A}) \in \delta_A\} = \delta_E$.

□

There exists a "neutral element" with respect to the composition operator. We will use this TSCA to lift the composition operator to arbitrary finite sets of TSCAs.

Definition 18 (Neutral TSCA)**.** *The neutral TSCA is defined as the TSCA N where $\Sigma_N = (\emptyset, \emptyset)$, $X_N = \emptyset$, $S_N = \{\emptyset\}$, $\iota_N = \emptyset$, and $\delta_N = \{(\emptyset, \emptyset, \emptyset)\}$. The neutral TSCA has no channels and no variables. Its sole and initial state is the empty channel valuation $v \in \emptyset^{\rightarrow} = [\emptyset \to M] = \{\emptyset\}$. It consists of one transition looping from the initial state to itself with the empty channel valuation.*

It is possible to compose the neutral TSCA with any other TSCA. It is the neutral element with respect to composition.

Theorem 7. *Let A be an arbitrary TSCA. Then, the TSCA A and the neutral TSCA N are compatible and $A \otimes N = A = N \otimes A$.*

Proof. Let A be an arbitrary TSCA. It holds that $O_A \cap O_N = O_A \cap \emptyset = \emptyset$. Thus Σ_A and Σ_N are compatible. As further $X_A \cap X_N = X_A \cap \emptyset = \emptyset$, we can conclude that A and N are compatible. The composition of A and N is $A \otimes N = (\Sigma, X, S, \iota, \delta)$ where $\Sigma = ((I_A \setminus \emptyset) \cup (\emptyset \setminus O_A), O_A \cup \emptyset) = (I_A, O_A)$, $X = X_A \cup \emptyset = X_A$, $S = \{s_A \cup s_B \mid s_A \in S_A \wedge s_B \in \{\emptyset\}\} = S_A$, $\iota = \iota_A \cup \emptyset = \iota_A$, $\delta = \{(s, \theta, t) \mid s|_{S_A} \xrightarrow{\theta|_{C(\Sigma_A)}}_{\delta_A} t|_{S_A} \wedge s|_{\{\emptyset\}} \xrightarrow{\theta|_{\emptyset}}_{\delta_N} t|_{\{\emptyset\}}\}$. As $s|_{\{\emptyset\}} \xrightarrow{\theta|_{\emptyset}}_{\delta_N} t|_{\{\emptyset\}}$ holds for each $\theta \in C(\Sigma)^{\rightarrow}$, the above is equal to $\{(s, \theta, t) \mid s|_{S_A} \xrightarrow{\theta|_{C(\Sigma_A)}}_{\delta_A} t|_{S_A}\} = \delta_A$. Hence, $A \otimes N = A$. By commutativity of $\otimes$ (cf. Theorem 6), we obtain $A = N \otimes A$. □

Theorem 6 guarantees that the TSCA resulting from composing several pairwise compatible TSCAs is independent of the order in which the TSCAs are composed. Theorem 7 shows that the neutral TSCA is a neutral element with respect to TSCA composition. We therefore lift the TSCA composition operator to the unique function $\bigotimes$ that takes a finite set of pairwise compatible TSCAs as input and outputs their composition under the operator $\otimes$ as usual, *i.e.,* $\bigotimes$ satisfies $\bigotimes \emptyset = N$ and $\bigotimes\{c\} = c$ for all TSCAs c and $\bigotimes(A \cup B) = (\bigotimes A) \otimes (\bigotimes B)$ for all all finite sets of TSCAs A and B such that $A \cap B = \emptyset$ and the TSCAs in $A \cup B$ are pairwise compatible. The operator is well-defined because of the properties stated in Theorem 6 and Theorem 7.

Naively applying the construction given in Definition 16 may cause the compound to consist of many unreachable states. Theorem 3 revealed that unreachable states can be safely removed from a TSCA without changing its behaviors. Unreachable states thus do not contribute to a TSCA's behavior. To defer a state explosion that occurs when composing several TSCAs with each other, adding unreachable states to TSCAs during a composition procedure should be avoided. Algorithm 2 depicts

an algorithm that takes two finite and composable TSCAs as input and outputs the trimmed TSCAs' compound. The algorithm performs a breadth-first search starting in the initial state of the compound. For each state determined as reachable, the algorithm calculates all transitions possible in the compound originating from the reachable state and checks whether the transitions' target has not been visited. In case the latter is true, the algorithm adds the state not yet visited to the set of states that are still to visit and proceeds as above.

Algorithm 2 Joined composition and trimming of finite TSCAs

Input: Two Finite and compatible TSCAs A and B
Output: Trimmed composition of A and B

```
define ι = ι_A ∪ ι_B as tuple                    /* initial state */
define δ ← ∅ as set                              /* transitions */
define R ← ∅ as set                              /* visited states */
define U ← ⊥ as empty stack                      /* states to visit */
push(ι, U)
while U ≠ ⊥ do
  s ← top(U)
  pop(U)
  R ← R ∪ {s}
  for all (u_1, θ_1, v_1) ∈ {t ∈ δ_A | ∃u : ∃θ : (s|_A, θ, u) = t} do
    for all (u_2, θ_2, v_2) ∈ {t ∈ δ_B | ∃u : ∃θ : (s|_B, θ, u) = t} do
      if θ_1|_{C(Σ_1)∩C(Σ_2)} = θ_2|_{C(Σ_1)∩C(Σ_2)} then
        define θ ← θ_1 + θ_2 as channel valuation
        δ ← δ ∪ {(s, θ, v_1 ∪ v_2)}
        if (v_1 ∪ v_2) ∉ R then
          push(v_1 ∪ v_2, U)
        end if
      end if
    end for
  end for
end while
return (Σ_1 ⊗ Σ_2, R, ι, δ)
```

Composition preserves I/O-determinism. This fact is important, because the size of the compound from composing several TSCAs is exponential in the number of the composed TSCAs. Thus, using the fact greatly reduces the complexity of determining whether a compound is I/O-deterministic if all the composition's participants are already I/O-deterministic. Section 5 describes the importance of I/O-determinism in detail: I/O-deterministic TSCAs induce a special structure when transforming them to Büchi automata, *i.e.,* the Büchi automata are always deterministic and weak, which enables to apply a simple complementation procedure.

Theorem 8. *If A and B are two I/O-deterministic and compatible TSCAs, then $A \otimes B$ is an I/O-deterministic TSCA.*

Proof. Let A and B be two I/O-deterministic and composable TSCAs. Let $A \otimes B = (\Sigma, X, S, \iota, \delta)$ denote the composition of A and B where $\Sigma = \Sigma_A \otimes \Sigma_B = (I, O)$. We need to show that $A \otimes B$ is I/O-deterministic. Suppose towards a contradiction that $A \otimes B$ is not I/O-deterministic. Then there exists a state $s \in S \subseteq X^{\rightarrow} = (X_A \cup X_B)^{\rightarrow}$ and a channel valuation $\theta \in C(\Sigma)^{\rightarrow}$ such that $|\{t \in S \mid s \xrightarrow{\theta}_{\delta} t\}| > 1$. This guarantees there exist $t, t' \in S$ with $t|_{X_A} \in S_A$ and $t|_{X_B} \in S_B$ and $t'|_{X_A} \in S_A$ and $t'|_{X_B} \in S_B$ such that $t \neq t'$ and $s \xrightarrow{\theta}_{\delta} t$ and $s \xrightarrow{\theta}_{\delta} t'$. By definition of composition for TSCAs we have that the following holds:

$s|_{X_A} \xrightarrow{\theta|_{C(\Sigma_A)}}_{\delta_A} t|_{X_A}$ and $s|_{X_A} \xrightarrow{\theta|_{C(\Sigma_A)}}_{\delta_A} t'|_{X_A}$ and $s|_{X_B} \xrightarrow{\theta|_{C(\Sigma_B)}}_{\delta_B} t|_{X_B}$ and $s|_{X_B} \xrightarrow{\theta|_{C(\Sigma_B)}}_{\delta_B} t'|_{X_B}$. Since $t \neq t'$, it holds that $t|_{X_A} \neq t'|_{X_A}$ or $t|_{X_B} \neq t'|_{X_B}$. The case $t|_{X_A} \neq t'|_{X_A}$ stands in contradiction to the assumption that A is I/O-deterministic, as this would imply $|\{t \in S_A \mid s|_{X_A} \xrightarrow{\theta|_{C(\Sigma_A)}} t\}| \geq 2$. Similarly, the case $t|_{X_B} \neq t'|_{X_B}$ stands in contradiction to the assumption that B is I/O-deterministic. □

Example 18 (The composition of two $TSCA_{CBC}$ instances is I/O-deterministic)**.** *Theorem 8 guarantees that the composition of CBC_0 and CBC_1 as depicted in Example 15 is I/O-deterministic, because CBC_0 and CBC_1 are I/O-deterministic and compatible. We will now reconsider this according to the proof of Theorem 8. If $CBC_0 \otimes CBC_1$ was not I/O-deterministic, the composition would have to have two transitions with the same channel valuation from a single state s to at least two other states t and t' (with $t \neq t'$). The fact that t and t' are different implies that the restrictions of t and t' to the internal variables of CBC_0 are different or the restrictions to the internal variables of CBC_1 are different. Therefore, in CBC_0 or CBC_1 there must be a transition from one source state to at least two different target states that have the same channel valuation. This is a contraction to the assumption that both CBC_0 and CBC_1 are I/O-deterministic.*

The behaviors of a compound $A \otimes B$ are all behaviors over $C(\Sigma_{A \otimes B})$ that are possible in A when restricted to the channels of A and possible in B when restricted to the channels of B. Section 5.4 later uses this fact in Theorem 18 to show that refinement of TSCAs is compatible with composition. This is an important property, which enables independent development of different system parts. The following formalizes this property.

Theorem 9. *Let A and B be two compatible TSCAs and let $C \stackrel{\text{def}}{=} A \otimes B$. It holds that $behs(C) = \{\alpha \in C(\Sigma_C)^{\infty} \mid \alpha|_{C(\Sigma_A)} \in behs(A) \wedge \alpha|_{C(\Sigma_B)} \in behs(B)\}$.*

Proof. Let A, B, and C be given as above.

$\subseteq$: Let $\alpha \in behs(C)$ and let $\sigma = s_0, \theta_1, s_1, \theta_2, s_2, ...$ be an execution of C such that $beh(\sigma) = \alpha$. By definition of execution its holds that $s_{j-1} \xrightarrow{\theta_j}_{\delta_C} s_j$ for all $j > 0$ and $s_0 = \iota_C$. By definition of composition it holds that $s_{j-1}|_{X_A} \xrightarrow{\theta_j|_{C(\Sigma_A)}}_{\delta_A} s_j|_{C(\Sigma_A)}$ and $s_{j-1}|_{X_B} \xrightarrow{\theta_j|_{C(\Sigma_B)}}_{\delta_B} s_j|_{C(\Sigma_B)}$ for all $j > 0$.

Further it holds that $s_0|_{X_A} = \iota_C|_{X_A} = (\iota_A \cup \iota_B)|_{X_A} = \iota_A$ and $s_0|_{X_B} = \iota_C|_{X_B} = (\iota_A \cup \iota_B)|_{X_B} = \iota_B$ because ι_A and ι_B are disjoint. We can conclude $\sigma_A \stackrel{\text{def}}{=} s_0|_{X_A}, \theta_1|_{C(\Sigma_A)}, s_1|_{X_A}, \theta_2|_{C(\Sigma_A)}, s_0|_{X_A}, ... \in execs(A)$ is an execution of A and $\sigma_B \stackrel{\text{def}}{=} s_0|_{X_B}, \theta_1|_{C(\Sigma_B)}, s_1|_{X_B}, \theta_2|_{C(\Sigma_B)}, s_0|_{X_B}, ... \in execs(B)$ is an execution of B. This implies $beh(\sigma_A) = \theta_1|_{C(\Sigma_A)}, \theta_2|_{C(\Sigma_A)}, ... \in behs(A)$ is a behavior of A and $beh(\sigma_B) = \theta_1|_{C(\Sigma_B)}, \theta_2|_{C(\Sigma_B)}, ... \in behs(B)$ is a behavior of B. We can observe that $beh(\sigma_A) = beh(\sigma)|_{C(\Sigma_A)} = \alpha|_{C(\Sigma_A)}$ and $beh(\sigma_B) =$

$beh(\sigma)|_{C(\Sigma_B)} = \alpha|_{C(\Sigma_B)}$. Thus, $\alpha|_{C(\Sigma_A)} \in behs(A)$ and $\alpha|_{C(\Sigma_B)} \in behs(B)$.

$\supseteq$: Let $\alpha = \theta_1, \theta_2, ... \in C(\Sigma_C)^\infty$ such that $\alpha|_{C(\Sigma_A)} \in behs(A)$ and $\alpha|_{C(\Sigma_B)} \in behs(B)$. Let $\sigma_A = s_0^A, \theta_1^A, s_1^A, \theta_2^A, s_2^A, ...$ be an execution of A such that $beh(\sigma_A) = \alpha|_{C(\Sigma_A)}$ and let $\sigma_B = s_0^B, \theta_1^B, s_1^B, \theta_2^B, s_2^B, ...$ be an execution of B such that $beh(\sigma_B) = \alpha|_{C(\Sigma_B)}$. By definition of execution it holds that $s_{j-1}^A \xrightarrow{\theta_j^A} s_j^A$ and $s_{j-1}^B \xrightarrow{\theta_j^B} s_j^B$ for all $j > 0$. As $\theta_i^A = \theta_i|_{C(\Sigma_A)}$ and $\theta_i^B = \theta_i|_{C(\Sigma_B)}$ for all $j > 0$, it holds that $s_{j-1}^A \xrightarrow{\theta_j|_{C(\Sigma_A)}} s_j^A$ and $s_{j-1}^B \xrightarrow{\theta_j|_{C(\Sigma_B)}} s_j^B$ for all $j > 0$. Using the definition of TSCA composition, we obtain $((s_{j-1}^A \cup s_{j-1}^B), \theta_j, (s_j^A \cup s_j^B)) \in \delta_C$ for all $j > 0$. As additionally $\iota_C = \iota_A \cup \iota_B = s_0^A \cup s_0^B$, it holds that $\sigma \stackrel{\text{def}}{=} (s_0^A \cup s_0^B), \theta_1, (s_1^A \cup s_1^B), \theta_2, (s_2^A \cup s_2^B), ... \in execs(C)$ is an execution of C. Observing that $beh(\sigma) = \alpha$, we can conclude $\alpha \in behs(C)$. □

Hiding is an important concept to achieve modularity. The channels present in the compound resulting from the composition of several other TSCAs is always the union of the output channels of the composed TSCAs. For specifying software architectures, it is often necessary to hide several output channels to the environment. This is, for example, useful to hide unnecessary information not relevant to the architecture's environment or to explicitly hide "secret" information. Hidden channels become internal channels of the compound. For example, the bottom architecture depicted in Figure 3 illustrates this: the output channel `q` of component `pos2` is not part of the interface of component `Mod8Counter`. It is hidden from the environment, *i.e.,* the TSCA representing the `Mod8Counter` is restricted to the output channels $\mathtt{x_0}$, $\mathtt{x_1}$, and $\mathtt{x_2}$.

Definition 19 (TSCA Channel Restriction)**.** *Let A be a TSCA and let $O \subseteq O_A$ be a set of output channels of A. The restriction of A to the channels in O is defined as the TSCA $A\restriction O = (\Sigma, X_A, S_A, \iota_A, \delta)$ where $\Sigma = (I_A, O)$ and $\delta = \{(s, \theta, t) \in S_A \times C(\Sigma)^\rightarrow \times S_A \mid \exists (u, \theta', v) \in \delta_A : u = s \wedge v = t \wedge \theta'|_{C(\Sigma)} = \theta\}$.*

The set of output channels in $A\restriction O$ is restricted to the channels in O. $A\restriction O$ has the same input channels, internal variables, and states as A. The TSCA $A\restriction O$ contains a transition for each transition of A where the transition's channel valuation is restricted to the channels present in $A\restriction O$.

Example 19 (Restriction of CBC_0)**.** *Example 15 describes the TSCA $CBC_0 = ((I_0, O_0), S_0, X_0, \iota_0, \delta_0)$. The restriction $CBC_0\restriction\{x_0\}$ of CBC_0 to the set of its output channels $\{o\}$ is depicted in Figure 12. It is defined as $CBC_0\restriction\{x_0\} = (\Sigma, S_0, X_0, \iota_0, \delta)$ where $\Sigma = (\{i, r\}, \{x_0\})$ with transition relation δ as depicted in Figure 12. Each individual transition label is restricted to the channels of $I_0 \cup \{x_0\} = \{i, r\} \cup \{x_0\}$.*

4.3. TSSPF semantics of TSCAs

This section defines the semantics of TSCAs by sets of TSSPFs and reveals an important relation between the composition operators: the semantics of the syntactic composition of two TSCAs A and B is equal to the composition of the semantics of the individual automata.

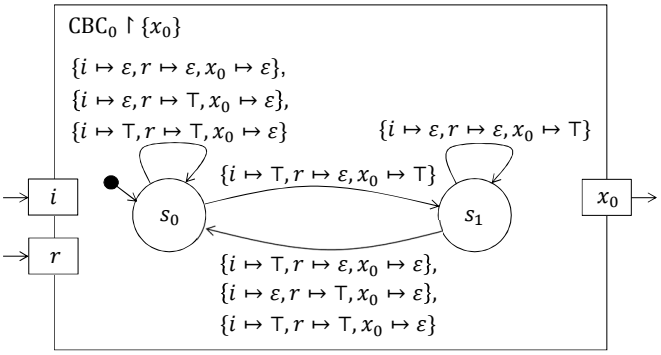

Figure 12: Graphical representation of the TSCA $CBC_0\restriction\{x_0\}$.

Definition 20 (TSSPF Semantics of a TSCA)**.** *The TSSPF semantics $[\![A]\!]$ of a TSCA $A = (\Sigma, X, S, \iota, \delta)$ with channel signature $\Sigma = (I, O)$ is defined as follows:*

$$[\![A]\!] \stackrel{\text{def}}{=} \{f \in [I^\Omega \xrightarrow{wc} O^\Omega] \mid \forall i \in I^\Omega : \exists \alpha \in behs(A) : i = h_\alpha|_I \wedge f(i) = h_\alpha|_O\}$$

Example 20 (TSSPF Semantics of CBC_0)**.** *The TSSPF semantics $[\![CBC_0]\!]$ of the TSCA $CBC_0 = ((I_0, O_0), S_0, X_0, \iota_0, \delta_0)$ (cf. Example 15) contains a single function f because CBC_0 is a deterministic component. For example, the function f maps the input communication history $h_I \in I_0^\Omega$ that satisfies $h(i).t = h(r).t = \varepsilon$ for all $t \in \mathbb{N}$ to the output channel history $h_O \in O_0^\Omega$ that satisfies $h_O(x_0).t = h_O(q_0).t = \varepsilon$ for all $i \in \mathbb{N}$. This holds because there exists a behavior $\alpha \in behs(CBC_0)$ (with execution looping in the initial state forever), which satisfies $\alpha.t(i) = \alpha.t(r) = \alpha.t(x_0) = \alpha.t(q_0) = \varepsilon$ for all $t \in \mathbb{N}$.*

For each behavior of a component, the semantics contain a function that maps inputs to outputs as encoded by the history induced by the behavior. Thus, no behavior is lost in the semantic mapping.

Theorem 10. *Let A be a component. For each $\alpha \in behs(A)$ there is a function $f \in [\![A]\!]$ such that $f(h_\alpha|_I) = h_\alpha|_O$.*

Proof. Analogous to proof of Theorem 11 in [16] by replacing the definition of maximality with $\forall i \in I^\Omega : i \in S|_I$. □

The semantics of components are well-formed, *i.e.,* components specify component semantics describing sets of TSSPFs.

Theorem 11. *The semantics $[\![A]\!]$ of a component A is component semantics describing.*

Proof. Analogous to proof of Theorem 12 in [16] by replacing the set the function f is chosen from with $[I^\Omega \xrightarrow{wc} O^\Omega]$. □

The semantics of the composition of two components is equal to the composition of their individual semantics:

Theorem 12. *For two composable components A and B with compatible signatures the following holds: $[\![A \otimes B]\!] = [\![A]\!] \otimes [\![B]\!]$.*

Proof. Analogous to proof of Theorem 13 in [16] by replacing the applications of $[\![\cdot]\!]$ for PAs and $\otimes$ for SPFs by applications of the corresponding definitions for TSCAs and TSSPFs. □

An important implication of the theorem is that we can first syntactically compose the individual automata of an architecture and then perform analysis on the semantics of the automaton encoding the behavior of the whole system. This gives another basis for analysis that does not necessarily require to compose the semantics of the individual components of a system as, for example, done in [38]. The next sections introduce a method for semantic differencing of TSCAs and additionally shows that semantic differencing for finite I/O-deterministic TSCAs is possible in polynomial time. This paper further defines a notion of system architecture based on TSCAs. Afterwards, we introduce a method for mitigating the state explosion problem during semantic differencing of finite system architectures. In our previous work [9], we only considered semantic differencing for TSPAs in general and we did not introduce the notion of I/O-determinism. It is straightforward to transfer the results to TSCAs. The definition of system architecture as introduced in this paper is not possible with TSPAs as introduced in [9] because TSPAs do not have a commutative and associative composition operator.

5. Semantic Differencing of Component Behavior: From TSCAs to BAs

After introducing the notations for Büchi Automata (BAs) used in this paper, this section presents a theorem stating that there is a non-deterministic BA for each finite TSCA that accepts exactly the behaviors of the TSCA. Afterwards, we show that refinement checking and semantic difference witness generation for finite TSCAs can be reduced to language inclusion checking and counterexample generation for non-deterministic BAs. For finite I/O-deterministic TSCAs, semantic differencing can even be reduced to language inclusion checking for deterministic BAs, which is possible in polynomial time in the sizes of the automata.

5.1. Büchi Automata

Büchi Automata [3, 8] are a variant of finite automata that are acceptors for infinite words and thus induce languages consisting of infinite words. They are well known and much used in model checking. Infinite words over an alphabet Π are infinite sequences of symbols in Π.

Definition 21 (Büchi Automaton)**.** *A BA is a tuple* (Π, Q, I, F, δ) *where* Π *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* $\delta \subseteq Q \times \Pi \times Q$ *is the transition relation.*

For convenience we again sometimes write $s \xrightarrow{\sigma}_\delta t$ instead of $t \in \delta(s, \sigma)$ and simply $s \xrightarrow{\sigma} t$ if δ is clear from the context. Let $\mathcal{B} = (\Pi, Q, I, F, \delta)$ be a BA. The size of $\mathcal{B}$, denoted $|\mathcal{B}|$ is defined as the number of states and transitions in B, *i.e.,* $|\mathcal{B}| = |Q| + |\delta|$. A run of $\mathcal{B}$ on a word $w = \sigma_1, \sigma_2 ... \in \Pi^\infty$ starting in a state $q_0 \in Q$ is an infinite sequence $q_0, q_1, ...$ such that $q_{j-1} \xrightarrow{\sigma_j}_\delta q_j$ for all $j > 0$. A run $q_0, q_1, ...$ is accepting if $q_0 \in I$ and $q_i \in F$ for infinitely many $i > 0$. The accepted language of $\mathcal{B}$ is defined as $\mathcal{L}(\mathcal{B}) \stackrel{\text{def}}{=} \{w \in \Pi^\infty \mid \text{there exists an accepting run for } w \text{ in } \mathcal{B}\}$. The BA $\mathcal{B}$ is called *deterministic* iff $|I| \leq 1$ and $\forall q \in Q : \forall \sigma \in \Pi : |\{t \in S \mid s \xrightarrow{\sigma} t\}| \leq 1$. $\mathcal{B}$ is called *total* iff $|I| = 1$ and $\forall q \in Q : \forall \sigma \in \Pi : |\delta(q, \sigma)| = 1$. A BA $\mathcal{B} = (\Pi, Q, I, F, \delta)$ is called *weak* iff for all pairs of states $p, q \in Q$ belonging to the same strongly connected component it holds that p is accepting iff q is accepting. Deterministic weak BAs can be minimized in polynomial time [27]. This enables to efficiently minimize intermediate BA representations of an architecture to mitigate a state explosion during composition. In the general case, the minimization problem is PSPACE-complete for non-deterministic BAs [4, 21] and NP-complete for deterministic BAs [41]. Checking language inclusion between two arbitrary non-deterministic Büchi automata is PSPACE-complete [23], though decidable, in general. Although the computational complexity is large, several approaches for checking language inclusion and counterexample (diff witness) generation have been implemented and produce promising results in practice [3]. Checking language inclusion $\mathcal{L}(A) \subseteq \mathcal{L}(B)$ is typically done in three steps by proving that there are no words in $\mathcal{L}(A)$, which are not included in $\mathcal{L}(B)$:

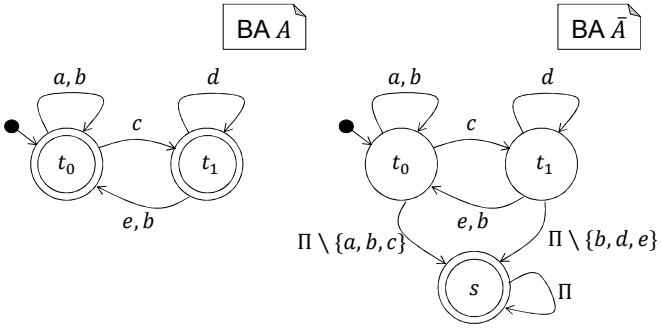

Figure 13: Two Büchi automata A and $\overline{A}$. The automaton $\overline{A}$ accepts the complementary language of A.

1. Construct a complementary automaton $\overline{B}$ of B that accepts exactly the words not accepted by B, *i.e.,* $\mathcal{L}(\overline{B}) = \Pi^\infty \setminus \mathcal{B}$.
2. Construct a Büchi automaton C that accepts exactly the words accepted by A and $\overline{B}$, *i.e.,* $\mathcal{L}(C) = \mathcal{L}(A) \cap \mathcal{L}(\overline{B})$.
3. Check whether $\mathcal{L}(C) = \emptyset$, which is possible by examining whether C contains a reachable final state that is part of a cycle.

The computational hardness of checking language inclusion arises from constructing the BA $\overline{B}$ that might be exponentially larger than the BA B in the general case [24, 40]. However, in case B is deterministic, the BA $\overline{B}$ can be constructed in polynomial time in the size of B [25].

Example 21 (Büchi Automata)**.** *Figure 13 depicts two BAs A and* $\overline{A}$*. The BA A is formally defined by* $A = (\Pi, Q, I, F, \delta)$ *where*

- $\Pi = \{a, b, c, d, e\}$,
- $Q = \{t_0, t_1, s\}$,
- $I = \{t_0\}$,
- $F = \{t_0, t_1\}$, *and*
- $\delta = \{(t_0, a, t_0), (t_0, b, t_0), (t_0, c, t_1), (t_1, d, t_1), (t_1, e, t_0), (t_1, b, t_0)\}$.

The automaton $\overline{A}$ is defined analogously. The BA $\overline{A}$ accepts exactly the complementary language of A, i.e., it holds that $\mathcal{L}(\overline{A}) = \Pi^\infty \setminus \mathcal{L}(A)$. Both automata are deterministic and weak.

In the next section, we present a translation from finite TSCAs to BAs and thereby reduce semantic differencing and refinement checking for finite TSCAs to the language inclusion problem for Büchi automata. We show that the translation transforms a rather large subclass of TSCAs to BAs that can be complemented in polynomial time in the sizes of the resulting BAs. The subclass contains all finite I/O-deterministic TSCAs.

5.2. From TSCAs to BAs

In model-driven development, models are the primary engineering artifacts, *i.e.,* engineers (manually) create finite models to describe parts of the system under development. Hence, we consider semantic differencing and refinement checking for architectures where the individual components have a finite state space, communicate over finitely many communication channels, and where the types of messages emitted via component interfaces are finite. There exists a non-deterministic BA for each finite TSCA that accepts exactly the TSCA's behaviors.

The BA associated to a finite TSCA $A = (\Sigma, X, S, \iota, \delta)$ with $\Sigma = (I, O)$ is defined as $ba(A) \stackrel{\text{def}}{=} (C(\Sigma)^{\rightarrow}, S, \{\iota\}, S, \delta)$. As the TSCA A is finite, the sets S, I, O, and δ are finite. As therefore $C(\Sigma)^{\rightarrow}$ is finite, $ba(A)$ is a well-defined BA. The size of $ba(A)$ is equal to the size of A. The following theorem shows that the language accepted by $ba(A)$ and the behaviors of A coincide.

Theorem 13. *For any finite TSCA A, it holds that $behs(A) = \mathcal{L}(ba(A))$.*

Proof. Let $A = (\Sigma, X, S, \iota, \delta)$ be a finite TSCA with channel signature $\Sigma = (I, O)$. Further let $ba(A) = (C(\Sigma)^{\rightarrow}, S, \{\iota\}, S, \delta)$ be the BA associated to A.

$\subseteq$: Let $s_0, \theta_1, s_1, \theta_2, s_2, ... \in execs(A)$ be an execution of A. By definition of execution $s_{j-1} \xrightarrow{\theta_j} s_j$ for all $j > 0$ and $s_0 = \iota$. Thus, $s_0, s_1, s_2, ...$ is a run of $\mathcal{B}$ on the word $\theta_1, \theta_2, ...$ Since all states $s \in S$ are accepting, the run is accepting. Thus, $beh(s_0, \theta_1, s_1, \theta_2, s_2, ...) = \theta_1, \theta_2, ... \in \mathcal{L}(\mathcal{B})$.

$\supseteq$: Assume that $\sigma = \sigma_1, \sigma_2, \sigma_3, ... \in \mathcal{L}(\mathcal{B})$ and let $q_0, q_1, q_2, ...$ be an accepting run of $\mathcal{B}$ on σ. By definition of run we have $q_{j-1} \xrightarrow{\sigma_j} q_j$ for all $j > 0$. Thus $\tau = q_0, \theta_1, q_1, \theta_2, ...$ is an execution of A. Therefore, by definition of behavior we have that $beh(\tau) = \sigma_1, \sigma_2, ... \in behs(A)$ is a behavior of A. □

Example 22. *The BA $ba(CBC_0)$ associated to the finite TSCA CBC_0 (cf. Example 15) is equal to the BA A depicted in Figure 13 when assuming $a = n_0^0$, $b = r^0$, $c = i_0^0$, $d = n_1^0$, $e = i_1^0$.*

The following reveals a sufficient condition that guarantees the translation of a TSCA to its associated BA yields a deterministic BA. As language inclusion checking for deterministic BAs is possible in polynomial time [25], we obtain a method for efficiently determining if the set of behaviors of a TSCA is a subset of the behaviors of another I/O-deterministic TSCA.

Theorem 14. *The associated BA $ba(A)$ of each finite and I/O-deterministic TSCA A is deterministic.*

Proof. Let $A = (\Sigma, X, S, \iota, \delta)$ be a finite and I/O-deterministic TSCA and let $ba(A) = (C(\Sigma)^{\rightarrow}, S, \{\iota\}, S, \delta)$ be the BA associated to A. The BA $ba(A)$ has a unique initial state. As the TSCA A is I/O-deterministic, it holds that $\forall s \in S : \forall \theta \in C(\Sigma)^{\rightarrow} : |\{t \in S \mid (s, \theta, t)\}| \leq 1$. This implies that $ba(A)$ is deterministic. □

Example 23 (The $ba(TSCA_{CBC})$ is deterministic)**.** *Example 10 shows that $TSCA_{CBC}$ is finite. Thus, the TSPAs state space S is also finite. $TSCA_{CBC}$ is I/O-deterministic, i.e., there is at most one state that the TSCA can change to, from a given source state and a given channel assignment. This follows from the fact that the TSCA is deterministic, which has been shown in Example 10 and the application of Theorem 2. According to the definition of BAs, a BA is deterministic if it has at most one initial state and for each state and for each input word, there is at most one state that the BA can change to. The BA $ba(TSCA_{CBC})$ has a single initial state and for each input, i.e., each channel assignment, there is only one transition from each state, because $TSCA_{CBC}$ is deterministic. With this (and the proof of Theorem 14), the constructed BA $ba(TSCA_{CBC})$ is deterministic.*

There exist non-deterministic BAs for which no deterministic BAs exist that accepts the same language. On the other hand, for each non-deterministic weak BA, there exists a deterministic weak BA that accepts the same language [27]. The translation from TSCAs to BAs always yields weak BAs, which can be determinized and minimized. Further, each deterministic and complete weak BA can be complemented in polynomial time by exchanging the automaton's sets of accepting and non-accepting states.

Theorem 15. *The associated BA $ba(A)$ of each finite TSCA A is weak.*

Proof. Let $A = (\Sigma, X, S, \iota, \delta)$ be a finite and I/O-deterministic TSCA and let $ba(A) = (C(\Sigma)^{\rightarrow}, S, \{\iota\}, S, \delta)$ be the BA associated to A. As every state in $ba(A)$ is accepting, it especially holds that each strongly connected component in $ba(A)$ solely contains accepting states. This implies that $ba(A)$ is weak. □

5.3. Semantic Differencing for Component Behavior

The semantics of components are defined as sets of TSSPFs. Each function $f \in [\![c]\!] \setminus [\![c']\!]$ in the semantics of one component c that is no member of the semantics of another component c' is a representative for the difference between the components' semantics. However, such a representative defines an output for each possible component input, even if the semantic difference is only given by a single input/output pair. Thus, such a TSSPF does not effectively reveal the differences between the

component semantics. In contrast, the exact input/output pairs for which there is a function in the semantics of one component that maps the input to the output and for which there is no function in the semantics of the other component mapping the input to the output clearly reveals a difference. If two components have different interfaces, *i.e.,* they read and write from and to different channels, each input/output pair of the first component indicates a difference to the semantics of the other component. However, if the components have channels of the same types one can easily avoid this problem by channel renaming and hiding [5]. Thus, we define the semantic difference for components having the same interfaces, only.

Definition 22 (Diff Witness)**.** *Let* $F_1, F_2 \subseteq [I^\Omega \xrightarrow{wc} O^\Omega]$ *be two sets of TSSPFs. A diff witness distinguishing* F_1 *from* F_2 *is a communication history* $w \in (I \cup O)^\Omega$ *satisfying* $\exists f_1 \in F_1 : f_1(w|_I) = w|_O \wedge \forall f_2 \in F_2 : f_2(w|_I) \neq w|_O$.

We denote by $\Delta(F_1, F_2)$ *the set of all diff witnesses distinguishing* F_1 *from* F_2.

A set of diff witnesses may be finite but is typically infinite and can thus not be completely enumerated.

Example 24 (Diff Witness)**.** *This example presents a diff witness between the* $TSCA_{CBC} = (\Sigma_{CBC}, X_{CBC}, S_{CBC}, \iota_{CBC}, \delta_{CBC})$ *and a modified version of it. The modified version* $TSCA_{mod} = (\Sigma_{CBC}, X_{CBC}, S_{CBC}, \iota_{CBC}, \delta_{mod})$ *has the same interface as* $TSCA_{CBC}$ *and a similar behavior – the only difference is that it does not emit* $\top$ *on the outgoing channel q if the state changes from b to a after an increase of the counted value. More technically,* $\delta_{mod} = (\delta_{CBC} \setminus \delta_{ba}) \cup \delta_{ba'}$, *where*
$\delta_{ba} = \{(b, \theta, a) \mid \theta(i) = \top \wedge \theta(r) = \varepsilon \wedge \theta(v) = \varepsilon \wedge \theta(q) = \top\}$ *and*
$\delta_{ba'} = \{(b, \theta, a) \mid \theta(i) = \top \wedge \theta(r) = \varepsilon \wedge \theta(v) = \varepsilon \wedge \theta(q) = \varepsilon\}$

Let $in = \{r \mapsto \langle \varepsilon^\infty \rangle, i \mapsto \langle \top, \top, \varepsilon^\infty \rangle\} \in I^\Omega$ *be an input history over the common interface of* $TSCA_{CBC}$ *and* $TSCA_{mod}$. *The input history describes two increase steps that change the state of the TSCA from a to b, back to a, and then remains in state a. For all* $h \in TSCA_{CBC}[in|_{\{q\}}]$ *and* $h' \in TSCA_{mod}[in|_{\{q\}}]$, *it holds that* $h.1 = \top$, *whereas* $h'.1 = \varepsilon$. *Therefore, for the given input history, the TSCAs produce different output histories.*

We consider architectures where the whole system behavior can be mapped to a TSCA. The following theorem reveals the relation between the differences of the behaviors and of the semantics of TSCAs.

Theorem 16. *Let* $A_1 = (\Sigma, S_1, \iota_1, \delta_1)$ *and* $A_2 = (\Sigma, S_2, \iota_2, \delta_2)$ *with* $\Sigma = (I, O)$ *be two TSCAs and let* $w \in (I \cup O)^\Omega$ *be a communication history. The following holds:* $w \in \Delta([\![A_1]\!], [\![A_2]\!]) \Leftrightarrow \exists \alpha \in behs(A_1) : w = h_\alpha \wedge \alpha \notin behs(A_2)$.

Proof. Let A_1, A_2, and w be given as above.

$\Rightarrow$: Assume $w \in \Delta([\![A_1]\!], [\![A_2]\!])$ is a diff witness. By definition of Δ, we have that there is a function $f_1 \in [\![A_1]\!]$ such that $f_1(w|_I) = w|_O$ and $f(w|_I) \neq w|_O$ for all $f \in [\![A_2]\!]$. In the following let f_1 be such a function that satisfies the above. By definition of $[\![\cdot]\!]$ we have that $\forall i \in I^\Omega : \exists \alpha \in behs(A_1) : i = h_\alpha|_I \wedge f_1(i) = h_\alpha|_O$. When substituting $w|_I$ for i, we get that $\exists \alpha \in behs(A_1) : w|_I = h_\alpha|_I \wedge f_1(w|_I) = h_\alpha|_O$. Since $f_1(w|_I) = w|_O$ we can substitute $w|_O$ for $f_1(w|_I)$ and obtain $\exists \alpha \in behs(A_1) : w|_I = h_\alpha|_I \wedge w|_O = h_\alpha|_O$, which is equivalent to $\exists \alpha \in behs(A_1) : w = h_\alpha$. In the following, let such an α with $w = h_\alpha$ be given. It remains to show $\alpha \notin behs(A_2)$. Towards a contradiction we assume $\alpha \in behs(A_2)$. By Theorem 10 we get there is a function $g \in [\![A_2]\!]$ such that $g(h_\alpha|_I) = h_\alpha|_O$. By definition of α we have $w = h_\alpha$ and thus $g(w|_I) = w|_O$. But since $w \in \Delta([\![A_1]\!], [\![A_2]\!])$, it holds that $\forall f \in [\![A_2]\!] : f(w|_I) \neq w|_O$. Substituting g for f yields a contradiction.

$\Leftarrow$: Assume there is an $\alpha \in behs(A_1)$ such that $w = h_\alpha$ and $\alpha \notin behs(A_2)$. Using Theorem 10 we get there is a function $f \in [\![A_1]\!]$ such that $f(h_\alpha|_I) = h_\alpha|_O$. By definition of w we have that $w = h_\alpha$ and thus obtain by substitution that $f(w|_I) = w|_O$. Thus there is a function $f \in [\![A_1]\!]$ such that $f(w|_I) = w|_O$. It remains to show that $g(w|_I) \neq w|_O$ for all $g \in [\![A_2]\!]$. Towards a contradiction we assume that there is a function $g \in [\![A_2]\!]$ such that $g(w|_I) = w|_O$. By definition of $[\![\cdot]\!]$ we get that $\forall i \in I^\Omega : \exists \beta \in behs(A_2) : i = h_\beta|_I \wedge g(i) = h_\beta|_O$. Substituting $w|_I$ for i we obtain $\exists \beta \in behs(A_2) : w|_I = h_\beta|_I \wedge g(w|_I) = h_\beta|_O$. Since by assumption $w|_I = h_\alpha|_I$ and $g(w|_I) = w|_O$ by definition of g, this is equivalent to $\exists \beta \in behs(A_2) : h_\alpha|_I = h_\beta|_I \wedge w|_O = h_\beta|_O$. By assumption we have $w = h_\alpha$ and thus obtain via substitution $\exists \beta \in behs(A_2) : h_\alpha|_I = h_\beta|_I \wedge h_\alpha|_O = h_\beta|_O$, which is equivalent to $\exists \beta \in behs(A_2) : h_\alpha = h_\beta$. Using the definitions of h_α and h_β, this is equivalent to $\exists \beta \in behs(A_2) : \alpha = \beta$, which is equivalent to $\alpha \in behs(A_2)$ and contradicts the assumption. □

In the previous section, we presented a translation from finite TSCAs to BAs. Each word accepted by a BA resulting from such a translation corresponds to a behavior of the input TSCA. Existing algorithms for checking language inclusion and counterexample generation for BAs can thus be used for refinement checking and diff witness generation of architectures as defined above: Given two TSCAs A_1 and A_2 we use the translation defined in Section 5.2 to obtain two Büchi automata $ba(A_1)$ and $ba(A_2)$ such that $\mathcal{L}(ba(A_1)) = behs(A_1)$ and $\mathcal{L}(ba(A_2)) = behs(A_2)$. Using Theorem 16 and Theorem 13 we can transform a word accepted by $ba(A_1)$ that is not accepted by $ba(A_2)$ to a corresponding diff witness that semantically distinguishes the automata A_1 and A_2. If A_2 is I/O-deterministic, the BA $ba(A_2)$ is deterministic and weak and can thus be easily complemented in polynomial time in the size of $\mathcal{B}_2$, which is equal to the size of A_2. Then, inclusion checking is possible in polynomial time in the sizes of $ba(A_1)$ and $ba(A_2)$.

5.4. Mitigating the State Explosion Problem When Applying Semantic Differencing to System Architectures

This section summarizes practical performance improvements to mitigate a state explosion during semantic differencing of system architectures consisting of multiple TSCAs. We first define an abstract notion of *system architecture* (SA) inspired by [33]. While [33] considers a black-box view on SAs, in this paper we assume a white-box view where component implementations are available. A SA consists of an interface observable by the system's environment given by a channel signature

and of finitely many components represented by TSCAs that are connected via their channels.

Definition 23 (System Architecture)**.** *A system architecture is a tuple $S = (\Sigma, C)$ where:*

- *$\Sigma = (I, O)$ is a channel signature,*
- *C is a finite non-empty set of pairwise compatible components,*
- *the channels of S exist in the composition of the TSCAs' channel signatures, i.e., $I = J$ and $O \subseteq P$ where $(J, P) = \bigotimes_{c \in C} \Sigma_c$ denotes the composition of the channel signatures of all TSCAs in C, and*
- *$(\bigotimes C)\restriction O$ is a component.*

S is called finite iff Σ is finite and each $c \in C$ is finite.

The channel signature Σ defines the SA's external interface. The set C consists of the SA's components. The channels encoded by the channel signature Σ are required to exist in the compound resulting from composing the SA's components. The last condition stating that $(\bigotimes C)\restriction O$ must be a component is the most abstract well-formedness rule guaranteeing the result from composing the architecture's components is a component itself. More restricting well-formedness rules implying that $(\bigotimes C)\restriction O$ is a component are also possible to describe more restricted SA subclasses. One example is to require each component $c \in C$ to be strongly causal with respect to all its channels. Another, more relaxed, example is to require each component $c \in C$ to be composable with each possible intermediate composition result $\bigotimes D$ for each $D \subseteq C \setminus \{c\}$. We omit the proofs showing that these two examples imply that $(\bigotimes C)\restriction O$ is a component. Each individual TSCA participating in a SA is interpreted as an atomic component, *i.e.,* is not considered to have any subcomponents. As the TSCAs' channel signatures must be pairwise compatible, multiple components may read from the same channel whereas only one component is permitted to write on a channel. The input channels of a SA are equal to the input channels of the TSCA resulting from the subcomponents' composition. The set of output channels must be a subset of the output of the TSCA resulting from the composition. With this, output channels not specified by the architecture are hidden to the environment.

Example 25 (System architecture of the `Mod8Counter`)**.** *This example presents the system architecture of the alternative representation of the `Mod8Counter`, depicted in Figure 5, as composition of the TSCAs of its subcomponents `pos0`, `pos1`, and `pos2`. The system architecture is $S_{Mod8b} = (\Sigma, C)$ with*

- *the channel signature $\Sigma_{Mod8} = (\{inc, res\}, \{x_0, x_1, x_2\})$ and*
- *the set of components $C = \{TSCA_{pos0}, TSCA_{pos1}, TSCA_{pos2}\}$.*

The input channel set of S_{Mod8b} is equal to the input channel set of the composition of the three TSCAs. The output channel set of S_{Mod8b} is a subset of the output channel set of the composition of the TSCAs in C. The output channel set of the composition of the TSCAs in C is $\{x_0, q_0, x_1, q_1, x_2, q_2\}$. Channels included in the set of output channels of the composition that are no elements of the set of output channels of the system architecture S_{Mod8b} are hidden. The composition $\bigotimes C$ is a component, as shown in Example 17. Intuitively, the restriction of this composition to the output channels O is also a component, because the restriction of output channels does not influence the TSCA's reactiveness. The system architecture is finite, because all $c \in C$ are finite (cf. Example 10) and Σ_{Mod8} is finite.

A system architecture's TSCA semantics is the result from restricting the channels of the compound resulting from composing the SA's components to the channels specified by the SA's interface. The behavior and TSSPF semantics are given by the behavior and TSSPF semantics of the TSCA semantics.

Definition 24 (TSCA, Behavior, and TSSPF Semantics of SAs)**.** *Let $S = (\Sigma, C)$ with $\Sigma = (I, O)$ be a SA. The TSCA semantics of S is defined as $tspa(S) = (\bigotimes C)\restriction O$. The behavior semantics of S is defined as $behs(S) \stackrel{\text{def}}{=} behs(tspa(S))$. The TSSPF semantics of S is defined as $[\![tspa(S)]\!]$.*

Composing SAs with each other is also possible as the TSCA semantics of a SA can be interpreted as a component, again.

In continuous architecting and especially in combination with agile software development methodologies, requirements typically change during system development. In case additional requirements are added or existing requirements are strengthened, underspecification in component behavior models typically needs to be restricted to adapt the current specification or implementation to match the additional requirements. The behavior of the system under development is said to be refined.

Definition 25 (Refinement)**.** *A TSCA A is called (behavior) refinement of a TSCA B, denoted $A \preceq B$, iff $\Sigma_A = \Sigma_B$ and $behs(A) \subseteq behs(B)$.*

Refinement is lifted to SAs: A SA S is called *refinement* of a SA S', denoted $S \preceq S'$, iff $tspa(S) \preceq tspa(S')$. As a refinement exhibits less behaviors as the original system, there cannot exist a diff witness distinguishing the refined system from the original one.

Theorem 17. *Let A and B be two TSCAs. If $A \preceq B$, then $\Delta([\![A]\!], [\![B]\!]) = \emptyset$.*

Proof. Let A and B be two TSCAs such that $A \preceq B$. Thus, it holds that $behs(A) \subseteq behs(B)$. Suppose towards a contradiction there exists a diff witness $w \in \Delta([\![A]\!], [\![B]\!]) \neq \emptyset$. Using Theorem 16, this implies there exists $\alpha \in behs(A)$ such that $w = h_\alpha$ and $\alpha \notin behs(B)$. This contradicts $behs(A) \subseteq behs(B)$. □

Example 26 (Refinement of the `Mod8Counter` system architecture)**.** *Consider the system architectures of the `Mod8Counter` as depicted in Figure 3 (a) with the TSCA specified in Appendix B and the system architecture as depicted in Figure 5. In the following, we will refer to the first as the system architecture S and to the latter as the system architecture S'. First, we will investigate if $S' \preceq S$ by showing that*

$tspa(S') \preceq tspa(S)$. Therefore, it must hold that $\Sigma_{S'} = \Sigma_S$ and $behs(tspa(S')) \subseteq behs(tspa(S))$. The first is satisfied, because both system architectures have the same channel signature $\Sigma_{S'} = \Sigma_S = (\{inc, res\}, \{x_0, x_1, x_2\})$. Further, it holds that $tspa(S') = (\bigotimes C_{S'})\restriction O_{S'} = (TSCA_{pos0} \otimes TSCA_{pos1} \otimes TSCA_{pos2})\restriction O_{S'}$ and $tspa(S) = (\bigotimes C_S)\restriction O_S = TSCA_{Mod8a}$. The result of $TSCA_{pos0} \otimes TSCA_{pos1} \otimes TSCA_{pos1}$ has been explained in Example 25. Due to the channel restriction, we have $tspa(S') = tspa(S)$ and therefore, $behs(tspa(S')) = behs(tspa(S))$ holds.

Behavior refinement is reflexive and transitive. More importantly, it is compatible with composition:

Theorem 18. *Let A, B, and C be TSCAs such that A and C are compatible and B and C are compatible. If $A \preceq B$, then $A \otimes C \preceq B \otimes C$.*

Proof. Let A, B, and C be given as above such that $A \preceq B$. Let $\alpha \in behs(A \otimes C)$. Using Theorem 9, this implies $\alpha|_{C(\Sigma_A)} \in behs(A)$ and $\alpha|_{C(\Sigma_C)} \in behs(C)$. As $A \preceq B$, it holds that $behs(A) \subseteq behs(B)$. Thus, as $\alpha|_{C(\Sigma_A)} \in behs(A)$ and $behs(A) \subseteq behs(B)$, we obtain $\alpha|_{C(\Sigma_A)} \in behs(B)$. In summary, it holds that $\alpha|_{C(\Sigma_A)} \in behs(B)$ and $\alpha|_{C(\Sigma_C)} \in behs(C)$. Using Theorem 9, this implies $\alpha \in behs(B \otimes C)$. □

Refinement is also preserved by TSCA restriction.

Theorem 19. *Let A and B be TSCAs and let $O \subseteq O_B$. If $A \preceq B$, then $A\restriction O \preceq B\restriction O$.*

Proof. Let A and B be TSCAs and let $O \subseteq \Sigma_B$. Assume $A \preceq B$. By definition $A \preceq B$ it holds that $\Sigma_A = \Sigma_B$. Let $I \stackrel{\text{def}}{=} I_A = I_B$.

Let $A' \stackrel{\text{def}}{=} A\restriction O$ denote the restriction of A and let $B' \stackrel{\text{def}}{=} B\restriction O$ denote the restriction of B. As $\Sigma_A = \Sigma_B$, it especially holds that $\Sigma_{A'} = \Sigma_{B'}$. Let $\sigma = s_0, \theta_1, s_1, \theta_2, s_2, ... \in execs(A')$ be an execution of A. By definition of execution, it holds that $s_{j-1} \xrightarrow{\theta_j}_{\delta_{A'}} s_j$ for all $j > 0$. By definition of TSCA restriction, we have that $s_{j-1} \xrightarrow{\theta_j}_{\delta_{A'}} s_j$ is equivalent to $\exists (s^A_{j-1}, \theta^A_j, s^A_j) \in \delta_A : s^A_{j-1} = s_{j-1} \wedge s^A_j = s_j \wedge \theta^A_j|_{I \cup O} = \theta_j$ for each $j > 0$. Let such θ^A_j with $\theta^A_j|_{(I \cup O)} = \theta_j$ be given for each $j > 0$. As $s_{j-1} \xrightarrow{\theta^A_j}_{\delta_A} s_j$ for each $j > 0$, it holds by definition of execution that $\sigma_A \stackrel{\text{def}}{=} s_0, \theta^A_1, s_1, \theta^A_2, s_2, ... \in execs(A)$ is an execution of A. As $A \preceq B$, it holds that $beh(\sigma_A) \in behs(B)$. Therefore, there exists an execution $\sigma_B \in execs(B)$ of B such that $beh(\sigma_B) = beh(\sigma_A)$. Hence, there exist $s^B_0, s^B_1, s^B_2 ... \in S_B$ such that $\sigma_B = s^B_0, \theta^A_1, s^B_1, \theta^A_2, s^B_2, ... \in execs(B)$. This is by definition of execution equivalent to $(s^B_{j-1}, \theta^A_j, s^B_j) \in \delta_B$ for each $j > 0$. Using the TSCA restriction definition, this implies $(s^B_{j-1}, \theta^A_j|_{(I \cup O)}, s^B_j) \in \delta_{B'}$ for each $j > 0$. Thus, $\tau \stackrel{\text{def}}{=} s^B_0, \theta^A_1|_{(I \cup O)}, s^B_1, \theta^A_2|_{(I \cup O)}, s^B_2, ... \in execs(B')$ is an execution of B'. As by definition $\theta^A_j|_{(I \cup O)} = \theta_j$ for each $j > 0$, we obtain $beh(\tau) = \theta_0, \theta_1, \theta_2, ... \in behs(B')$. Observing that $\tau = \sigma$ and $beh(\tau) \in behs(B')$, we obtain $beh(\sigma) \in behs(B')$. We can conclude that for each execution $\sigma \in execs(A')$ there exists an execution $\tau \in execs(B')$ such that $beh(\sigma) = beh(\tau)$. Hence by definition of behaviors, $behs(A') \subseteq behs(B')$. □

Changing a SA to a successor version for adapting to evolved requirements often only requires to adapt the implementations of a proper subset of the SA's components without changing the architecture's topology, *i.e.,* the SA's interface is left unchanged and components neither need to be added nor removed but some component implementations are changed. In this case, it is often not strictly necessary to check whether the TSCA corresponding to the new SA is a refinement of the TSCA corresponding to the original architecture. It suffices to show that the composition of the evolved sub-architecture with any common subsystem of the original and the evolved SA is a refinement of the composition of the original sub-architecture with the same common subsystem:

Theorem 20. *Let $S_A = (\Sigma, C_A)$ and $S_B = (\Sigma, C_B)$ be two SAs having the same channel signature Σ. If there exists a set of components $Sub \subseteq (C_A \cap C_B)$ such that $\bigotimes((C_A \setminus C_B) \cup Sub) \preceq \bigotimes((C_B \setminus C_A) \cup Sub)$, then $S_A \preceq S_B$.*

Proof. Let $S_A = (\Sigma, C_A)$ and $S_B = (\Sigma, C_B)$ be two syntactically conform SAs with channel signature $\Sigma = (I, O)$. Suppose there exists a set of components $Sub \subseteq C_A \cap C_B$ such that $\bigotimes((C_A \setminus C_B) \cup Sub) \preceq ((C_B \setminus C_A) \cup Sub)$. Let $C = ((C_A \setminus C_B) \cup Sub)$ and let $C' = ((C_B \setminus C_A) \cup Sub)$.

In the following we show that $(\bigotimes C)$ and $(\bigotimes C_A \setminus C)$ as well as $\bigotimes C'$ and $\bigotimes C_B \setminus C'$ are compatible, which shows that the corresponding compositions are well-defined: As S_A is a SA, the components in C_A are all pairwise compatible. Thus, the components in $C \subseteq C_A$ and the components in $C_A \setminus C \subseteq C_A$ are also pairwise compatible. Therefore, $(\bigotimes C)$ and $(\bigotimes C_A \setminus C)$ are well-defined. As it holds that $C_A = C \cup (C_A \setminus C)$ and $C \cap (C_A \setminus C) = \emptyset$, applying the first part of Theorem 6 at most $|C|$ times, we obtain that $(\bigotimes C)$ and c are compatible for each $c \in C_A \setminus C$. As all components in C_A are pairwise compatible and each component $c \in C_A$ is compatible to $(\bigotimes C)$, applying the first part of Theorem 6 at most $|C_A \setminus C|$ times, we obtain that $(\bigotimes C)$ and $(\bigotimes C_A \setminus C)$ are compatible. A similar argument shows that $\bigotimes C'$ and $\bigotimes C_B \setminus C'$ are compatible.

In the following we show that $C_A \setminus C = C_B \setminus C'$, which enables to apply Theorem 18: It holds that $C_A \setminus C = C_A \setminus ((C_A \setminus C_B) \cup Sub) = (C_A \setminus (C_A \setminus C_B)) \setminus Sub = ((C_A \setminus C_A) \cup (C_A \cap C_B)) \setminus Sub = (C_A \cap C_B) \setminus Sub$. Using a similar argument, we obtain $C_B \setminus C' = C_B \setminus ((C_B \setminus C_A) \cup Sub) = (C_B \setminus (C_B \setminus C_A)) \setminus Sub = ((C_B \setminus C_B) \cup (C_B \cap C_A)) \setminus Sub = (C_B \cap C_A) \setminus Sub$. We can conclude $C_A \setminus C = C_B \setminus C'$.

Having shown the compatibility and $C_A \setminus C = C_B \setminus C'$ and since by assumption $\bigotimes C \preceq \bigotimes C'$, Theorem 18 guarantees $(\bigotimes C) \otimes (\bigotimes C_A \setminus C) \preceq (\bigotimes C') \otimes (\bigotimes C_B \setminus C')$. It holds that $C \cap (C_A \setminus C) = \emptyset = C' \cap (C_B \setminus C')$ and that all components in $C \cup (C_A \setminus C) = C_A$ and in $C' \cup (C_B \setminus C') = C_B$ are pairwise compatible. Thus, by definition of $\bigotimes$, the above is equivalent to $\bigotimes C_A \preceq \bigotimes C_B$. Since Theorem 19 guarantees that hiding preserves refinement, this implies $(\bigotimes C_A)|_O \preceq (\bigotimes C_B)|_O$. This is by definition of refinement equivalent to $S_A \preceq S_B$. □

Nevertheless, it might be the case that no such subsystem as described in Theorem 20 exists. Thus, in the worst case, the complete TSCAs for both architectures have to be considered.

However, we believe in practice this rarely occurs. The above leads to the following algorithm for mitigating the state explosion problem during semantic differencing of finite system architectures:

Algorithm 3 Mitigating the state explosion problem during refinement checking of system architectures.

Input: Two finite SAs $S_A = (\Sigma_A, C_A)$ and $S_B = (\Sigma_B, C_B)$.
Output: Yes, if $S_A \preceq S_B$, and $w \in \Delta([\![S_A]\!], [\![S_B]\!])$, otherwise.

```
define C = ⨂(C_A \ C_B) as TSCA
define C' = ⨂(C_B \ C_A) as TSCA
for all S ⊆ C_A ∩ C_B in increasing size do
  if behs(S ⊗ C) ⊆ behs(S ⊗ C') then
    return Yes            /* Composition without hiding */
  end if
end for
if behs(S_A) ⊆ behs(S_B) then
  return Yes              /* Composition with hiding */
else
  return w ∈ Δ([[S_A]], [[S_B]])
end if
```

In case the if-condition in the for-loop is satisfied, Theorem 20 guarantees the refinement relation holds. In case the condition is not satisfied for any $S \subseteq C_A \cap C_B$, it has to be checked whether the complete SA S_A refines the SA S_B. The difference between comparing $\bigotimes C_A$ with $\bigotimes C_B$ and $tspa(S_A)$ with $tspa(S_B)$ is that the former comparison does not consider hiding of internal channels, while the latter does. For the behavior inclusion checks and diff witness generation, existing algorithms for language inclusion checking between BAs may be used (cf. Section 5.1 and Section 5.3).

Example 27 (Application of Algorithm 3)**.** *Consider the system architectures of the* `Mod8Counter` *as depicted in Figure 3 (c) and the system architecture as depicted in Figure 5. We denote to the first one as S_A and to the second one as S_B. The goal is to determine whether $S_B \preceq S_A$ holds. Applying semantic differencing checking to these two system architectures reveals they refine each other. Both also refine the initial specification for the* `Mod8Counter` *as explained in Appendix B. More details on the evaluation regarding refinement checking between the three architectures are given in Section 6.3.*

6. Implementation and Evaluation

This section recapitulates the MontiArcAutomaton ADL [35, 37], presents the application of refinement checking to its models and evaluates our approach.

6.1. The MontiArcAutomaton ADL

The MontiArcAutomaton ADL [35, 37] comprises the modeling elements common to many popular component & connector ADLs [29], *i.e.,* hierarchical components with interfaces of typed, directed ports and unidirectional connectors (typed FIFO channels) exchanging messages between these ports. The components are black-boxes and either atomic or composed: atomic components yield behavior descriptions in form of embedded automata (following the I/O$^{\omega}$ [39] paradigm) or in form of Java implementations. Such automata and Java implementations are transformable to TSCAs for semantic differencing. The behavior of composed components solely emerges from the interaction of their subcomponents. Composing the TSCAs belonging to a composed component's subcomponent implementations results in a TSCA modeling the composed component's behavior. With this, semantic differencing of composed components is possible. Components are scheduled by a global clock and perform cycles of

- reading all messages on incoming ports;
- computing behavior (which might entail invoking subcomponents)
- producing a single message to each outgoing port.

Each computation consumes a time slice, *i.e.,* the output for messages received at the global clock's i-th tick is processed at its i+1-th tick earliest. All MontiArcAutomaton components are thereby strongly causal. The MontiArcAutomaton ADL also distinguishes between component types and their instances, supports component type inheritance, generic type parameters for components (*e.g.,* to be used with generic port types), and constructor-like configuration of these instances.

```
01 component Elevator {
02   port in Bool req1, in Bool at1,
03     // ... further ports ...
04     out Bool open, out Bool close,
05     out Clear clear;
06
07   component Control ctrl; // named
08   component Motor m;      // subcomponent
09   component Door d;       // instances
10
11   connect req1 -> control.req1;
12   // ... further connectors ...
13   connect control.clear -> clear;
14 }
```

Figure 14: Textual representation of the component `Elevator`.

The MontiArcAutomaton ADL is a textual modeling language implemented with the MontiCore [22] language workbench. The textual representation of the composed component type `Elevator` is illustrated in Figure 14. It begins with the keyword "component", followed by the component type's name and a body delimited by curly brackets (l. 1). The body contains an interface of typed ports (ll. 2-5), declares three subcomponents (ll. 7-9), and multiple connectors (ll. 11-13). The subcomponent declarations reference component types imported from artifacts (such as `Control`).

6.2. Semantic Differencing of MontiArcAutomaton Components

The implementation comprises a translation from MontiArcAutomaton architectures to semantically equivalent TSCAs.

TSCAs are only handled internally as representatives for sets of TSSPFs modeling component semantics and are not explicitly modeled by component developers. Each atomic component directly translates to a TSCA. The TSCA of a composed component is computed by composing the TSCAs of its subcomponents according to the architectural configuration defined by the composed component's connectors. A composed component's TSCA is either constructed using the composition operator's definition (cf Definition 16) or using Algorithm 2 to directly compute the trimmed TSCA of the compound. The implementation further consists of a translation from TSCAs to BAs and generators that produce models in the "BA format", which is the input format of the tool RABIT [3]. In case a BA does not refine another BA, RABIT provides a counterexample serving as a concrete disproof for refinement. The counterexamples are translated back to diff witnesses, which technically are finite prefixes of behaviors of one component that are no behaviors of another component. An engineer can use the witness to either manually inspect the component implementation for the syntactic reasons causing the semantic difference, or create a unit test where the component is provided the input encoded by the witness. When executing the unit test, the engineer may employ the usual debugging techniques provided by all common integrated development environments to identify the component implementation's elements causing the diff witness. Using the tool chain described above enables automated refinement checking and diff witness generation for MontiArcAutomaton architectures and ultimately supports engineers in detected the semantic differences between component implementations.

6.3. Semantic Differencing Evaluation

We evaluated the approach to semantic differencing with six MontiArcAutomaton architectures previously used for evaluation in [9, 38] and the modulo-8 counter architectures used as running example throughout this paper. We specifically chose the first six architectures for evaluation since the approach presented in [38] failed for some specifications, which we considered to be challenging, and to enable comparability. The architectures were slightly modified for this evaluation to resolve technical MontiArcAutomaton version compatibility issues. The example models as well as the BAs resulting from the translations are available online [1]. This paper extends the previous evaluation of [9] with the modulo-8 counter architecture that is used as running example. Further, the previous evaluation [9] always naively composes TSCAs using the definition of the composition operator (cf. Definition 16). This paper extends this evaluation by further applying the advanced composition method that simultaneous trims the compound while composing the composition's participants (cf. Algorithm 2). We reused the completion strategies [38] for completing the automata implementations of the architectures' atomic components.

The first architecture is given by an implementation of an elevator control system (ECS) (cf. Section 2). It comprises 3 composed and 5 atomic components. The second example consists of four variants of a mobile robot. We only report on the evaluation of the most challenging variant. This variant comprises 4 components in total whereof 3 components are atomic. Another architecture implements a pump station consisting of 3 composed and 10 atomic components. The modulo-8 counter specification is completely defined in Figure B.16. The result from executing the refinement checks presents in this paper slightly differ from the results presented in [9] because we repeated the evaluation of the pre-existing examples to enable comparability between the two different composition method variants. We conducted the evaluations of both composition variants on the same computer at the same date.

Table 1: Time for refinement checking and diff witness calculation.

		$\Delta([\![\cdot]\!], [\![\cdot]\!])$	$\Delta([\![\cdot]\!], \mathit{Chaos})$	$\Delta(\mathit{Chaos}, [\![\cdot]\!])$
Naive	Floors	62ms	536ms	885ms
	Elevator	83ms	2510ms	5927ms
	ECS	461ms	7124ms	15339ms
	SensorReading	62ms	753ms	1401ms
	Controller	12ms	17ms	19ms
	Pumpstation	120ms	321ms	570ms
	MobileRobot	61ms	67ms	85ms
	Mod8Counter	14ms	17ms	15ms
Trim	Floors	69ms	560ms	914ms
	Elevator	39ms	2525ms	5927ms
	ECS	94ms	9263ms	15850ms
	SensorReading	57ms	787ms	1390ms
	Controller	11ms	13ms	16ms
	Pumpstation	112ms	326ms	543ms
	MobileRobot	23ms	57ms	76ms

In [38], for each of the architectures, three specification checks are executed: it is checked whether the semantics of a component is equal to itself, whether a component refines a component with the same interfaces that implements arbitrary behavior, *i.e.,* all possible behaviors, and whether the semantics of a component are equal to the semantics of a component implementing arbitrary behavior. We performed the same checks on a computer with 3.0 GHz Intel Core i7 CPU, 16 GB Ram, Windows 10, and RABIT 2.4 using our translation from MontiArcAutomaton architectures to BAs and the language inclusion checking tool RABIT [3] (cf Section 6.2).

Table 1 summarizes the computation times of RABIT given the BAs resulting from the transformation as input. For the component ECS constructed using the naive composition method, for instance, checking whether it refines itself took 461ms, checking refinement with arbitrary behavior took 7124ms, and calculating a diff witness distinguishing the component from arbitrary behavior took 15339ms. Table 2 depicts the sizes of the automata resulting from the translations and the time required to construct a TSCA from its subcomponents' TSCAs using the denoted composition method. For component ECS, for instance, it took 3465ms to construct the TSCA using the naive composition method. The TSCA and the BA resulting from the transformation have 746 states and 98496 transitions. RABIT reported the tool has reduced the BA to 8 states and 1728 transitions after internal preprocessing. For every component we modeled arbitrary behavior (Chaos) with a

Table 2: The numbers of states and transitions of the TSCAs translated from the architectures and of the generated BAs.

			TSCA/BA		BA AP		Chaos
		time	#states	#trans.	#states	#trans.	#trans.
Naive	Floors	25ms	32	1024	32	1024	23328
	Elevator	460ms	34	10206	1	729	236196
	ECS	3465ms	746	98496	8	1728	472392
	SensorReading	7ms	2	1296	2	1296	69984
	Controller	1ms	1	9	1	9	108
	Pumpstation	19ms	6	3888	4	2592	17496
	MobileRobot	4ms	150	2700	12	216	1152
	Mod8Counter	0ms	8	32	8	32	32
Trim	Floors	267ms	32	1024	32	1024	23328
	Elevator	10ms	1	729	1	729	236196
	ECS	2829ms	8	1728	8	1728	472392
	SensorReading	118ms	2	1296	2	1296	69984
	Controller	1ms	1	9	1	9	108
	Pumpstation	3482ms	6	3888	4	2592	17496
	MobileRobot	10ms	12	216	12	216	1152

TSCA consisting of one state and a transition for every possible component input/output combination. The TSCA and the BA modeling arbitrary behavior for component ECS, for instance, comprise 472392 transitions (*cf.* Table 2). In contrast to the translation from MontiArcAutomaton architectures to the model checker Mona [38], our implementation succeeded for all example architectures. The longest computation time of our evaluation (15850ms, *cf.* Table 1) resulted from semantic differencing arbitrary behavior with the ECS component. We additionally used the implementation to automatically verify semantic equivalence of the three architectures depicted in Figure 3. We checked whether the specifications are semantically equivalent by checking refinement in both directions. Proving equivalence between the initial specification and the first structural refinement took 41ms. Checking equivalence between the initial specification and the second structural refinement took 47ms. The same check between the first and the second structural refinements was possible in 46ms.

The composition method that includes trimming the compounds yields a smaller composition duration in case the compound is smaller than the compound obtained from using the naive composition method (cf. Table 2). In case both composition methods yield the same compound, the naive composition method outperforms the method that includes trimming. This is plausible because of the overhead caused by trimming the TSCA. We conclude that our translation provides promising results. Nevertheless, the evaluation was only performed on a few specific architectures. Thus, the results are not generalizable to all possible architectures: the time needed by our tool may vary strongly from system to system.

7. Discussion

If the semantics domain of an ADL is overly general, undecidability of the underlying mathematical problems renders automated formal verification impossible. Then, architecture properties have to be proven manually, which is too expensive to be carried out in continuous architecture modeling and thus hinders employing agile development in architecture modeling projects: little changes to requirements or implementations can entail changing many manually performed proofs. In contrast, where automated formal verification is possible, sound and complete proofs can be generated automatically, supporting agile implementation evolution.

Focus is a comprehensive framework that supports specifying the observable input/output behavior of interactive systems. Its complexity requires carrying out proofs for system behavior verification manually. Focus provides various constructs for describing the semantics of distributed systems [36]. Examples are relations, set-based functions, sets of functions, assumption/guarantee predicates, or state-based representations. As identified in [36], the most fine-grained domain for describing the semantics of distributed systems using Focus are sets of SPFs. Independent of the style, specifications can describe timed or untimed behavior. Untimed behavior only considers the causality regarding the order of inputs and outputs. Timed specifications additionally concern causality regarding the passage of time. Many requirements are not only concerned with the order of messages but also state requirements with respect to passage of time. Thus, we employ a variant of the timed subset of Focus and thereby use sets of TSSPFs as semantics domain [36, 39].

Our approach is limited to systems where the data types' domains are finite and is restricted to the time-synchronous model of computation. However, our system model fits well into the kinds of systems developed for embedded systems such as automotive or robotics applications. Thus, our results enable fully automated tool support for many systems in such domains. Emphasizing that our approach cannot be generalized to the timed model of Focus as, for example, used in [16], is important: Timed SPFs (*cf.* [16, 36, 39]), for instance, are too general to be applicable to our approach. A timed SPF processes infinite sequences of finite sequences (of arbitrary lengths) of messages. Each of the finite sequences represents a finite stream of messages received or sent by a component in a single time unit. In contrast, TSSPFs only process single messages per time unit. The set of finite streams of messages over a non-empty finite data type is already infinite. Thus, for each time unit, a timed SPF needs to define a possible behavior for infinitely many tuples of input streams, whereas a TSSPF needs to define a reaction for all possible tuples of input messages, which are finitely many if the messages' data types are finite. From a practical viewpoint it is rarely required to specify the reaction in a time unit in response to the receipt of an arbitrary number of messages. Usually it either requires to handle single messages (TSSPFs) or sequences of messages where the length of the sequence is bounded by an arbitrary but fixed natural number. The latter can be reduced to the former by introducing lists of fixed length as message types.

The underlying theoretical problem for semantic differencing used in our approach is language inclusion checking between Büchi automata. Its complexity can be considered as another limitation of our approach. However, our main focus is not verifying a system's properties (*e.g.,* refinement or semantic differ-

encing) within seconds, which is most often already rendered impossible due to the complex nature of the safety critical system under development. We believe that nonetheless the possibility to apply formal fully automated verification (*e.g.,* over night) greatly facilitates continuous architecture modeling.

8. Related Work

Studies on the verification techniques of ADLs have been conducted, *e.g.,* in [43] and [45]. The study in [45] surveys verification techniques supported by ADLs with formal semantics, the translation of architectures to inputs for model checkers, and tool support as well as usability, scalability, and expressiveness. As supported by our approach, the study states that architecture verification for practical applications requires tool-support and automation. The study in [43] compares different verification tools and applies them to various ADLs. All architectures are transformed into intermediate labeled transition systems before the verification tools are applied, hampering the direct comparison with our approach.

The following surveys concrete approaches for formally analyzing hierarchical architecture descriptions. AutoFOCUS 3 [18] is a tool for the development of reactive embedded systems that also bases its semantics on FOCUS [7]. Although AutoFOCUS 3 supports model checking architectures against LTL and CTL formulas that specify properties concerning component behavior [10], we are not aware of a fully automated refinement checking method for AutoFOCUS 3. The π-ADL supports statistical model checking for verifying dynamic software architectures against DynBLTL properties [11]. To this effect, a statistical model of finite system executions is built and the probability of satisfying a property within a confidential bound is calculated. This approach is particularly tailored to dynamic architectures and is only concerned with finite traces. In contrast, our approach deals with infinite traces, static architectures, and full certainty. Refinement of architectures specified with timed I/O is described in [20]. Similar to behaviors of TSCAs, the semantics of a timed automaton is given by a set of traces. Refinement between timed I/O automata is defined similar as in our approach by trace inclusion. However, timed I/O automata are only marked with one message per transition and composition is defined differently. Further, the timing concept of I/O automata is more powerful and complicated than the one of our approach [16]. A game-based extension of the timed I/O automaton model enabling tool supported refinement checking has been proposed in [12]. Another approach to automated refinement checking based on the time-synchronous frame of FOCUS is described in [34, 38]. This approach is based on a relational semantics domain where the semantics of a component is given as a relation between the component's possible inputs and outputs. In contrast, our approach uses a more fine grained [36] semantics domain consisting of sets of functions. Refinement checking in [34, 38] relies on translating component semantics into WS1S and is implemented using the model checker Mona [13]. The approach suffers from the tool's high computational complexity, which is grounded in the non-elementary complexity of solving W1S1 problems. In contrast, we define a translation to Büchi automata and thereby obtain a PSPACE-complete complexity for refinement checking. While the relational approach is based on analyzing the result from composing the semantics of the individual components of a system, our approach first syntactically composes the individual components and bases analysis on the semantics of the compound.

9. Conclusion

We have presented an implementation of stepwise refinement for C&C ADLs using a subset of the Focus semantics for time-synchronous, distributed, interactive systems that is powerful enough to model complex and realistic systems. Based on previous work [9], we describe an approach to transform component models into time-synchronous channel automata that is based on an associative, commutative, and semantically compositional, syntactic composition operator for time-synchronous channel automata. Using this operator, the automata are composed syntactically and translated into Büchi automata, where their refinement can be checked through language inclusion. To this effect, we proved that the operational semantics of a finite time-synchronous channel automaton and the language accepted by the Büchi automaton resulting from the transformation coincide. This enables fully automated refinement checking for software architecture models in reasonable time.

We extended the previous approach [9] to improve its performance through technical enhancements of the underlying formal system model and extended previous evaluations. We further defined a notion of system architecture based on a white-box view where component implementations are assumed to be available. For such system architectures, we presented an algorithm leading to practical performance improvements for refinement checking.

This form of stepwise refinement supports continuous architecting through ensuring evolved components adhere to properties already proven for their predecessors. This ultimately reduces the effort for component evolution and, hence, facilitates continuous architecting.

References

[1] MontiArcAutomaton Models. `http://www.monticore.de/robotics/verification/`, [Online; accessed 2018-05-24].

[2] RABIT Tool Homepage, 2016. `http://www.languageinclusion.org/` [accessed 2016-12-31].

[3] Parosh Aziz Abdulla, Yu-Fang Chen, Lorenzo Clemente, Lukáš Holík, Chih-Duo Hong, Richard Mayr, and Tomáš Vojnar. Advanced Ramsey-Based Büchi Automata Inclusion Testing. In *International Conference on Concurrency Theory, CONCUR 2011*, 2011.

[4] Stephan Barth. Deciding Monadic Second Order Logic over ω-Words by Specialized Finite Automata. In *Integrated Formal Methods: 12th International Conference, IFM 2016, Reykjavik, Iceland, June 1-5, 2016, Proceedings*, 2016.

[5] Manfred Broy. A Logical Basis for Component-Oriented Software and Systems Engineering. *The Computer Journal*, 2010.

[6] Manfred Broy and Max Fuchs. The Design of Distributed Systems - An Introduction to FOCUS. Technical report, TU Munich, 1992.

[7] Manfred Broy and Ketil Stølen. *Specification and Development of Interactive Systems. Focus on Streams, Interfaces and Refinement*. Springer Verlag Heidelberg, 2001.

[8] Julius Richard Büchi. On a Decision Method in Restricted Second Order Arithmetic. In *Logic, Methodology and Philosophy of Science. Proceeding of the 1960 International Congress*. Stanford University Press, 1962.

[9] Arvid Butting, Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. Semantic Differencing for Message-Driven Component & Connector Architectures. In *International Conference on Software Architecture (ICSA'17)*, pages 145–154. IEEE, 2017.

[10] Alarico Campetelli, Florian Hölzl, and Philipp Neubeck. User-friendly Model Checking Integration in Model-based Development. In *International Conference on Computer Applications in Industry and Engineering*, 2011.

[11] Everton Cavalcante, Jean Quilbeuf, Louis-Marie Traonouez, Flávio Oquendo, Thaís Batista, and Axel Legay. Statistical Model Checking of Dynamic Software Architectures. In *European Conference on Software Architecture*, 2016.

[12] Alexandre David, Kim G. Larsen, Axel Legay, Ulrik Nyman, and Andrzej Wasowski. Timed I/O Automata: A Complete Specification Theory for Real-time Systems. In *ACM International Conference on Hybrid Systems: Computation and Control*, 2010.

[13] Jacob Elgaard, Nils Klarlund, and Anders Møller. MONA 1.x: New techniques for WS1S and WS2S. In *Computer-Aided Verification*, 1998.

[14] Robert France and Bernhard Rumpe. Model-Driven Development of Complex Software: A Research Roadmap. In *Future of Software Engineering 2007 at ICSE.*, 2007.

[15] Max Fuchs. Formal Design of a Modulo-N Counter. Technical Report TUM-I9512, Technische Univerität München, 1995.

[16] Radu Grosu and Bernhard Rumpe. Concurrent Timed Port Automata. Technical report, TU Munich, 1995.

[17] Radu Grosu, Ketil Stølen, and Manfred Broy. A Denotational Model for Mobile Point-to-Point Data-flow Networks with Channel Sharing, 1997.

[18] Florian Hölzl and Martin Feilkas. AutoFocus 3 - A Scientific Tool Prototype for Model-Based Development of Component-Based, Reactive, Distributed Systems. In *Model-Based Engineering of Embedded Real-Time Systems*, 2007.

[19] Bengt Jonsson. A Fully Abstract Trace Model for Dataflow and Asynchronous Networks. *Distributed Computing*, 1994.

[20] Dilsun K. Kaynar, Nancy A. Lynch, Roberto Segala, and Frits W. Vaandrager. Timed I/O Automata: A Mathematical Framework for Modeling and Analyzing Real-Time Systems. In *IEEE Real-Time Systems Symposium (RTSS 2003)*, 2003.

[21] Dexter Kozen. Lower Bounds for Natural Proof Systems. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, 1977.

[22] Holger Krahn, Bernhard Rumpe, and Steven Völkel. MontiCore: Modular Development of Textual Domain Specific Languages. In *Proceedings of Tools Europe*, 2008.

[23] Orna Kupferman and Moshe Y. Vardi. Verification of Fair Transition Systems. In *International Conference on Computer Aided Verification*, 1996.

[24] Orna Kupferman and Moshe Y. Vardi. Complementation Constructions for Nondeterministic Automata on Infinite Words. In *Tools and Algorithms for the Construction and Analysis of Systems: 11th International Conference, TACAS 2005*, 2005.

[25] Robert P. Kurshan. Complementing Deterministic Büchi Automata in Polynomial Time. *Journal of Computer and System Sciences*, 1987.

[26] Edward A Lee. CPS Foundations. In *Proceedings of the 47th Design Automation Conference*, pages 737–742. ACM, 2010.

[27] Christof Löding. Efficient minimization of deterministic weak ω-automata. *Information Processing Letters*, 2001.

[28] Zohar Manna and Amir Pnueli. Verifying Hybrid Systems. In *Hybrid Systems*, pages 4–35. Springer, 1993.

[29] Nenad Medvidovic and Richard N. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Transactions on Software Engineering*, 2000.

[30] Peter Naur and Brian Randell, editors. *Software Engineering: Report of a conference sponsored by the NATO Science Committee, Garmisch, Germany, 7-11 Oct. 1968, Brussels, Scientific Affairs Division, NATO*, 1969.

[31] Object Management Group. MDA Guide Version 1.0.1, June 2003. `http://www.omg.org/news/meetings/workshops/UML_2003_Manual/00-2_MDA_Guide_v1.0.1.pdf` [Online; accessed 2015-12-17].

[32] Object Management Group. OMG Unified Modeling Language (OMG UML), Superstructure Version 2.3 (10-05-05), May 2010. `http://www.omg.org/spec/UML/2.3/Superstructure/PDF/` [accessed 2017-01-13].

[33] Jan Philipps and Bernhard Rumpe. Refinement of Information Flow Architectures. In *Proceedings of the 1st International Conference on Formal Engineering Methods (ICFEM'97)*. IEEE Computer Society, 1997.

[34] Jan Oliver Ringert. *Analysis and Synthesis of Interactive Component and Connector Systems*. Shaker Verlag, 2014.

[35] Jan Oliver Ringert, Alexander Roth, Bernhard Rumpe, and Andreas Wortmann. Language and Code Generator Composition for Model-Driven Engineering of Robotics Component & Connector Systems. *Journal of Software Engineering for Robotics (JOSER)*, 2015.

[36] Jan Oliver Ringert and Bernhard Rumpe. A Little Synopsis on Streams, Stream Processing Functions, and State-Based Stream Processing. *International Journal of Software and Informatics*, 2011.

[37] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. *Architecture and Behavior Modeling of Cyber-Physical Systems with MontiArcAutomaton*. Shaker Verlag, 2014.

[38] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. Model-Based Specification of Component Behavior with Controlled Underspecification. In *Modellbasierte Entwicklung eingebetteter Systeme (MBEES'16)*, 2016.

[39] Bernhard Rumpe. *Formale Methodik des Entwurfs verteilter objektorientierter Systeme*. Doktorarbeit, TU Munich, 1996.

[40] S. Safra. On the complexity of omega -automata. In *Proceedings of the 29th Annual Symposium on Foundations of Computer Science*, 1988.

[41] Sven Schewe. Minimisation of Deterministic Parity and Buchi Automata and Relative Minimisation of Deterministic Finite Automata. *Computing Research Repository - CORR*, 2010.

[42] Frank Strobl and Alexander Wisspeintner. Specification of an Elevator Control System. Technical report, TU Munich, 1999.

[43] Jeffrey J.P. Tsai and Kuang Xu. A comparative study of formal verification techniques for software architecture specifications. *Annals of Software Engineering*, 2000.

[44] Markus Völter, Thomas Stahl, Jorn Bettin, Arno Haase, Simon Helsen, and Krzysztof Czarnecki. *Model-Driven Software Development: Technology, Engineering, Management*. Wiley, 2013.

[45] Pengcheng Zhang, Henry Muccini, and Bixin Li. A Classification and Comparison of Model Checking Software Architecture Techniques. *Journal of Systems and Software*, 2010.

Appendix A. `Mod8Counter` component in FOCUS

In MontiArcAutomaton, there is an explicit language construct (the connector) to indicate that two ports are connected. Besides this, MontiArcAutomaton distinguishes component types and component instances. Therefore, MontiArcAutomaton obtains unique port names by the fully qualified name of component instance and the port name. On the contrary, FOCUS has no notion of component type and has no explicit construct to indicate connectors. With this, MontiArcAutomaton is better suited for praxis, whereas FOCUS abstracts from implementation details to avoid notational clutter and improve formal representation. Thus, a MontiArcAutomaton architecture is conceptually transformed to a FOCUS architecture by omitting component types and by renaming ports such that they have identical names iff they are connected. A transformed version of the component `mod8Counter` as depicted in Figure 3 is depicted in Figure A.15.

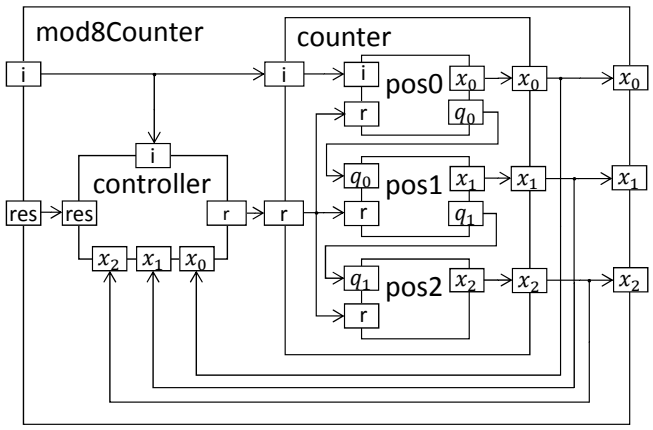

Figure A.15: FOCUS architecture of the `mod8Counter`.

Appendix B. TSCA of the `Mod8Counter` component

This section explains the TSCA of the initial specification of the `Mod8Counter` component as presented in Figure 3 (a). Figure B.16 demonstrates the TSCA in its graphical representation, where abbreviations for states and transitions are used. Transitions that increase the counted value start with the letter *i*, those that reset the value start with *r*, and those that do not alter the counted value start with an *n*. The textual representation of the TSCA and the abbreviations are explained in the following.

The TSCA depicted in Figure B.16 is a tuple $TSCA_{Mod8a} = (\Sigma, X, S, \iota, \delta)$, where

- $\Sigma = (\{\text{res,inc}\}, \{x_0,x_1,x_2\})$,
- the internal channels are $X = \{lv\}$ with $type(lv) = \{0, .., 7\}$,
- the set of states is defined by the set of all functions $S = X^{\rightarrow} = \{\theta \in [\{lv\} \rightarrow M] \mid \theta(lv) \in \mathbb{N} \wedge 0 \leq \theta(lv) \leq 7\}$, where for notational simplicity, we denote by $s_i = \{lv \mapsto i\}$,
- the initial state is $\iota = \{s_0\}$,
- the transition relation $\delta = I \cup R \cup N$ comprises the sets of increasing transitions $I = \bigcup_{k=0,..,8} i_k$, resetting transitions $R = \bigcup_{k=0,..,16} r_k$, and state conserving transitions $N = \bigcup_{k=0,..,8} n_k$, where
  - $i_0 = \{(s_0, \theta, s_1) \mid \theta(\text{res}) = \varepsilon \wedge \theta(\text{inc}) = \top \wedge \theta(x_0) = \top \wedge \theta(x_1) = \varepsilon \wedge \theta(x_2) = \varepsilon\}$
  - $i_1 = \{(s_1, \theta, s_2) \mid \theta(\text{res}) = \varepsilon \wedge \theta(\text{inc}) = \top \wedge \theta(x_0) = \varepsilon \wedge \theta(x_1) = \top \wedge \theta(x_2) = \varepsilon\}$
  - $i_2 = \{(s_2, \theta, s_3) \mid \theta(\text{res}) = \varepsilon \wedge \theta(\text{inc}) = \top \wedge \theta(x_0) = \top \wedge \theta(x_1) = \top \wedge \theta(x_2) = \varepsilon\}$
  - $i_3 = \{(s_3, \theta, s_4) \mid \theta(\text{res}) = \varepsilon \wedge \theta(\text{inc}) = \top \wedge \theta(x_0) = \varepsilon \wedge \theta(x_1) = \varepsilon \wedge \theta(x_2) = \top\}$
  - $i_4 = \{(s_4, \theta, s_5) \mid \theta(\text{res}) = \varepsilon \wedge \theta(\text{inc}) = \top \wedge \theta(x_0) = \top \wedge \theta(x_1) = \varepsilon \wedge \theta(x_2) = \top\}$
  - $i_5 = \{(s_5, \theta, s_6) \mid \theta(\text{res}) = \varepsilon \wedge \theta(\text{inc}) = \top \wedge \theta(x_0) = \varepsilon \wedge \theta(x_1) = \top \wedge \theta(x_2) = \top\}$
  - $i_6 = \{(s_6, \theta, s_7) \mid \theta(\text{res}) = \varepsilon \wedge \theta(\text{inc}) = \top \wedge \theta(x_0) = \top \wedge \theta(x_1) = \top \wedge \theta(x_2) = \top\}$
  - $i_7 = \{(s_7, \theta, s_0) \mid \theta(\text{res}) = \varepsilon \wedge \theta(\text{inc}) = \top \wedge \theta(x_0) = \varepsilon \wedge \theta(x_1) = \varepsilon \wedge \theta(x_2) = \varepsilon\}$
  - $r_0 = \{(s_0, \theta, s_0) \mid \theta(\text{res}) = \top \wedge \theta(\text{inc}) = \varepsilon \wedge \theta(x_0) = \varepsilon \wedge \theta(x_1) = \varepsilon \wedge \theta(x_2) = \varepsilon\}$
  - $r_1 = \{(s_1, \theta, s_0) \mid \theta(\text{res}) = \top \wedge \theta(\text{inc}) = \varepsilon \wedge \theta(x_0) = \varepsilon \wedge \theta(x_1) = \varepsilon \wedge \theta(x_2) = \varepsilon\}$
  - $r_2 = \{(s_2, \theta, s_0) \mid \theta(\text{res}) = \top \wedge \theta(\text{inc}) = \varepsilon \wedge \theta(x_0) = \varepsilon \wedge \theta(x_1) = \varepsilon \wedge \theta(x_2) = \varepsilon\}$
  - $r_3 = \{(s_3, \theta, s_0) \mid \theta(\text{res}) = \top \wedge \theta(\text{inc}) = \varepsilon \wedge \theta(x_0) = \varepsilon \wedge \theta(x_1) = \varepsilon \wedge \theta(x_2) = \varepsilon\}$
  - $r_4 = \{(s_4, \theta, s_0) \mid \theta(\text{res}) = \top \wedge \theta(\text{inc}) = \varepsilon \wedge \theta(x_0) = \varepsilon \wedge \theta(x_1) = \varepsilon \wedge \theta(x_2) = \varepsilon\}$
  - $r_5 = \{(s_5, \theta, s_0) \mid \theta(\text{res}) = \top \wedge \theta(\text{inc}) = \varepsilon \wedge \theta(x_0) = \varepsilon \wedge \theta(x_1) = \varepsilon \wedge \theta(x_2) = \varepsilon\}$
  - $r_6 = \{(s_6, \theta, s_0) \mid \theta(\text{res}) = \top \wedge \theta(\text{inc}) = \varepsilon \wedge \theta(x_0) = \varepsilon \wedge \theta(x_1) = \varepsilon \wedge \theta(x_2) = \varepsilon\}$
  - $r_7 = \{(s_7, \theta, s_0) \mid \theta(\text{res}) = \top \wedge \theta(\text{inc}) = \varepsilon \wedge \theta(x_0) = \varepsilon \wedge \theta(x_1) = \varepsilon \wedge \theta(x_2) = \varepsilon\}$
  - $r_8 = \{(s_0, \theta, s_0) \mid \theta(\text{res}) = \top \wedge \theta(\text{inc}) = \top \wedge \theta(x_0) = \varepsilon \wedge \theta(x_1) = \varepsilon \wedge \theta(x_2) = \varepsilon\}$
  - $r_9 = \{(s_1, \theta, s_0) \mid \theta(\text{res}) = \top \wedge \theta(\text{inc}) = \top \wedge \theta(x_0) = \varepsilon \wedge \theta(x_1) = \varepsilon \wedge \theta(x_2) = \varepsilon\}$
  - $r_{10} = \{(s_2, \theta, s_0) \mid \theta(\text{res}) = \top \wedge \theta(\text{inc}) = \top \wedge \theta(x_0) = \varepsilon \wedge \theta(x_1) = \varepsilon \wedge \theta(x_2) = \varepsilon\}$
  - $r_{11} = \{(s_3, \theta, s_0) \mid \theta(\text{res}) = \top \wedge \theta(\text{inc}) = \top \wedge \theta(x_0) = \varepsilon \wedge \theta(x_1) = \varepsilon \wedge \theta(x_2) = \varepsilon\}$
  - $r_{12} = \{(s_4, \theta, s_0) \mid \theta(\text{res}) = \top \wedge \theta(\text{inc}) = \top \wedge \theta(x_0) = \varepsilon \wedge \theta(x_1) = \varepsilon \wedge \theta(x_2) = \varepsilon\}$
  - $r_{13} = \{(s_5, \theta, s_0) \mid \theta(\text{res}) = \top \wedge \theta(\text{inc}) = \top \wedge \theta(x_0) = \varepsilon \wedge \theta(x_1) = \varepsilon \wedge \theta(x_2) = \varepsilon\}$

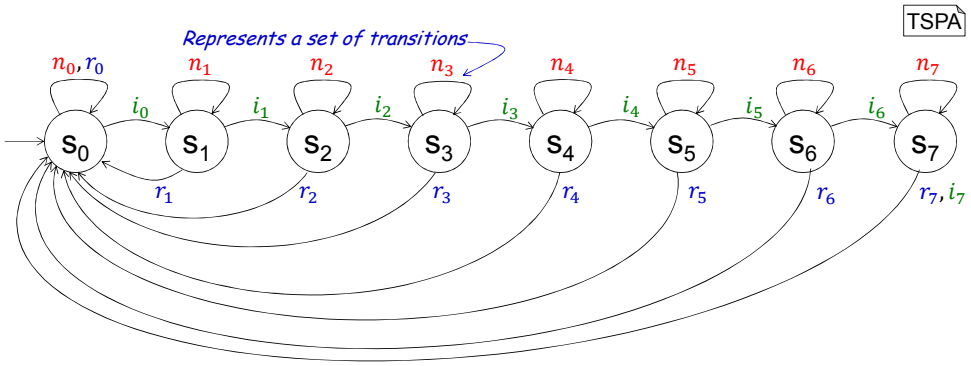

Figure B.16: TSCA of a modulo 8 counter.

- $r_{14} = \{(s_6, \theta, s_0) \mid \theta(\text{res}) = \top \wedge \theta(\text{inc}) = \top \wedge \theta(x_0) = \varepsilon \wedge \theta(x_1) = \varepsilon \wedge \theta(x_2) = \varepsilon\}$
- $r_{15} = \{(s_7, \theta, s_0) \mid \theta(\text{res}) = \top \wedge \theta(\text{inc}) = \top \wedge \theta(x_0) = \varepsilon \wedge \theta(x_1) = \varepsilon \wedge \theta(x_2) = \varepsilon\}$
- $n_0 = \{(s_0, \theta, s_0) \mid \theta(\text{res}) = \varepsilon \wedge \theta(\text{inc}) = \varepsilon \wedge \theta(x_0) = \varepsilon \wedge \theta(x_1) = \varepsilon \wedge \theta(x_2) = \varepsilon\}$
- $n_1 = \{(s_1, \theta, s_1) \mid \theta(\text{res}) = \varepsilon \wedge \theta(\text{inc}) = \varepsilon \wedge \theta(x_0) = \top \wedge \theta(x_1) = \varepsilon \wedge \theta(x_2) = \varepsilon\}$
- $n_2 = \{(s_2, \theta, s_2) \mid \theta(\text{res}) = \varepsilon \wedge \theta(\text{inc}) = \varepsilon \wedge \theta(x_0) = \varepsilon \wedge \theta(x_1) = \top \wedge \theta(x_2) = \varepsilon\}$
- $n_3 = \{(s_3, \theta, s_3) \mid \theta(\text{res}) = \varepsilon \wedge \theta(\text{inc}) = \varepsilon \wedge \theta(x_0) = \top \wedge \theta(x_1) = \top \wedge \theta(x_2) = \varepsilon\}$
- $n_4 = \{(s_4, \theta, s_4) \mid \theta(\text{res}) = \varepsilon \wedge \theta(\text{inc}) = \varepsilon \wedge \theta(x_0) = \varepsilon \wedge \theta(x_1) = \varepsilon \wedge \theta(x_2) = \top\}$
- $n_5 = \{(s_5, \theta, s_5) \mid \theta(\text{res}) = \varepsilon \wedge \theta(\text{inc}) = \varepsilon \wedge \theta(x_0) = \top \wedge \theta(x_1) = \varepsilon \wedge \theta(x_2) = \top\}$
- $n_6 = \{(s_6, \theta, s_6) \mid \theta(\text{res}) = \varepsilon \wedge \theta(\text{inc}) = \varepsilon \wedge \theta(x_0) = \varepsilon \wedge \theta(x_1) = \top \wedge \theta(x_2) = \top\}$
- $n_7 = \{(s_7, \theta, s_7) \mid \theta(\text{res}) = \varepsilon \wedge \theta(\text{inc}) = \varepsilon \wedge \theta(x_0) = \top \wedge \theta(x_1) = \top \wedge \theta(x_2) = \top\}$