Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 260 (2010) 125–153
www.elsevier.com/locate/entcs

On the Observable Behaviour of Composite Components[1]

Rolf Hennicker, Stephan Janisch

Institut für Informatik
Ludwig-Maximilians-Universität München
{hennicker, janisch}@pst.ifi.lmu.de

Alexander Knapp

Institut für Informatik
Universität Augsburg
knapp@informatik.uni-augsburg.de

Abstract

The crucial strength of the component paradigm lies in the possibility to encapsulate behaviours. In this work, we focus on the observable behaviour of composite components which encapsulate the behaviour of (possibly large) assemblies of connected subcomponents. We first present our general component model which is equipped with a precise formal semantics allowing us to distinguish systematically different kinds of behaviours for ports, for components, and for component assemblies; technically we use UML2 notation for describing component structures and I/O-transition systems for behaviours. Then we investigate an efficient method for the computation of the observable behaviour of composite components which can circumvent the possibly infeasible intermediate computation of the usually complex behaviour of underlying assemblies if there are *behaviourally neutral* subcomponents. Finally, we utilise the fact that components are connected via ports such that checks for behavioural neutrality of components can be reduced to checks for behavioural neutrality of connected ports in the case of *weakly deterministic* port behaviours.

Keywords: hierarchical components, observable behaviour, I/O-transition system

1 Introduction

Components, being based on the notion of strong encapsulation, foster a compositional approach to system construction and analysis. Port-based component models, as e.g. the ROOM model [30], stress this encapsulation aspect by taking ports with provided and required interfaces to form the explicit and exclusive communication windows of components. The behavioural properties of a component should thus

[1] This research has been partially supported by the GLOWA-Danube project (01LW0602A2) sponsored by the German Federal Ministry of Education and Research.

doi:10.1016/j.entcs.2009.12.035

be governed only by the observable behaviour of a component on its boundary. In particular, when composing components hierarchically all subcomponent implementation details and internal interactions become hidden behind the facade of the super-component.

Being aware of the various software component models [28,21] that have been proposed already, the first goal of this study is to combine many of the ideas found in the literature to obtain a comprehensive model for component-based system design which satisfies the following requirements. The component model should (i) support as a concrete syntax UML2 notation, which in our view provides, as a de-facto standard for industrial applications, a powerful notation for modelling the structural aspects of components; (ii) support ports and, of course, hierarchical composition; (iii) be equipped with a precise formal semantics for both, the structural and the behavioural aspects of components thus resolving ambiguities left in the UML and also in other notations; and (iv) be able to discriminate systematically between different kinds of behaviours, like port, component and assembly behaviour, and also between specified and derived behaviours.

Building upon [4,19] our component model will first be described by an appropriate metamodel and then precisely defined in terms of an algebraic approach. We distinguish between simple (i.e., basic) components, component assemblies, i.e., network structures of components connected via their ports, and composite components which make up large scale components by encapsulating assemblies. For the formal representation of behaviours we use I/O-transition systems which are based on interface automata [12]. Explicit I/O-labellings allow to distinguish between input, output and internal actions which can be hidden to compute (derive) the observable behaviour of components. The behaviour of an assembly is also an I/O-transition system which is derived from the composition of the observable behaviours of the connected components of the assembly.

The second goal of this study is to provide support for an efficient construction and behavioural analysis of large scale component systems. In the context of component-based systems this usually means construction and analysis of assembly behaviour or, in other words, facing the state explosion problem, which has been the source of investigation at many places. There are, for instance, approaches to compositional reachability analysis (CRA) such as [10] and, even more prominent under the umbrella of component-based systems, approaches which focus on efficient analysis of safety and liveness properties such as [5,9,16,18,17]. However, the mentioned studies focus essentially on the analysis of assembly behaviour, whereas our focus is on the construction of the *observable* behaviour of a composite component which encapsulates an assembly. Given a composite component CC that encapsulates an assembly a with behaviour $beh(a)$, the observable behaviour $obs(CC)$ of CC is formally defined by observational abstraction from $beh(a)$. The idea is then to identify components within the assembly a which do not play any role for the observable behaviour of CC. Formally, this idea will be reflected by our notion of *behavioural neutrality*. Then, instead of computing the behaviour $beh(a)$ of the complete assembly it is sufficient to compute the behaviour of a *simpler* as-

sembly, say s, where all behaviourally neutral components have been removed from a, such that the observational abstractions from $beh(s)$ and $beh(a)$ are observationally equivalent. As a consequence, $obs(CC)$ is then observationally equivalent to the observational abstraction of $beh(s)$ which may be computable much easier. A major result of this paper is an algorithm which shows how to reduce an assembly for the efficient computation of the observable behaviour of a composite component. Our reduction strategy is particularly useful for acyclic topologies containing a distinguished "rooted" component, but is also sound for arbitrary topologies.

The cost of the reduction algorithm depends primary on the cost of the neutrality checks which are performed between connected components of the assembly and hence depends on the complexity of the observable behaviours of the subcomponents. But we can reduce the cost of neutrality checks between components if we utilise the fact that components are connected via ports, that ports have themselves, usually much simpler, behavioural protocols and that component behaviours should conform to the behaviour of their ports. We show, as a further important result, that it is sufficient to perform neutrality checks for the behaviours of connected ports provided that ports to be checked for neutrality have a *weakly deterministic* behavioural protocol.

The remainder of this paper is structured as follows: Section 2 summarises definitions, operators and facts for I/O-transition systems. In Sect. 3 we review our component model by means of a metamodel and an example and complement the description by a precise algebraic definition. We consider behaviours of ports and of simple components and we describe how assembly behaviours and observable behaviours of components can be formally derived from their parts. In Sect. 4 we focus on our main results concerning simplified computation of the observable behaviour of composite components and we discuss the important role of ports for efficient neutrality checks. Section 5 provides a detailed discussion on related work and, finally, Sect. 6 summarises our approach and outlines future work.

2 I/O-Transition Systems

We use I/O-transition systems to describe behaviours of ports, components, and assemblies with their provided (input) and required (output) operations as well as their internal actions. Our definition of an I/O-transition system follows the notion of interface automata of de Alfaro and Henzinger [12]. However, we include an *invisible* (or *silent*) action τ for abstracting from internal actions and studying the observable behaviour of components which was not an issue in [12]. Moreover, we do not require input-determinism.

An *I/O-labelling*, *iol* for short, $L = (I, O, T)$ consists of three mutually disjoint sets of *input* labels I, *output* labels O, and *internal* labels T; we write $\bigcup L$ for the set of labels $I \cup O \cup T$. An *I/O-transition system*, *iots* for short, $A = (L, Q, q_0, \Delta)$ is given by an iol L, a set of *states* Q, an *initial state* $q_0 \in Q$ and a *transition relation* $\Delta \subseteq Q \times (\bigcup L \cup \{\tau\}) \times Q$ (with $\tau \notin \bigcup L$). We write $L(A)$ for the iol of A, and $\mathscr{L}(A)$ for $\bigcup L(A)$.

2.1 Observational Equivalence

Behaviours of components are compared using observational equivalence of iotss. Observational equivalence is based on weak bisimulations abstracting from invisible τ-actions.

For an iots $A = (L, Q, q_0, \Delta)$ we define the *τ-closure* of Δ as $\hat{\Delta} \subseteq Q \times (\bigcup L \cup \{\tau\}) \times Q$ as follows: For an $l \in \bigcup L$, $(q, l, q') \in \hat{\Delta}$, if q' is reachable from q w.r.t. Δ by a sequence of transitions containing exactly one transition labelled with l and arbitrarily many τ-transitions; i.e., there are $q = q_1, \ldots, q_m \in Q$ and $q'_1, \ldots, q'_n = q' \in Q$ such that $(q_i, \tau, q_{i+1}) \in \Delta$ for $1 \leq i < m$, $(q_m, l, q'_1) \in \Delta$, and $(q'_j, \tau, q'_{j+1}) \in \Delta$ for $1 \leq j < n$. Furthermore, $(q, \tau, q') \in \hat{\Delta}$, if $q = q'$ or q' is reachable from q w.r.t. Δ by τ-transitions only; i.e., there are $q = q_1, \ldots, q_n = q' \in Q$ such that $(q_i, \tau, q_{i+1}) \in \Delta$ for $1 \leq i < n$.

A *weak bisimulation* between two iotss $A = (L, Q_A, q_{0,A}, \Delta_A)$ and $B = (L, Q_B, q_{0,B}, \Delta_B)$ with the same iol L is a relation $R \subseteq Q_A \times Q_B$ such that for all $(q_A, q_B) \in R$ and all $a \in \bigcup L \cup \{\tau\}$ the following holds:

(i) $\forall q'_A \in Q_A \,.\, (q_A, a, q'_A) \in \Delta_A \supset \exists q'_B \in Q_B \,.\, (q_B, a, q'_B) \in \hat{\Delta}_B \wedge (q'_A, q'_B) \in R$,

(ii) $\forall q'_B \in Q_B \,.\, (q_B, a, q'_B) \in \Delta_B \supset \exists q'_A \in Q_A \,.\, (q_A, a, q'_A) \in \hat{\Delta}_A \wedge (q'_A, q'_B) \in R$.

The iotss A and B are *observationally equivalent*, denoted by $A \approx B$, if there exists a weak bisimulation R between A and B with $(q_{0,A}, q_{0,B}) \in R$.

2.2 Operators

For deriving behaviours in our component framework we will use the following operators on iotss: hiding, relabelling and the formation of products. Hiding and relabelling on iotss are generalisations of the usual operators used in various process algebras (see, e.g., [26,23]), the product of iotss is defined in accordance with the product of interface automata [12].

Hiding. Hiding is used to turn a subset of the labels of an iots into the invisible action τ. Formally, the *hiding* of an iol $L = (I, O, T)$ w.r.t. a subset $H \subseteq \bigcup L$ is the iol $L/H = (I \setminus H, O \setminus H, T \setminus H)$. The *hiding* of an iots $A = (L, Q, q_0, \Delta)$ w.r.t. a label set $H \subseteq \mathscr{L}(A)$ is the iots $A/H = (L/H, Q, q_0, \Delta/H)$ where $\Delta/H = \{(q, \tau, q') \mid (q, a, q') \in \Delta \wedge a \in H\} \cup \{(q, a, q') \mid (q, a, q') \in \Delta \wedge a \notin H\}$.

In many cases we will choose $H = T$, i.e., we will hide all internal labels. Then, for an iol $L = (I, O, T)$, we write $L\xi$ for L/T and, for an iots A over L, we write $A\xi$ for A/T.

Relabelling. A relabelling is used for renaming labels and for changing the kind of labels. Formally, a *relabelling* $\rho : L \to L'$ from an iol $L = (I, O, T)$ to an iol $L' = (I', O', T')$ is defined by a function from $\bigcup L$ to $\bigcup L'$ for which we also write ρ. The *relabelling* of an iots $A = (L, Q, q_0, \Delta)$ w.r.t. a relabelling $\rho : L \to L'$ is the iots $A\rho = (L', Q, q_0, \Delta\rho)$ where $\Delta\rho = \{(q, \rho(l), q') \mid (q, l, q') \in \Delta \wedge l \in \bigcup L\} \cup \{(q, \tau, q') \mid (q, \tau, q') \in \Delta\}$.

A particular application of relabelling is the internalisation of (some) input and

output labels. For an iol $L = (I, O, T)$ and a subset $X \subseteq I \cup O$, we define the *internalisation relabelling* $\theta_X : (I, O, T) \to (I \setminus X, O \setminus X, T \cup X)$ with $\theta_X(l) = l$ for $l \in \bigcup L$.

Given two relabellings ρ_1 and ρ_2 which coincide on all elements in the intersection of their domains, we denote their union (as function graphs) by $\rho_1 \cup \rho_2$.

Product. The formation of the product of two iotss expresses their parallel composition with synchronisation on identical input and output labels. To construct the product the iolss of the given iotss must be composable. Two iolss $L_1 = (I_1, O_1, T_1)$ and $L_2 = (I_2, O_2, T_2)$ are *composable* if $I_1 \cap I_2 = \emptyset$, $O_1 \cap O_2 = \emptyset$, $T_1 \cap (I_2 \cup O_2 \cup T_2) = \emptyset$, and $T_2 \cap (I_1 \cup O_1 \cup T_1) = \emptyset$. The *shared* labels of composable iolss L_1 and L_2, written $L_1 \bowtie L_2$, are given by $(I_1 \cap O_2) \cup (O_1 \cap I_2)$. The *product* of two composable iolss L_1 and L_2 is the iol $L_1 \otimes L_2 = ((I_1 \cup I_2) \setminus (L_1 \bowtie L_2), (O_1 \cup O_2) \setminus (L_1 \bowtie L_2), T_1 \cup T_2 \cup (L_1 \bowtie L_2))$ which moves the shared labels to the internal labels. Two iotss A_1 and A_2 are *composable* if $L(A_1)$ and $L(A_2)$ are composable. The *product* of two composable iotss $A_1 = (L_1, Q_1, q_{0,1}, \Delta_1)$ and $A_2 = (L_2, Q_2, q_{0,2}, \Delta_2)$ is the iots $A_1 \otimes A_2 = (L_1 \otimes L_2, Q_1 \times Q_2, (q_{0,1}, q_{0,2}), \Delta)$ where

$$\begin{aligned}\Delta = {} & \{((q_1, q_2), a, (q_1', q_2)) \mid (q_1, a, q_1') \in \Delta_1 \wedge q_2 \in Q_2 \wedge a \notin L_1 \bowtie L_2\} \cup \\ & \{((q_1, q_2), a, (q_1, q_2')) \mid (q_2, a, q_2') \in \Delta_2 \wedge q_1 \in Q_1 \wedge a \notin L_1 \bowtie L_2\} \cup \\ & \{((q_1, q_2), a, (q_1', q_2')) \mid (q_1, a, q_1') \in \Delta_1 \wedge (q_2, a, q_2') \in \Delta_2 \wedge a \in L_1 \bowtie L_2\} \ .\end{aligned}$$

Observational equivalence is a congruence w.r.t. hidings, relabellings and products. For hiding and relabelling this follows directly from the definitions and corresponds to the respective facts for standard process algebras, like [23], since observational equivalence is independent of the kind of the labels. The preservation of observational equivalence by the product follows from the facts that observational equivalence is a congruence w.r.t. parallel composition with synchronisation of shared labels [23] and that the equality of labellings is preserved by the product.

Lemma 2.1 *Let A, B, and C be iotss.*

(i) *If $A \approx B$ and $H \subseteq \mathscr{L}(A)$, then $A/H \approx B/H$.*

(ii) *If $A \approx B$ and $\rho : L(A) \to L'$ is a relabelling, then $A\rho \approx B\rho$.*

(iii) *If $A \approx B$ and A and C are composable, then $A \otimes C \approx B \otimes C$.*

The product is commutative and associative up to observational equivalence. Commutativity is straightforward from the definitions; associativity carries over from the corresponding fact for interface automata [12] taking into account also τ-transitions. Furthermore, hiding of non-shared labels commutes with the formation of products.

Lemma 2.2 *Let A, B, and C be iotss.*

(i) *If A and B are composable, then $A \otimes B \approx B \otimes A$.*

(ii) *If A, B, and C are pairwise composable, then $(A \otimes B) \otimes C \approx A \otimes (B \otimes C)$.*

(iii) *If A and B are composable with shared labels $S = L(A) \bowtie L(B)$, and $H \subseteq$*

$\mathscr{L}(A)$ *with* $H \cap S = \emptyset$, *then* $(A/H) \otimes B \approx (A \otimes B)/H$.

For a finite index set I, we write $\bigotimes_{i \in I} A_i$ for the product of the iotss A_i with $i \in I$, which is justified by Lem. 2.2(ii).

3 Component Model

Building upon ideas from ROOM [30], we consider components to be strongly encapsulated behaviours. Encapsulation is achieved by ports which regulate any interaction of components with their environment. Components can be hierarchically structured containing again an assembly of components and connectors. We first give an overview of our component model by means of a (semi-formal) metamodel and illustrate its application by an example. We then complement the discussion with a formal algebraic component model which forms the basis of the subsequent analyses.

3.1 Metamodel

Figure 1 shows the metamodel of our component model which is an extension of the model in [19] taking more specifically into account different kinds of behaviours and connectors. The model is similar to UML2's view on components [27]. Its embedding into the UML2 is sketched in [19] and carries over to the extended model straightforwardly.

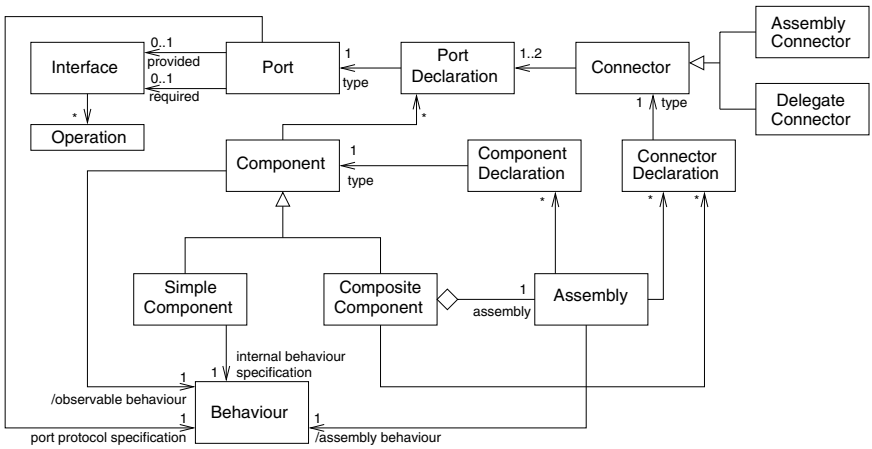


Figure 1. Component metamodel

A *port* describes a view on a component (like particular functionality or communication), the operations offered and needed in the context of this view and the mandatory sequencing of operation calls from the outside and from the inside. The operations offered by a port are summarised in its *provided interface*; the operations needed in its *required interface*. The sequencing of operations being called from and on a port is described in a *port protocol specification*. To be precise, a port is in fact considered as a port type that can be used in local port declarations of a component.

There are two kinds of components, *simple components* and *composite components* which are abstracted in the metaclass *component*. Any kind of component has a set of *port declarations*, which introduce local port names with corresponding port types, and an associated *observable behaviour*. In our metamodel a component represents in fact a component type that can be used in component declarations when building component assemblies. Each component should be correct with respect to its ports, i.e., the protocol of its ports should indeed be supported by the observable behaviour of the component. For each simple component an *internal behaviour specification* is given. A composite component encapsulates an assembly of components and declares delegate connectors. An *assembly* defines the internal structure of the composite component in terms of a set of local component declarations and local (binary) *assembly connector declarations* that connect local components via their ports. Non-connected (open) ports of local components may be connected to so-called *relay ports* of the surrounding composite component, using *delegate connector declarations*. Relay ports are not explicitly distinguished in the metamodel. Also an assembly has an associated behaviour. As indicated by the slash symbol in Fig. 1 the observable behaviour of a component as well as the assembly behaviour are derived behaviours. The observable behaviour of simple components is derived from the components' given internal behaviour specification; for composite components the observable behaviour is derived from the behaviour of its assembly which in turn is derived from the observable behaviours of the local components within the assembly and their connections.

3.2 Example: A Compressing Proxy System

We illustrate our component model by the compressing proxy system also used by Bernardo et al. [5]. In contrast to [5] we use here an additional component for the compression of graphical data which will be useful for our discussions later on: An HTTP proxy server mediates connections between a web server and their clients. In order to increase network bandwidth, the proxy server may apply different compression techniques depending on the kind of information transferred. The proxy server distinguishes between textual (`txt`) and graphical (`gif`) data and applies different compression tools before sending the data further downstream.

We use UML2 notation for concretely specifying the static structure of component systems. The behaviours will semantically be based on I/O-transition systems (iotss, see Sect. 2). We do not intend to propose a particular, e.g., process algebraic or state machine, syntax for describing behaviours. In fact, any concrete syntax could be used as long as an interpretation by iotss is provided. Here, we use the LTSA tool [23] for the graphical representation of iotss. For the transitions in the LTSA graphs we indicate that a label i is an input (provided) label by the visual representation $i\_$ and, symmetrically, that a label o is an output (required) label by $\_o$. An initial state is indicated by 0. Several transitions between two states will be shown as a single transition with a set of labels. [2]

[2] The LTSA tool supports the process algebra FSP [23]. Hence, to obtain the graphs, we had to define the behaviours used in the example by appropriate FSP terms. These terms are not shown here because they

Static Structure

We describe the static structure of the compressing proxy system by a composite component type CompressingProxy which consists of an assembly of three simple components with types Adaptor, GZip, and GifToJpg introduced by the respective component declarations adapt : Adaptor, gzip : GZip, and gifToJpg : GifToJpg in Fig. 2 [3]. The adaptor is responsible for distinguishing between textual and graphical data and forwarding the textual data for compression to gzip, which employs the gzip utility, and the graphical data to gifToJpg converting GIF- into JPG-images.

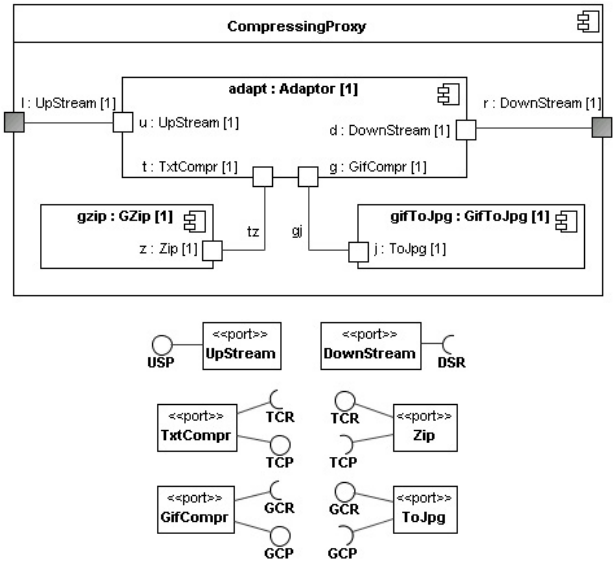


Figure 2. Static structure of a compression proxy server

The simple components as well as the composite component show port declarations (like t : TxtCompr or l : UpStream). The port declarations of the composite component are called relay ports. Port declarations are interconnected by assembly connectors inside the assembly (like the connector tz between t : TxtCompr and z : Zip) [4] and delegate connectors to the outside (like the anonymous connector between u : UpStream and l : UpStream).

The provided and required interfaces of the port types of the compressing proxy system are depicted with the UML ball-and-socket notation on the right-hand side of Fig. 2. The operations of these interfaces are as follows: openTxt, openGif (USP); compressed (DSR); txt, endTxt (TCR); bufFul, zip, endZip (TCP); gif (GCR); and jpg (GCP). We do not consider operations with parameters here. If two ports are connected by an assembly connector, the provided interface of the one port has to be equal to the required interface of the other, and vice versa. [5]

---

are irrelevant for our study.

[3] The UML2 declarations in Fig. 2 also show multiplicities, indicating how many instances of a component or port may exist. However, we only specified singletons (multiplicity 1) leaving the discussion of arbitrary multiplicities to future work.

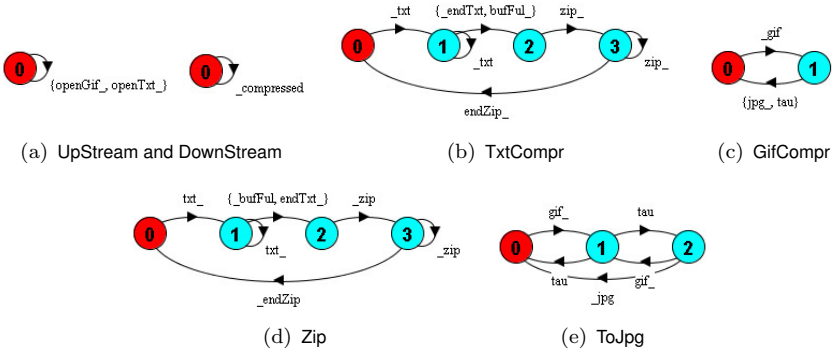
[4] UML2 would allow for arbitrary n-ary connectors with $n > 2$ which we do not consider here.

[5] In general, one could use a more flexible condition such that the required interface of one port is included

Behaviour

Based on the static structure of the compressing proxy system the informal description of its intended behaviour reads as follows: A proxy of type CompressingProxy receives stream-based data on its port l which is delegated to the port u of the contained component adapt. The adaptor distinguishes textual and graphical data received at u and forwards textual data for compression via port t and graphical via port g. After receiving the compression result, adapt sends the data further downstream using d which is relayed to r.

Specified Behaviours for Ports and Simple Components. A port protocol regulates the communication sequences according to the particular view of the port. The port protocols are specified by the component designer as iotss; the port protocols for the compressing proxy system are depicted in Fig. 3. They use the operations of the provided interface of the port as inputs and the operations of its required interface as outputs. The transitions of the protocol may show the invisible action τ (like in GifCompr or ToJpg) to reflect a possible internal choice of the port's owner component; but they do not show internal actions.



(a) UpStream and DownStream (b) TxtCompr (c) GifCompr

(d) Zip (e) ToJpg

Figure 3. Protocols for the ports in Fig. 2

In particular, the protocol of the port (type) TxtCompr used in the component (type) Adaptor, see Fig. 3(b), has as inputs bufFul, zip, and endZip; and as outputs txt and endTxt. After having sent an initial part of textual data by txt, there is a choice between sending further text pieces by txt until the end of the text is reached (notified by endTxt), or receiving the message bufFul which indicates buffer overflow of the compression tool. In the first case, the port waits to receive zip data (zip) until transmission end is signalled by endZip. In the second case, the port must retrieve the already compressed data via zip and communication proceeds until eventually the end of the text will be reached. In both cases a new communication can be started by sending again an initial part of the next textual data to be compressed via txt. As an example for a protocol with τ-transitions, consider Fig. 3(c) for GifCompr: After sending gif the port can receive the compressed image by jpg but it can also perform an internal choice not to wait for the result of the previous request.

in the provided interface of the other one. However, it is technically more convenient to use the more restrictive condition from above.

The internal behaviour of simple components, which are basic building blocks, has also to be specified by the component designer as an iots. Since a component can only communicate with its environment via its ports, any message received or sent has now the form $p.m$ where p is a port name and m is a message according to the provided or required interface of the port. In contrast to port behaviours, we also consider internal actions of components represented by internal transition labels.

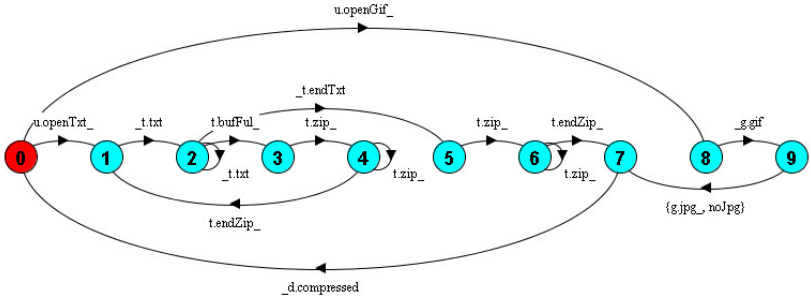


Figure 4. Behaviour of Adaptor [6]

For instance, the internal behaviour of the simple component (type) Adaptor is specified in Fig. 4. The messages received are u.openTxt, u.openGif, t.zip, t.bufFul, t.endZip, and g.jpg; the messages sent are d.compressed, t.txt, t.endTxt, and g.gif; and the only internal action is noJpg. The behaviour specification introduces an internal transition $(9, \mathsf{noJpg}, 7)$, reifying the τ-transition of the port protocol in Fig. 3(c). The transition's trigger is not further specified; one may think that the component waits only a fixed amount of time for the compression of `gif` data, and if the compression takes too long, it sends the `gif` data further downstream. We do not show specifications of the internal behaviour of the simple components GZip and GifToJpg. Since these components are equipped with one port only, their behaviour specifications would not differ too much from their port protocol specifications (Fig. 3(d), 3(e)), but, for of GifToJpg, the internal behaviour specification would reveal the internal decisions indicated by the τ-transitions in the protocol of ToJpg.

Derived Behaviours for Components, Assemblies and Composite Components. When using a component, be it simple or composite, in a component system, the designer of the system is not interested in the internal behaviour of the component, but only in the behaviour which is observable from the outside of the component. Thus, each component is equipped with an observable behaviour. For a simple component this observable behaviour can be computed from the internal behaviour by hiding all internal messages. In the case of Adaptor the single internal action noJpg is turned into τ.

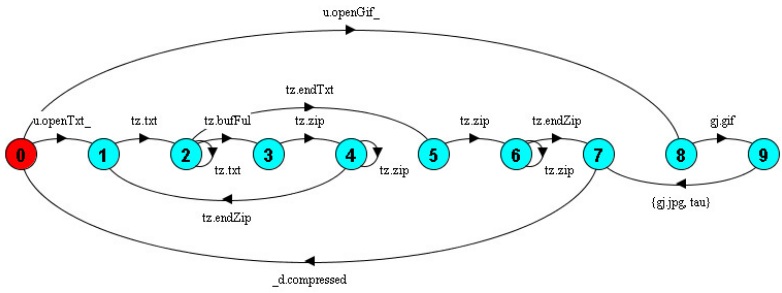
Components are used in assemblies via component declarations showing a name and a component type. In order to distinguish the different components inside an

[6] The behaviour corresponds exactly to the adaptor behaviour in [5] if we remove the communication concerning the port `g` for compression of graphical data, that is, if we remove the states `8` and `9` and the respective transitions.

assembly, the labels of their observable behaviours are prefixed by the name of the component declaration, thus turning, e.g., u.openGif of Adaptor into adapt.u.openGif.

The overall behaviour of an assembly shows both, the interaction behaviour of the used components that are connected within the assembly via their ports, and the communication potential of the open ports of components that are not connected within the assembly. Hence, an assembly behaviour is a derived behaviour, given by the composition of the observable behaviours of all declared components of the assembly, synchronised according to the given (binary) assembly connectors. Note that we use the *observable* behaviour of components when constructing an assembly behaviour since in this way internal component behaviours are abstracted away before the composition. This is particularly useful when constructing hierarchical components which encapsulate an assembly and which can again be used as parts of other assemblies on the next hierarchy level. The synchronisation within an assembly maps the labels of the ports connected by an assembly connector into one common name, thus making them shared. Open ports do not appear in assembly connectors, therefore those parts of component behaviours related to open ports remain unsynchronised. In general, we also consider the case where a port is neither open nor connected to another port. Technically, these ports are closed by applying unary connectors. The internal actions of an assembly behaviour are the synchronised transitions of the composition, representing communication of the components via their connected ports and the not further used actions on closed ports. The behaviour on the remaining open ports is described by their input and output actions which are the only external actions of an assembly.

For instance the behaviour of the assembly underlying the CompressingProxy component is computed by the synchronisation of the observable behaviours of the three components adapt : Adaptor, gzip : Gzip, and gifToJpg : GifToJpg according to the connectors of their ports. The assembly behaviour has 30 states and 65 transitions (including τ-transitions). After minimisation we obtain the iots shown in Fig. 5(a). The synchronisation labels are internal actions; e.g., the output label adapt.g.gif of adapt : Adaptor and the input label gifToJpg.j.gif of gifToJpg : GifToJpg are synchronised in accordance with the connector gj by the internal label gj.gif.



(a) Assembly behaviour of CompressingProxy

(b) Observable behaviour of CompressingProxy

Figure 5. Assembly behaviour and observable behaviour of the compressing proxy system

Finally, the observable behaviour of composite components is derived using

an external view of its underlying assembly where the synchronised communications between the components connected within the assembly are abstracted into τ-transitions. Moreover, the input and output actions of a composite component are delegated between its relay ports and the open ports of the assembly. Figure 5(b) shows the resulting observable behaviour of CompressingProxy after minimisation w.r.t. observational equivalence.

Correctness of Components

Each component should be correct with respect to its ports, i.e., the views of its ports should indeed be backed by the observable behaviour of the component. Component correctness can be checked for a component C and one of its port declarations $p : P$ as follows: First, all labels of the component's observable behaviour which are not of the form $p.m$, that is, do not pertain to the port p, are abstracted into τ, i.e., we construct the projection of the observable behaviour of C on the port p. Second, all input and output labels of the protocol of P are prefixed by p. Then C is correct w.r.t. its port $p : P$ if the projected component behaviour and the prefixed iots of the protocol are observationally equivalent. C is correct, if it is correct w.r.t. all of its ports.

For instance, component Adaptor is correct w.r.t. its port t : TxtCompr: We replace all labels in Fig. 4 not prefixed by t by τ and prefix all labels of Fig. 3(b) by t. Then it can be checked (using, e.g., the minimisation functionality of the LTSA tool) that these iotss are observationally equivalent. In fact, all components in the example, including CompressingProxy itself, are correct. The correctness of the latter can be easily checked by comparing the respective projections of Fig. 5(b) with the iotss in Fig. 3(a).

3.3 Algebraic Component Model

We complement the metamodel presentation of our component model with an algebraic description, which defines formally all previously mentioned concepts and behaviours in terms of algebraic structures and iotss resp.; see Sect. 2. In particular, we distinguish between those behaviours which have to be provided by the component developer and those that are computed (derived), by rendering the latter as definitions. We use *italics* for denoting all kinds of derived operators.

For the computation of the different behaviours we employ several relabelling functions on iolss and iotss, which specialise the relabelling as introduced in Sect. 2.2. We assume a primitive domain Nm of *names*.

A prefix relabelling prefixes all labels in an iots by a given name. For an iol $L = (I, O, T)$ and some name n, we define the iol $n.L = (n.I, n.O, n.T)$ where $n.I = \{n.i \mid i \in I\}$ and similarly for $n.O$ and $n.T$. The *prefix relabelling* $\rho_n : L \to n.L$ is defined by $\rho_n(l) = n.l$ for $l \in \bigcup L$. Given an iots A and a name n, we write $n.A$ for the iots $A\rho_n$.

A match relabelling maps differently prefixed input or output labels into a single common name. For an iol $L = (I, O, T)$, $X \subseteq$ Nm and $y \in$ Nm, we define the iol

$L\mu_{(X,y)} = (I\mu_{(X,y)}, O\mu_{(X,y)}, T)$ where $I\mu_{(X,y)} = \{y.l \mid \exists x \in X\,.\,x.l \in I\} \cup \{l \mid l \in I \wedge \forall x \in X\,.\,l \neq x.l'\}$ and similarly for $O\mu_{(X,y)}$. The *match relabelling* $\mu_{(X,y)} : L \to L\mu_{(X,y)}$ is defined by $\mu_{(X,y)}(x.l) = y.l$ if $x \in X$ and $x.l \in I \cup O$, and $\mu_{(X,y)}(l') = l'$ otherwise.

A *binary synchronisation relabelling* $\sigma_{(X,y)}$ is the match relabelling $\mu_{(X,y)}$ with $|X| = 2$. A *unary synchronisation relabelling* $\sigma_{(x,y)}$ is the composition $\theta_W \circ \mu_{(\{x\},y)}$ of a match relabelling and an internalisation with $W = \{y.l \mid y.l \in I\mu_{(\{x\},y)} \cup O\mu_{(\{x\},y)}\}$. A *relay relabelling* $\rho_{(x,y)}$ is the match relabelling $\mu_{(\{x\},y)}$.

Ports. We assume a domain of messages Msg, interfaces If and ports (more precisely, port types) Port, together with functions msg : If $\to \wp$Msg to return the messages constructed from the operations of an interface, and prv : Port $\to$ If and req : Port $\to$ If for the provided and required interfaces of a port. For a port P we write $msg(P)$ for $\mathrm{msg}(\mathrm{prv}(P)) \cup \mathrm{msg}(\mathrm{req}(P))$. We also assume a domain of port declarations PortDcl with a function nm : PortDcl $\to$ Nm for the name and a function ty : PortDcl $\to$ Port for the port (type); we write $p : P$ for a port declaration d with $\mathrm{nm}(d) = p$ and $\mathrm{ty}(d) = P$.

For each port $P \in$ Port we assume given its *protocol (specification)*, written $\mathrm{prot}(P)$, which is an I/O-transition system $((I, O, T), Q, q_0, \Delta)$ with $I = \mathrm{msg}(\mathrm{prv}(P)), O = \mathrm{msg}(\mathrm{req}(P))$ and $T = \emptyset$.

Definition 3.1 *The* protocol *of a port declaration $p : P$ is given by $prot(p : P) = p.\mathrm{prot}(P)$.*

Components. We assume a domain Cmp of components (more precisely, component types) and a function ports : Cmp $\to \wp$PortDcl returning the ports declared for a component. For a component C and port declaration $p : P$ we write $C[p : P]$ if $p : P \in \mathrm{ports}(C)$. Like for ports, we assume a domain of component declarations CmpDcl with a function nm : CmpDcl $\to$ Nm for the name and a function ty : CmpDcl $\to$ Cmp for the component (type); we write $c : C$ for a component declaration d with $\mathrm{nm}(d) = c$ and $\mathrm{ty}(d) = C$. The ports of a component declaration are given by $ports(c : C) = \{c.p : P \mid p : P \in \mathrm{ports}(C)\}$.

For each component $C \in$ Cmp there is a derived *observable behaviour*, written $obs(C)$, which is an iots $((I, O, T), Q, q_0, \Delta)$ with $I = \bigcup\{p.\mathrm{msg}(\mathrm{prv}(P)) \mid p : P \in \mathrm{ports}(C)\}$, $O = \bigcup\{p.\mathrm{msg}(\mathrm{req}(P)) \mid p : P \in \mathrm{ports}(C)\}$ and $T = \emptyset$. How the observable behaviour of a component is defined depends on whether the component is simple or composite; cf. Def. 3.6 and Def. 3.7 below.

Definition 3.2 *The* observable behaviour *of a component declaration $c : C$ is given by $obs(c : C) = c.obs(C)$.*

Definition 3.3 *The* behaviour *of a component C* observable at *port $p : P \in \mathrm{ports}(C)$ is given by $obs_{p:P}(C) = obs(C)/H$ with $H = \mathscr{L}(obs(C)) \setminus \{p.m \mid m \in msg(P)\}$.*

Definition 3.4 *A component C is* correct w.r.t. *a port declaration $p : P \in \mathrm{ports}(C)$, if $obs_{p:P}(C) \approx prot(p : P)$. A component C is* correct, *if it is correct w.r.t. all $p : P \in \mathrm{ports}(C)$.*

Connectors. We assume a domain Conn of connectors (more precisely, connector types) with a function ports : Conn $\to \wp$PortDcl yielding the connected port declarations such that $1 \leq |\text{ports}(K)| \leq 2$ for each $K \in \text{Conn}$. We assume a domain of connector declarations ConnDcl with a function nm : ConnDcl $\to$ Nm for the name and ty : ConnDcl $\to$ Conn for the connector (type); we write $k : K$ for a connector declaration d with $\text{nm}(d) = k$ and $\text{ty}(d) = K$. Moreover, if $\text{ports}(K) = \{p : P, q : Q\}$ we write $k : (p : P, q : Q)$ and if $\text{ports}(K) = \{p : P\}$ we write $k : (p : P)$. In the latter case, k is a unary connector which will be technically needed in our reduction algorithm later on.

The domain Conn has two disjoint sub-domains AsmConn $\subseteq$ Conn, DlgConn $\subseteq$ Conn of assembly and delegate connectors, respectively. Delegate connectors are always binary, i.e., for each $K \in \text{DlgConn}$ we have $|\text{ports}(K)| = 2$. For an assembly connector with port declarations $\{p_1 : P_1, p_2 : P_2\}$ the required interface $\text{req}(P_1)$ has to be equal to the provided interface $\text{prv}(P_2)$ and vice versa. For a delegate connector the provided and required interfaces of its port declarations must coincide.

Assemblies. We assume a domain Asm with functions cmps : Asm $\to \wp$CmpDcl returning an assembly's declared components and conns : Asm $\to \wp$ConnDcl yielding its declared connectors. An assembly $a \in \text{Asm}$ has to be well-formed: (i) it shows only assembly connectors, i.e., if $k : K \in \text{conns}(a)$, then $K \in \text{AsmConn}$; (ii) only ports of components inside a are connected, i.e., for all $k : K \in \text{conns}(a)$ we have that $\text{ports}(K) \subseteq \bigcup\{\mathit{ports}(c : C) \mid c : C \in \text{cmps}(a)\}$; and (iii) there is at most one connector for each port, i.e., if $c.p : P \in \bigcup\{\mathit{ports}(c : C) \mid c : C \in \text{cmps}(a)\}$ and $k : K, k' : K' \in \text{conns}(a)$ with $c.p : P \in \text{ports}(K) \cap \text{ports}(K')$, then $k : K = k' : K'$.

For an assembly a we define $\mathit{cmp} : \bigcup\{\mathit{ports}(c : C) \mid c : C \in \text{cmps}(a)\} \to \text{cmps}(a)$ by $\mathit{cmp}(c.p : P) = c : C$ if $c.p : P \in \mathit{ports}(c : C)$. The components of an assembly a may show *open* ports which are not connected and we let $\mathit{open}(a) = \bigcup\{\mathit{ports}(c : C) \mid c : C \in \text{cmps}(a)\} \setminus \bigcup\{\text{ports}(K) \mid k : K \in \text{conns}(a)\}$.

Definition 3.5 *The* behaviour *of an assembly a is given by*

$$\mathit{beh}(a) = \bigotimes_{c:C \in \text{cmps}(a)} (\mathit{obs}(c : C))\sigma \ ,$$

where $\sigma = \bigcup\{\sigma_{(\{c.p,d.q\},k)} \mid \exists k : K \in \text{conns}(a) \,.\, \text{ports}(K) = \{c.p : P, d.q : Q\}\} \cup \bigcup\{\sigma_{(c.p,k)} \mid \exists k : K \in \text{conns}(a) \,.\, \text{ports}(K) = \{c.p : P\}\}$.

We write $\langle \mathcal{C}; \mathcal{K} \rangle$ for an assembly a with the set of component declarations $\text{cmps}(a) = \mathcal{C}$ and the set of connector declarations $\text{conns}(a) = \mathcal{K}$. We abbreviate the synchronisations used in the computation of the assembly behaviour in Def. 3.5 by $\sigma_{\mathcal{K}}$.

Simple Components. We assume a sub-domain SCmp $\subseteq$ Cmp of simple components.

For each $SC \in \text{SCmp}$ we assume given its *behaviour (specification)*, written $\text{beh}(SC)$, which is an iots $((I, O, T), Q, q_0, \Delta)$ where $I = \{p.\text{msg}(\text{prv}(P)) \mid p : P \in \text{ports}(SC)\}$ and $O = \{p.\text{msg}(\text{req}(P)) \mid p : P \in \text{ports}(SC)\}$.

Definition 3.6 *The* observable behaviour *of a simple component SC is given by* $\mathit{obs}(SC) = \text{beh}(SC)\xi$.

Recall that the operator ξ hides the internal labels of an iots, thus $obs(SC)$ shows the external view of the behaviour of C.

Composite Components. We assume a sub-domain CCmp $\subseteq$ Cmp of composite components, disjoint to SCmp. We assume functions asm : CCmp $\to$ Asm returning the assembly of a composite component, and conns : CCmp $\to \wp$ConnDcl returning the connectors declared in a composite component. Similar to assemblies we require a composite component CC to be well-formed: (i) it shows only delegate connectors, i.e., if $k : K \in \text{conns}(CC)$, then $K \in$ DlgConn; (ii) all open ports of the $\text{asm}(CC)$ are connected, i.e., for all $c.p : P \in \mathit{open}(\text{asm}(CC))$ there is $k : K \in \text{conns}(CC)$ such that $c.p : P \in \text{ports}(K)$; and (iii) all relay ports are connected, i.e., for all $r : R \in \text{ports}(CC)$ there is a unique $k : K \in \text{conns}(CC)$ with $\text{ports}(K) = \{c.p : P, r : R\}$ and $c.p : P \in \mathit{open}(\text{asm}(CC))$.

Definition 3.7 *The* observable behaviour *of a composite component CC is given by*

$$obs(CC) = (beh(\text{asm}(CC))\xi)\rho \ ,$$

where $\rho = \bigcup\{\rho_{(c.p,r)} \mid \exists k : K \in \text{conns}(CC) \,.\, \text{ports}(K) = \{c.p : P, r : R\}\}$.

We write $\langle a; \mathcal{P}; \mathcal{K}\rangle$ for a composite component CC with assembly $\text{asm}(CC) = a$, set of (relay) port declarations $\text{ports}(CC) = \mathcal{P}$ and set of (delegate) connector declarations $\text{conns}(CC) = \mathcal{K}$.

4 On the Computation of Observable Behaviours

Let us now discuss how we can efficiently construct the observable behaviour of a component. Since for simple components a specification of its internal behaviour must be explicitly provided, their observable behaviour can be directly constructed and analysed. The problem with analysing the observable behaviour of a composite component is that in this case the observable behaviour of the component relies on the behaviour of the underlying assembly whose construction may be very complex or even not feasible due to state explosion (even if the observable behaviours of the subcomponents are first minimised w.r.t. observational equivalence). To overcome this problem, we realise that several components within an assembly may not play any role for the observable behaviour of the surrounding composite component CC. These subcomponents of CC are called behaviourally neutral and may be removed from the assembly without affecting the observable behaviour of CC.

4.1 Behaviourally Neutral Components and Reduction Strategy

For the formal definition of behavioural neutrality we have to take into account that if we remove components and their connectors from an assembly, components with artificially opened ports remain. These ports should not be available for new connections which is modelled by using unary connectors. Behavioural neutrality is now defined in terms of observational equivalence between the behaviour of an

assembly with two connected components and the behaviour of an assembly with a single component and the binary connector replaced by a unary one.

Definition 4.1 *The component declaration* $d : D[q : Q]$ *is* behaviourally neutral *at* q *for the component declaration* $c : C[p : P]$ *at* p*, if*

$$beh(\langle c : C, d : D; k : (c.p : P, d.q : Q)\rangle) \approx beh(\langle c : C; k : (c.p : P)\rangle) \ .$$

Note that a component declaration $d : D[q : Q]$ can only be neutral at q for another component declaration if $q : Q$ is the only non-trivial port declaration of D, because otherwise the labellings of the compared assemblies would not be the same. Notions similar to behavioural neutrality are used by Bernardo et al. [5, Def. 4.3] under the name of compatibility and Cheung and Kramer [10, Sect. 6.3] under the name of transparency. In fact, Def. 4.1 lifts a notion of neutrality between iotss to component systems, which is formally defined as follows: An iots B is called *neutral* for an iots A, if A and B are composable and $A\theta_S \approx A \otimes B$ where $S = L(A) \bowtie L(B)$ are the shared labels of A and B and θ_S internalises S in A. Intuitively, neutrality expresses that B does not restrict the behaviour given by A if B is composed with A.

Based on the notion of behavioural neutrality, we can describe our reduction strategy for assemblies: If a component at the border of an assembly topology is behaviourally neutral for the next component attached to it, this leaf can not have a behavioural effect on the remaining assembly. We can thus reduce the assembly by removing a neutral leaf component. In order to obtain again a syntactically well-formed assembly, the particular binary connector is removed and replaced by a unary connector (Lem. 4.2). In a second step, we can eliminate also the unary connector, by hiding the port to which the neutral leaf was connected (Lem. 4.3).

Formally, the leaves of an assembly are component declarations which are connected by exactly one binary assembly connector and do not show open ports: For $a \in \text{Asm}$ we let $leaves(a) \subseteq \text{cmps}(a)$ such that $d : D \in leaves(a)$, if, and only if, $open(a) \cap ports(d : D) = \emptyset$ and $|\{k : K \mid k : K \in \text{conns}(a) \wedge |\text{ports}(K)| = 2 \wedge \text{ports}(K) \cap ports(d : D) \neq \emptyset\}| = 1$.

Lemma 4.2 *Let* $a = \langle c : C[p : P], d : D[q : Q], \mathcal{C}; k : (c.p : P, d.q : Q), \mathcal{K}\rangle$ *be an assembly with* $\mathcal{C} \subseteq \text{CmpDcl}$*,* $\mathcal{K} \subseteq \text{ConnDcl}$*, and* $d : D \in leaves(a)$*. If* $d : D$ *is behaviourally neutral at* q *for* $c : C$ *at* p *then*

$$beh(\langle c : C, d : D, \mathcal{C}; k : (c.p : P, d.q : Q), \mathcal{K}\rangle) \approx beh(\langle c : C, \mathcal{C}; k : (c.p : P), \mathcal{K}\rangle) \ .$$

Proof By definition of assembly behaviours, we have that

$$\begin{aligned} &beh(\langle c : C, d : D, \mathcal{C}; k : (c.p : P, d.q : Q), \mathcal{K}\rangle) = \\ &\qquad obs(c : C)\sigma \otimes obs(d : D)\sigma \otimes \textstyle\bigotimes_{c':C' \in \mathcal{C}} obs(c' : C')\sigma \ , \\ &beh(\langle c : C, \mathcal{C}; k : (c.p : P), \mathcal{K}\rangle) = obs(c : C)\sigma' \otimes \textstyle\bigotimes_{c':C' \in \mathcal{C}} obs(c' : C')\sigma' \end{aligned}$$

with synchronisations $\sigma = \sigma_{(\{c.p,d.q\},k)} \cup \sigma_{\mathcal{K}}$ and $\sigma' = \sigma_{(c.p,k)} \cup \sigma_{\mathcal{K}}$. The neutrality of $d : D$ at q for $c : C$ at p amounts to $obs(c : C)\sigma_{(c.p,k)} \approx obs(c : C)\sigma_{(\{c.p,d.q\},k)} \otimes obs(d :$

$D)\sigma_{(\{c.p,d.q\},k)}$. Thus, by Lem. 2.1(ii), we have $obs(c : C)\sigma' \approx obs(c : C)\sigma \otimes obs(d : D)\sigma$. As neither $\sigma_{(c.p,k)}$ nor $\sigma_{(\{c.p,d.q\},k)}$ change $\bigotimes_{c':C'\in\mathcal{C}} obs(c' : C')$, we also have $\bigotimes_{c':C'\in\mathcal{C}} obs(c' : C')\sigma = \bigotimes_{c':C'\in\mathcal{C}} obs(c' : C')\sigma'$, and the claim follows by applying Lem. 2.1(iii). □

For a component C with $p : P \in \text{ports}(C)$ the *hiding of the port* $p : P$ *in* C, written $C \setminus (p : P)$, results in a component $C' \in \text{Cmp}$ such that $\text{ports}(C') = \text{ports}(C) \setminus \{p : P\}$ and $obs(C') = obs(C)/\mathscr{L}(obs_{p:P}(C))$.

Lemma 4.3 *Let* $c : C \in \text{CmpDcl}$ *and* $p : P \in \text{ports}(C)$. *Then*

$$beh(\langle c : C, \mathcal{C}; k : (c.p : P), \mathcal{K}\rangle)/\{k.m \mid m \in \text{msg}(P)\} \approx beh(\langle c : C \setminus (p : P), \mathcal{C}; \mathcal{K}\rangle) .$$

Proof By definition of assembly behaviours and hiding of ports we have that

$$\begin{aligned} &beh(\langle c : C, \mathcal{C}; k : (c.p : P), \mathcal{K}\rangle)/H = (c.obs(C)\sigma \otimes \textstyle\bigotimes_{c':C'\in\mathcal{C}} obs(c' : C')\sigma)/H \ , \\ &beh(\langle c : C \setminus (p : P), \mathcal{C}; \mathcal{K}\rangle) = \\ &\qquad c.(obs(C)/\mathscr{L}(obs_{p:P}(C)))\sigma_{\mathcal{K}} \otimes \textstyle\bigotimes_{c':C'\in\mathcal{C}} obs(c' : C')\sigma_{\mathcal{K}} \end{aligned}$$

with synchronisations $\sigma = \sigma_{(c.p,k)} \cup \sigma_{\mathcal{K}}$ and hiding $H = \{k.m \mid m \in msg(P)\}$. As H does not affect $\mathcal{C}$ and $\mathcal{K}$, the right-hand side of the first equation is observationally equivalent to $c.obs(C)\sigma/H \otimes \bigotimes_{c':C'\in\mathcal{C}} obs(c' : C')\sigma$ by Lem. 2.2(iii). Now $c.obs(C)\sigma/H = (c.obs(C)\sigma_{(c.p,k)}/H)\sigma_{\mathcal{K}}$. By definition of unary synchronisation relabellings we have for $L(obs(C)) = (I, O, T)$ and $L(obs(C)\sigma_{(c.p,k)}) = (I', O', T')$ that $k.m \in T'$ iff $p.m \in I \cup O$. Thus we obtain $c.obs(C)\sigma_{(c.p,k)}/H \approx c.(obs(C)/\mathscr{L}(obs_{p:P}(C)))$. Since removing $\sigma_{(c.p,k)}$ from σ does not change $\bigotimes_{c':C'\in\mathcal{C}} obs(c' : C')$, we have $\bigotimes_{c':C'\in\mathcal{C}} obs(c' : C')\sigma = \bigotimes_{c':C'\in\mathcal{C}} obs(c' : C')\sigma_{\mathcal{K}}$ and the claim follows by applying Lem. 2.1(iii). □

The two lemmas allow us to remove neutral components one after the other. The *neutral leaves* of an assembly a are defined by $neutralleaves(a) \subseteq \text{cmps}(a)$ such that $d : D \in neutralleaves(a)$, if, and only if, there are component declarations $c : C[p : P], d : D[q : Q] \in \text{cmps}(a)$, $d : D \in leaves(a)$, $k : (c.p : P, d.q : Q) \in \text{conns}(a)$ and $d : D$ is behaviourally neutral at q for $c : C$ at p. An assembly a is called *finite* if $\text{cmps}(a)$ is finite. Now define the *syntactical reduction* of a finite assembly $a \in \text{Asm}$ by $red : \text{Asm} \to \text{Asm}$ such that $red(a)$ is the assembly computed by the following algorithm:

```
while neutralleaves(a) ≠ ∅
do d : D ← choose from neutralleaves(a)
   let a = ⟨c : C, d : D, C; k : (c.p : P, d.q : Q), K⟩
                with d : D neutral at q for c : C at p
   a ← ⟨c : C \ (p : P), C; K⟩ od
```

Of course, in general, the assembly a and the reduced assembly $red(a)$ will not be observationally equivalent, because by hiding of ports $red(a)$ shows less communication than a. However, after hiding all internal actions with the ξ-operator we obtain observationally equivalent behaviours.

Lemma 4.4 *Let a be a finite assembly. Then $beh(a)\xi \approx beh(red(a))\xi$.*

Proof The algorithm terminates for each finite assembly a. For each iteration of the `while`-loop let a_0 be the assembly on entry and a_1 the one on exit. We show $beh(a_0)\xi \approx beh(a_1)\xi$. Let $a_0 = \langle c : C, d : D, \mathcal{C}; k : (c.p : P, d.q : Q), \mathcal{K}\rangle$ with $d : D$ neutral at q for $c : C$ at p; then $a_1 = \langle c : C \setminus (p : P), \mathcal{C}; \mathcal{K}\rangle$. By Lem. 4.2, Lem. 4.3 with $H = \{k.m \mid m \in \mathrm{msg}(P)\}$, and Lem. 2.1(i) we have

$$beh(a_0)/H = beh(\langle c : C, d : D, \mathcal{C}; k : (c.p : P, d.q : Q), \mathcal{K}\rangle)/H \approx beh(\langle c : C, \mathcal{C}; k : (c.p : P), \mathcal{K}\rangle)/H \approx beh(\langle c : C \setminus (p : P), \mathcal{C}; \mathcal{K}\rangle) = beh(a_1)\ .$$

As all labels in H are internal, we obtain the invariant $beh(a_0)\xi \approx beh(a_1)\xi$. The claim of the lemma follows from the invariant. □

The reduction strategy for assemblies is directly applicable to the computation of the observable behaviour of composite components

Theorem 4.5 *Let $\langle a; \mathcal{P}; \mathcal{K}\rangle \in$ CCmp with a finite. Then*

$$obs(\langle a; \mathcal{P}; \mathcal{K}\rangle) \approx obs(\langle red(a); \mathcal{P}; \mathcal{K}\rangle)\ .$$

Proof By definition, $leaves(a)$ do not have open ports. Thus no components with a delegate connector to a relay port can be removed from a by the reduction algorithm. Therefore, $\langle red(a); \mathcal{P}; \mathcal{K}\rangle$ is a well-formed composite component. Then the result follows from Lem. 4.4 by renaming actions on open ports of a, and hence of $red(a)$, to actions on the relay ports $\mathcal{P}$. □

Our reduction strategy is sound for arbitrarily structured finite assemblies. But it is particularly useful for assemblies with an *acyclic* topology and for composite components which are *rooted*, i.e., exactly one of the assembly components is connected to the relay ports of the composite component, like in Fig. 6. Then, in

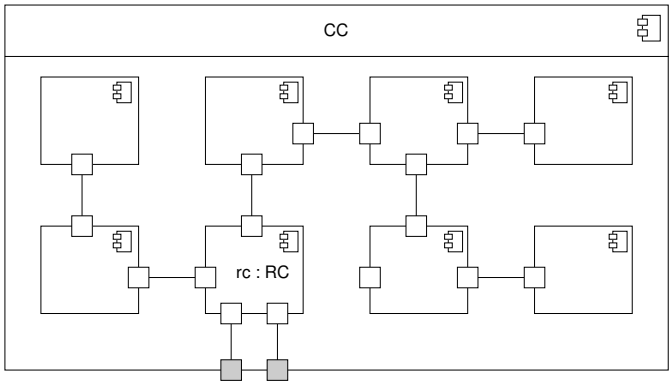


Figure 6. Composite component with acyclic assembly

the best case, all subcomponents but the root rc : RC can be removed such that $obs(\mathsf{CC}) \approx obs(\mathsf{rc} : \mathsf{RC})\rho$ where ρ is a relay relabelling. Note that acyclic topologies do not present a too severe restriction in terms of practical applicability. There is,

e.g., a direct match with the architectural pattern "Blackboard" [8] for the design of distributed communications using a common data structure. Also, the architectural design of a large part of the Common Component Modelling Example [28,19] adheres to an even more restricted acyclic structure, called "star topology" [5], where each leaf is attached to one centre component. Our reduction strategy can also be effectively applied to assembly topologies containing cycles as long as there are neutral leaves around that can be removed. When a cycle is reached we would propose to encapsulate the cycle into a new, nested composite component and to proceed as before by looking again for neutral leaves.

A closer look reveals that in order to decide on neutrality along Def. 4.1, one needs to compute the composition of a leaf component behaviour with the behaviour of the attached component, which can still be quite expensive. To further optimise our method we are interested in criteria to avoid composition of full behaviours of components and to compose smaller transition systems instead. In Sect. 4.3 we show that port behaviours, in particular weakly deterministic port behaviours, can be used to provide such criteria. For this purpose we first need some further technical definitions and results on I/O-transition systems which are presented in the following Sect. 4.2.

4.2 Weakly Deterministic I/O-Transition Systems and Neutrality

Weak Traces

Given an iots $A = (L, Q, q_0, \Delta)$, if we remove in each finite trace of A all occurrences of τ then we obtain the weak traces of A. Formally, for an iots $A = (L, Q, q_0, \Delta)$ the *transitive τ-closure* of Δ is the relation $\hat{\Delta}^* \subseteq Q \times \bigcup L^* \times Q$ defined as follows: For a non-empty sequence of labels $\lambda = l_1 \dots l_{n-1} \in \bigcup L^*$, $(q, \lambda, q') \in \hat{\Delta}^*$, if there are $q = q_1, \dots, q_n = q' \in Q$ such that $(q_i, l_i, q_{i+1}) \in \hat{\Delta}$ for $1 \leq i < n$; for the empty sequence of labels ε, $(q, \varepsilon, q') \in \hat{\Delta}^*$, if $(q, \tau, q') \in \hat{\Delta}$; see Sect. 2.1 for the definition of $\hat{\Delta}$. A *weak trace* of an iots $A = (L, Q, q_0, \Delta)$ is a finite (possibly empty) sequence λ of labels in $\bigcup L$ with $(q_0, \lambda, q) \in \hat{\Delta}^*$ for some $q \in Q$; the set of weak traces of A is denoted by $\mathscr{T}(A)$.

Lemma 4.6 *If A and B are iotss with $A \approx B$, then $\mathscr{T}(A) = \mathscr{T}(B)$.*

Proof Weakly bisimilar iotss have, up to occurrences of τ, the same finite traces. Hence they have the same weak traces. □

Weakly Deterministic I/O-Transition Systems

An iots A is weakly deterministic if all finite traces of A which coincide up to τ lead to observationally equivalent elements of A. Formally, an iots $A = (L, Q, q_0, \Delta)$ is *weakly deterministic* if for all weak traces $\lambda \in \mathscr{T}(A)$ the following holds: If $(q_0, \lambda, q_1) \in \hat{\Delta}^*$ and $(q_0, \lambda, q_2) \in \hat{\Delta}^*$ then $q_1 \approx q_2$, i.e., there is a weak bisimulation R between A and A with $(q_0, q_0) \in R$ and $(q_1, q_2) \in R$.

The notion of weak determinism corresponds to Milner's (weak) determinacy [26, Def. 11.3, Ex. 11.1]; in particular, weak determinism is preserved by observational equivalence [26, Prop. 11.4]. Based on strong bisimulation which treats τ's like

ordinary internal labels also the notion of *strong determinism* may be defined (see Milner [26, Def. 11.2], Cheung and Kramer [10, Def. A.2]), replacing in the definition of weak determinism the requirement of weak bisimulation by the strong notion. Then it may be observed that for iots without τ-transitions strong and weak determinism coincide. In App. A we prove that an iots A is weakly deterministic, if there is an observationally equivalent, minimal iots B without τ-transitions; such a B is a weakly deterministic iots without τ-transitions and hence strongly deterministic. By the transition minimisation procedure of Eloranta [13], a finite τ-free minimal iots is unique up to graph-isomorphism. Thus, for ensuring that a finite iots is weakly deterministic, it suffices to minimise the iots w.r.t. the number of states and transitions and to check that the resulting iots has no τ-transitions and that from each state there is at most one transition for each label.

Weak Determinism and Neutrality

Let us recall that an iots B is called *neutral* for an iots A, if A and B are composable and $A\theta_S \approx A \otimes B$ where $S = L(A) \bowtie L(B)$ are the shared labels of A and B and θ_S internalises S in A. We extend a sufficient criterion, called "interface theorem", from Cheung and Kramer [10] for determining neutrality to our setting. This criterion has been formulated for communicating finite-state processes where parallel composition involves an error state for failing communications and relies on traces, strong process equivalence and strongly deterministic finite-state processes. We generalise the assumptions to include not necessarily finite iots and use observational equivalence and weakly deterministic iotss. [7]

Proposition 4.7 *Let A and B be two composable iotss with $\mathscr{L}(B) \subseteq \mathscr{L}(A)$, and let $S = L(A) \bowtie L(B)$ and $H = \mathscr{L}(A) \setminus \mathscr{L}(B)$. Let B be weakly-deterministic. Then B is neutral for A if, and only if, $\mathscr{T}(A/H) \subseteq \mathscr{T}(B)$.*

Proof First, let $\mathscr{T}(A/H) \subseteq \mathscr{T}(B)$ hold. Let $A = (L_A, Q_A, q_{0,A}, \Delta_A)$, $B = (L_B, Q_B, q_{0,B}, \Delta_B)$, $A\theta_S = (L_A\theta_S, Q_A, q_{0,A}, \Delta_{A\theta_S})$, $A \otimes B = (L_A \otimes L_B, Q_A \times Q_B, (q_{0,A}, q_{0,B}), \Delta_{A\otimes B})$. Let $R = \{(q_A, (q_A, q_B)) \in Q_A \times (Q_A \times Q_B) \mid \exists \lambda \in \mathscr{L}(A)^* \,.\, (q_{0,A}, \lambda, q_A) \in \hat{\Delta}_A^* \wedge (q_{0,B}, \lambda/H, q_B) \in \hat{\Delta}_B^*\}$ where λ/H is the sequence of labels which results from λ when removing all labels in H. Then $(q_{0,A}, (q_{0,A}, q_{0,B})) \in R$. In order to show that R is a weak bisimulation between $A\theta_S$ and $A \otimes B$, let $(q_A, (q_A, q_B)) \in R$.

Let $(q_A, a, q'_A) \in \Delta_{A\theta_S}$. If $a \in H$ or $a = \tau$, then $((q_A, q_B), a, (q'_A, q_B)) \in \Delta_{A\otimes B}$ and $(q'_A, (q'_A, q_B)) \in R$. If $a \in S$, let $\lambda \in \mathscr{L}(A)$ be a weak trace with $(q_{0,A}, \lambda, q_A) \in \hat{\Delta}_A^*$ and $(q_{0,B}, \lambda/H, q_B) \in \hat{\Delta}_B^*$. Then $\lambda a \in \mathscr{T}(A)$ and thus $(\lambda/H)a = (\lambda a)/H \in \mathscr{T}(A/H) \subseteq \mathscr{T}(B)$. Hence there is a $q_B^{(1)} \in Q_B$ such that $(q_{0,B}, \lambda/H, q_B^{(1)}) \in \hat{\Delta}_B^*$ and $(q_B^{(1)}, a, q_B^{(2)}) \in \Delta_B$. As B is weakly deterministic, $q_B^{(1)} \approx q_B$, and thus there is a $q'_B \in Q_B$ with $(q_B, a, q'_B) \in \hat{\Delta}_B$. Thus $((q_A, q_B), a, (q'_A, q'_B)) \in \hat{\Delta}_{A\otimes B}$ and $(q'_A, (q'_A, q'_B)) \in R$.

[7] This generalisation is mentioned by Cheung and Kramer [10, p. 354]; their definition of weak bisimulation, however, does not take into account that τ-transitions may be simulated by idling.

Let $((q_A, q_B), a, (q'_A, q'_B)) \in \Delta_{A\otimes B}$. Then either $a = \tau$ or $a \in S$ or $a \in H$, since $\mathscr{L}(B) \subseteq \mathscr{L}(A)$. If $(q_A, \tau, q'_A) \in \Delta_A$ then $q'_B = q_B$, $(q_A, \tau, q'_A) \in \Delta_{A\theta_S}$ and $(q'_A, (q'_A, q_B)) \in R$; if $(q_B, \tau, q'_B) \in \Delta_B$ then $q'_A = q_A$, $(q_A, \tau, q_A) \in \hat{\Delta}_{A\theta_S}$, and $(q_A, (q_A, q'_B)) \in R$. If $a \in S$, then $(q_A, a, q'_A) \in \Delta_{A\theta_S}$ and $(q'_A, (q'_A, q'_B)) \in R$. If $a \in H$, then $(q_A, a, q'_A) \in \Delta_{A\theta_S}$, $q'_B = q_B$, and $(q'_A, (q'_A, q_B)) \in R$.

Now let $A\theta_S \approx A \otimes B$. Then $A/H \otimes B \approx (A \otimes B)/H$ by Lem. 2.2(iii), as $S \cap H = \emptyset$. Furthermore $(A\theta_S)/H \approx (A/H)\theta_S$, again since $S \cap H = \emptyset$. Thus $(A/H)\theta_S \approx A/H \otimes B$ by Lem. 2.1(i). Since $\mathscr{L}(A/H) = \mathscr{L}(B)$, all labels between A/H and B are shared, and thus $\mathscr{T}(A/H \otimes B) \subseteq \mathscr{T}(B)$. By Lem. 4.6, we have $\mathscr{T}(A/H) = \mathscr{T}((A/H)\theta_S) = \mathscr{T}(A/H \otimes B) \subseteq \mathscr{T}(B)$. □

The following corollary is the crucial result needed for the efficient computation of component behaviours hereafter. Essentially it says that neutrality of a weakly deterministic iots B for some iots A can be propagated to the neutrality of B for some more complex iots C, if A is observationally equivalent to some view C/H on the larger behaviour C.

Corollary 4.8 *Let A, B, and C be iotss such that $\mathscr{L}(B) = \mathscr{L}(A) \subseteq \mathscr{L}(C)$. Let B be composable with A and C where $S = L(A) \bowtie L(B)$ are the shared labels. Let $H = \mathscr{L}(C) \setminus \mathscr{L}(A)$ and $C/H \approx A$. Let B be weakly deterministic. Then B is neutral for A if, and only if, B is neutral for C.*

Proof First, since $C/H \approx A$ we have, by Lem. 4.6, $\mathscr{T}(C/H) = \mathscr{T}(A)$. B is neutral for A iff (by Prop. 4.7, since $\mathscr{L}(B) = \mathscr{L}(A)$) $\mathscr{T}(A) \subseteq \mathscr{T}(B)$ iff $\mathscr{T}(C/H) \subseteq \mathscr{T}(B)$ iff (by Prop. 4.7, taking C for A) B is neutral for C. □

4.3 Neutrality and Port Protocols

In this section, we focus on the behavioural neutrality checks which are performed component-wise in our reduction algorithm of Sect. 4.1. We show that these neutrality checks may be optimised by considering instead of component behaviours the protocols of connected ports.

Components communicate exclusively via ports; therefore it should be sufficient to compare instead of the observable behaviour of two components only the protocols of their connected ports which are usually given by much smaller iotss. Of course, this can only be sound if the observable behaviour of a component, which includes also the sequencing of input and output actions for all of its declared ports, fits to the individual protocols specified for each port, i.e., if the component is correct; see Def. 3.4.

Port neutrality is defined similarly to component neutrality, cf. Def. 4.1, by turning a port declaration $p : P$ into a component declaration $\tilde{p} : \tilde{P}$.

Definition 4.9 *For a port declaration $p : P$ let $\tilde{p} : \tilde{P}$ be a component declaration with* ports$(\tilde{P}) = \{p : P\}$ *and* $obs(\tilde{P}) = prot(p : P)$. *A port declaration $q : Q$ is* behaviourally neutral *for a port declaration $p : P$, if*

$$beh(\langle \tilde{p} : \tilde{P}, \tilde{q} : \tilde{Q}; k : (\tilde{p}.p : P, \tilde{q}.q : Q)\rangle) \approx beh(\langle \tilde{p} : \tilde{P}; k : (\tilde{p}.p : P)\rangle) \ .$$

Unfolding the definition of port neutrality results in a characterisation in terms of iotss: $p : P$ is neutral for $q : Q$ iff $\mathrm{prot}(P) \otimes \mathrm{prot}(Q) \approx \mathrm{prot}(P)\theta_S$ with $S = L(\mathrm{prot}(P)) \bowtie L(\mathrm{prot}(Q))$. Hence, ports can only be behaviourally neutral if their labellings are inverse, i.e., if input and output labels mutually coincide.

Port neutrality has the important consequence that the observable behaviour of a correct component is not affected if a port of the component is connected to a neutral port of another component, provided that this port has a weakly deterministic behaviour. The following theorem is an application of the results discussed on the level of I/O-transition systems in Sect. 4.2. It is at the core of a more efficient check for neutrality in assemblies. Note that component C may have more than one port.

Theorem 4.10 *Let $c : C, d : D \in$ CmpDcl with C and D correct components, $p : P \in \mathrm{ports}(C)$ and $\mathrm{ports}(D) = \{q : Q\}$. If $\mathrm{prot}(Q)$ is weakly deterministic and $q : Q$ is behaviourally neutral for $p : P$, then $d : D$ is behaviourally neutral at q for $c : C$ at p.*

Proof We need to show that

$$beh(\langle \tilde{p} : \tilde{P}, \tilde{q} : \tilde{Q}; k : (\tilde{p}.p : P, \tilde{q}.q : Q)\rangle) \approx beh(\langle \tilde{p} : \tilde{P}; k : (\tilde{p}.p : P)\rangle) \quad \text{implies}$$
$$beh(\langle c : C, d : D; k : (c.p : P, d.q : Q)\rangle) \approx beh(\langle c : C; k : (c.p : P)\rangle) \ .$$

By unfolding definitions this amounts to show that

$$\tilde{p}.prot(p : P)\sigma_2 \otimes \tilde{q}.prot(q : Q)\sigma_2 \approx \tilde{p}.prot(p : P)\sigma_1 \quad \text{implies}$$
$$obs(c : C)\sigma_2' \otimes obs(d : D)\sigma_2' \approx obs(c : C)\sigma_1'$$

with $\sigma_2 = \sigma_{(\{\tilde{p}.p,\tilde{q}.q\},k)}$, $\sigma_1 = \sigma_{(\tilde{p}.p,k)}$, $\sigma_2' = \sigma_{(\{c.p,c.q\},k)}$ and $\sigma_1' = \sigma_{(c.p,k)}$. Obviously it suffices to prove that

$$prot(p : P)\sigma_{(\{p,q\},k)} \otimes prot(q : Q)\sigma_{(\{p,q\},k)} \approx prot(p : P)\sigma_{(p,k)} \quad \text{implies}$$
$$obs(c : C)\sigma_{(\{p,q\},k)} \otimes obs(d : D)\sigma_{(\{p,q\},k)} \approx obs(c : C)\sigma_{(p,k)} \ .$$

Since D is correct and shows only a single port we can replace the antecedent by

$$prot(p : P)\sigma_{(\{p,q\},k)} \otimes obs(d : D)\sigma_{(\{p,q\},k)} \approx prot(p : P)\sigma_{(p,k)} \ .$$

For the succedent observe that also C is correct w.r.t. $p : P$, i.e. $obs(C)/H \approx prot(p : P)$ with H as in Def. 3.3, i.e., we have to prove that

$$(obs(C)/H)\sigma_{(\{p,q\},k)} \otimes obs(d : D)\sigma_{(\{p,q\},k)} \approx (obs(C)/H)\sigma_{(p,k)} \quad \text{implies}$$
$$obs(c : C)\sigma_{(\{p,q\},k)} \otimes obs(d : D)\sigma_{(\{p,q\},k)} \approx obs(c : C)\sigma_{(p,k)} \ .$$

Now, H does not interfere with by $\sigma_{(\{p,q\},k)}$ and $\sigma_{(p,k)} = \theta_S \circ \sigma_{(\{p,k\},k)}$; thus writing A for $obs(C)\sigma_{(\{p,q\},k)}/H$, B for $obs(d : D)\sigma_{(\{p,q\},k)}$ and C for $obs(C)\sigma_{(\{p,q\},k)}$, we simply have an instance of Cor. 4.8. □

The theorem allows for a port-based computation of the neutral leaves of an assembly which links our analysis back to the results of Sect. 4.1.

Corollary 4.11 *Let a be an assembly, $d : D \in leaves(a)$, $\mathrm{ports}(D) = \{q : Q\}$ and $\{k : (c.p : P, d.q : Q)\} \in \mathrm{conns}(a)$. If $\mathrm{prot}(Q)$ is weakly deterministic, D and $cmp(c.p : P)$ are correct, and $q : Q$ is behaviourally neutral for $p : P$ then $d : D \in neutralleaves(a)$.*

Proof By Thm. 4.10 we derive neutrality of the leaf component D from the given port neutrality, and with the definition of neutral leaves the claim follows. □

4.4 Application to the Compressing Proxy Example

Let us now apply our reduction algorithm of Sect. 4.1 and the results of this section to the efficient computation of the observable behaviour of CompressingProxy. We start with the underlying assembly of CompressingProxy, call it a, which contains the three components adapt : Adaptor, gzip : GZip, and gifToJpg : GifToJpg. First, we choose the leaf gzip : GZip and consider its port z : Zip. Obviously, the port protocol of Zip is weakly deterministic, since there are no τ-transitions and the protocol is already (strongly) deterministic; cf. Fig. 3(d). Moreover, the component (types) GZip and Adaptor are correct w.r.t. their ports. Then we check that the port z : Zip is behaviourally neutral for the (connected) port t : TxtCompr of Adaptor by constructing the port product prot(TxtCompr) ⊗ prot(Zip) shown in Fig. 7.

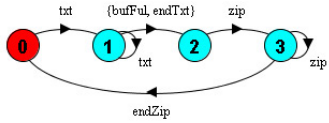


Figure 7. Port product of TxtCompr and Zip

Obviously, the port product is observationally equivalent to prot(TxtCompr)θ_S (cf. Fig. 3(b)) where the shared labels $S = \{$txt, endTxt, zip, endZip, bufFul$\}$ are internalised. Hence, by Cor. 4.11, gzip : GZip is a neutral leaf of the assembly. Our algorithm now removes this leaf and hides the port t : TxtCompr, i.e., the component type Adaptor is replaced by the component type Adaptor' whose observable behaviour is obtained, by definition of port hiding, from Adaptor by hiding all labels related to t : TxtCompr. The resulting behaviour of Adaptor' has a lot of τ-transitions and can be minimised to an iots with 4 states and 5 transitions. Let a' be the new assembly obtained from a after the first reduction step.

In the second step, we choose the leaf gifToJpg : GifToJpg of a' and consider its port j : ToJpg. Unfortunately, the port protocol of ToJpg is not weakly deterministic, since there is a weak trace gif which leads to state 1 and also to state 2 (cf. Fig. 3(e)) but both states cannot be observationally equivalent since the protocol is already minimised. Thus we cannot apply Cor. 4.11 and we have to show directly on the level of component behaviours that gifToJpg : GifToJpg is behaviourally neutral (at port j) for adapt : Adaptor' (at port g). The neutrality check is indeed successful and amounts to the construction of an iots with 12 states and 23 transitions before it is minimised. Our algorithm now removes also the leaf gifToJpg : GifToJpg from a' and hides the port g : GifCompr of Adaptor'. Thus, we obtain as the final result of the reduction an assembly a'' consisting of a single component adaptor : Adaptor" where Adaptor"

has only two ports u : UpStream and d : DownStream. The observable behaviour of Adaptor" has only two states and three transitions after minimisation and hence the same holds for the behaviour of a''. Finally, we apply Thm. 4.5 which shows that obs(CompressingProxy) is observationally equivalent to the observable behaviour of a composite component which encapsulates the very simple assembly a'' and which results in the observable behaviour shown in Fig. 5(b). The most expensive task in our reduction was the neutrality check between gifToJpg : GifToJpg and adapt : Adaptor' which led to the construction of a product iots with 12 states and 23 transitions (before minimisation). On the other hand, if we would have directly computed the observable behaviour of CompressingProxy based on the behaviour of the original assembly a than the most expensive task would have been the construction of the behaviour of a which has 30 states and 65 transitions (also before minimisation).

Let us remark again that the behaviour of a'' is not at all observationally equivalent to the behaviour of our original assembly a and that the results of our approach rely fundamentally on the additional abstraction steps that are applied when the observable behaviour of a composite component is defined.

5 Related Work

Our approach integrates aspects from different but closely related domains. As a concrete syntax we use the UML2 [27] for the specification of port-based components. Our component model, however, is a formally defined software component model [21] which can be used for architectural programming in the sense of, e.g., Java/A [4], ArchJava [2], ComponentJ [29], and also SOFA [7] or Fractal [6]. Our component model is backed by I/O-transition systems and observational equivalence which resembles ADL approaches such as Wright [3], Darwin [22], and more recently PADL [5]. The analysis carried out for computing composite component behaviours more efficiently relies on a reduction strategy similar to contextual CRA [10]. Neutrality between transition systems, as a criterion for state space reduction, is closely related to the transparency property of interface processes [10] and also to the notion of compatibility used in PADL [5] to detect deadlock-related architectural mismatches. We detail on each of the mentioned aspects in the following.

The port-based component model was originally inspired by ROOM [30], which later evolved to UML for Real-Time Systems [31] which in turn in the meantime has been incorporated into UML2 [27]. UML2 introduces notions of components, ports and structured classifiers which are, not surprisingly, a perfect match with the syntactic requirements of our port-based approach to components. Hence we use UML2 as a concrete syntax for the specification of port-based components. Given that the UML2 shows a number of semantical variation points, and also ambiguities, see e.g. [11,14], we provided a metamodel for our component model which is, on the one hand, easily mappable to the corresponding notions of the UML2 and provides, on the other hand, a convenient starting point for our formalisation of static structure and behaviour of port-based components. In this sense our semantics may also be used to remedy the ambiguities of UML2 port semantics as discussed in [11].

We have introduced a formal, algebraic layer for our component model which is distinguished from its semantic backend, I/O-transition systems. Based on a precise understanding of the structural properties of hierarchical systems with port-based components, we identified conceptually and formally which kinds of behaviour are involved and, in particular, discussed the difference between specified and derived behaviours. PADL [5] and its extension as described in [1], deals also with formal modelling and verification of software architectures using component-oriented techniques. This approach extends a process algebra by architectural concepts like "architectural type" [1, Def. 2.4] which is similar to our notion of an assembly. In [1] PADL comes with various kinds of extension mechanisms based on architectural type invocation. Architectural type invocations do not hide local interactions which makes a principal difference to our notion of composite component which encapsulates an assembly by hiding all its internal communications. In our view, this encapsulation is methodologically and technically quite important for building components in a hierarchical way.

Closer to our approach is Darwin [24,20] as a language for design specifications of distributed systems. In particular, in combination with the Tracta method [15] it becomes evident that we do not only share key abstractions such as components encapsulated by ports but also the semantic domain of labelled transition systems (LTS) together with a compositional reachability analysis (CRA) for composed systems which utilises specifications of the static structure. Even though I/O-transition systems provide a more convenient way to be precise about whether an action is a message reception, a message sending or an internal action, we could have used LTSs for the purpose of this article as well. However, since we are currently also working on a refinement relation where the difference between input and output becomes important, we just sticked to our semantic domain of I/O-transition systems.

In Darwin the importance of a separate configuration language for the specification of structural aspects is stressed [20]. Given such a "common structural view", behaviour of primitive components described by LTSs is added in a "behavioural view" and used for an architectural analysis of the composed system behaviour [24]. An operational semantics is developed translating notions of Darwin such as "binding" or "external interface" to an LTS operator such as relabelling or hiding [15]. Obviously, our work is related in several concerns. First, and as argued above, with the UML2 we are also in favour of a clearly separated language for the specification of structural aspects. In fact, the UML2 notation for structured classifier resembles the graphical notation of Darwin. However, we use an algebraic layer between the UML2 specifications of components and the I/O-transition systems describing their behaviour.

Second, touching an, as we think, new aspect which is instrumental in CRA, we consider ports not as a mere structural, syntactical aid in system construction, but require explicitly specified port behaviours instead. Ports become first-class citizens which is neither the case in the contextual CRA applied to Darwin [10] nor in the analysis developed for PADL [5]. In [10] automatically derived context constraints are used to construct the LTS behaviour of composed systems more efficiently.

Context constraints take the form of interface processes which capture the interplay between a set of composed processes playing the role of an environment for a single fixed process as part of the composition. If the composition of interface and fixed process results in a smaller transition system, it is substituted. The correctness of the approach relies on a transparency property which requires a strong semantic equivalence between a (possibly) composed process P and its composition $P \parallel I$ with the interface process I. Criteria which guarantee the transparency are identified in an interface theorem. Our notion of neutrality between component behaviours uses observational equivalence and formulates criteria based on weakly deterministic port behaviours. Even though the possibility of using weaker equivalences is mentioned in [10, p. 354], it is not elaborated. Moreover, the criteria do not make use of port behaviours.

PADL [5] applies observational equivalence for a notion of compatibility between components which expresses the same idea as our notion of behavioural neutrality. In PADL compatibility is used to detect architectural mismatches and it is shown that pairwise compatibility is a sufficient criterion to derive deadlock-freedom of an acyclic assembly from the deadlock-freedom of its local components. This technique is, however, not appropriate for an explicit construction of assembly behaviours. Nevertheless, since we are interested in the encapsulated behaviour of an assembly given by the observable behaviour of a composite component, we can abstract from component communications within an assembly which allows us to utilise again behavioural neutrality (compatibility) for an efficient construction of the observable behaviour of a composite component. Moreover, as observed in our study, to decide on neutrality of component behaviours can still be quite expensive. Therefore, as discussed in the comparison with [10] above, a further important difference between our approach and [5] concerns the integration of explicit port behaviours which makes the check for neutrality more efficient, under the assumption of weakly deterministic port protocols.

6 Conclusion

We have studied composite components and their observable behaviour based on a metamodel which clearly distinguishes different kinds of behaviours that are relevant for the construction and analysis of hierarchical systems using port-based components. We have provided conditions for a more efficient computation of the observable behaviour of a composite component using syntactically reduced assemblies. At the core of our reduction strategy is a notion of neutrality between component behaviours. The approach is particularly useful for rooted composite components with acyclic assemblies, since then it might be the case that it is only the root component which remains in the reduced assembly.

We identified criteria to derive neutrality between components from the neutrality between their ports. Thus, on the one hand, we improve the efficiency of the neutrality check involved in our reduction strategy, and, on the other hand, we shed light on the general usefulness of ports as first-class citizens of a software

component model. Until it comes to verification, ports are without doubt useful from a methodological point of view, though playing a mere syntactical role. We have shown that ports are also semantically important for the behavioural analysis of component-based systems. Hence, we consider the port-based component model with explicit port protocols as a useful extension of related results discussed above.

The focus of this study is the explicit semantic characterisation of hierarchical port-based components rather than a property-based analysis of port, component or assembly behaviours. But, of course, our computation of observable component behaviours provides the basis on which properties of components that are compatible with observational equivalence can be studied. For instance, if we know that the observable behaviour of a composite component is deadlock-free (in the sense of PADL [5]), then also the behaviour of the underlying assembly must be deadlock-free, since deadlock-freeness is preserved by "unhiding". In particular, in the context of component systems that are built on various hierarchical layers our techniques allow us to climb up the hieararchy and to analyse systems on the appropriate abstraction level. It would be interesting to study to which extent our approach could be combined with techniques such as deadlock-analysis for interaction systems [25], analysis of safety and liveness properties for communicating LTSs with Tracta [15], or the compositional verification of Real-Time UML designs [16].

In future work, our study should be extended to take into account not only derived observable behaviours but also requirement specifications for the observable behaviour of components as done, e.g., by the frame protocols used in [7]. We believe that then the proposed techniques to compute the observable behaviour of a composite component can be efficiently applied to verify refinement correctness between specification and observable behaviour. Moreover, we plan to extend our results to asynchronous message passing within the assemblies of connected components and to support architectural programming by adjusting Java/A [4] and its analysis tools to the results of this paper.

Acknowledgement

We thank the anonymous reviewers of the first version of this paper for many valuable remarks and suggestions and we are grateful to Mila Majster-Cederbaum for intensive and stimulating discussions on components and component behaviours.

References

[1] Alessandro Aldini and Marco Bernardo. On the Usability of Process Algebra: An Architectural View. *Theo. Comp. Sci.*, 335(2–3):281–329, 2005.

[2] Jonathan Aldrich. *Using Types to Enforce Architectural Structure.* PhD thesis, University of Washington, 2003.

[3] Robert Allen and David Garlan. A Formal Basis for Architectural Connection. *ACM Trans. Softw. Eng. Meth.*, 6(3):213–249, 1997.

[4] Hubert Baumeister, Florian Hacklinger, Rolf Hennicker, Alexander Knapp, and Martin Wirsing. A Component Model for Architectural Programming. In *Proc. FACS'05*, volume 160 of *Elect. Notes Theo. Comp. Sci.*, pages 75–96. Elsevier, 2006.

[5] Marco Bernardo, Paolo Ciancarini, and Lorenzo Donatiello. Architecting Families of Software Systems with Process Algebras. *ACM Trans. Softw. Eng. Meth.*, 11(4):386–426, 2002.

[6] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma, and Jean-Bernard Stefani. The FRACTAL Component Model and its Support in Java. *Softw., Pract. Exper.*, 36(11–12):1257–1284, 2006.

[7] Tomáš Bureš, Petr Hnětynka, and František Plášil. SOFA 2.0: Balancing Advanced Features in a Hierarchical Component Model. In *Proc. SERA'06*, pages 40–48. IEEE, 2006.

[8] Frank Buschmann, Kevlin Henney, and Douglas C. Schmidt. *Pattern Oriented Software Architecture Vol. 5: On Patterns and Pattern Languages*. Wiley, 2007.

[9] Cyril Carrez, Alessandro Fantechi, and Elie Najm. Behavioural Contracts for a Sound Composition of Components. In *Proc. FORTE'03*, volume 2767 of *Lect. Notes Comp. Sci.*, pages 111–126. Springer, 2003.

[10] Shing Chi Cheung and Jeff Kramer. Context Constraints for Compositional Reachability Analysis. *ACM Trans. Softw. Eng. Meth.*, 5(4):334–377, 1996.

[11] Arnaud Cuccuru, Sébastien Gérard, and Ansgar Radermacher. Meaningful Composite Structures. In *Proc. MoDELS'08*, volume 5301 of *Lect. Notes Comp. Sci.*, pages 828–842. Springer, 2008.

[12] Luca de Alfaro and Thomas A. Henzinger. Interface-based Design. In Manfred Broy, Johannes Grünbauer, David Harel, and C. A. R. Hoare, editors, *Engineering Theories of Software-intensive Systems*, volume 195 of *NATO Science Series: Mathematics, Physics, and Chemistry*, pages 83–104. Springer, 2005.

[13] Jaana Eloranta. Minimizing the Number of Transitions with Respect to Observation Equivalence. *BIT Num. Math.*, 31(4):576–590, 1991.

[14] Harald Fecher, Jens Schönborn, Marcel Kyas, and Willem P. de Roever. 29 New Unclarities in the Semantics of UML 2.0 State Machines. In *Proc. ICFEM'05*, volume 3785 of *Lect. Notes Comp. Sci.*, pages 52–65. Springer, 2005.

[15] Dimitra Giannakopoulou, Jeff Kramer, and Shing Chi Cheung. Behaviour Analysis of Distributed Systems Using the Tracta Approach. *Autom. Softw. Eng.*, 6(1):7–35, 1999.

[16] Holger Giese, Matthias Tichy, Sven Burmester, and Stephan Flake. Towards the Compositional Verification of Real-Time UML designs. In *Proc. FSE'03*, pages 38–47. ACM, 2003.

[17] Gregor Gössler, Susanne Graf, Mila Majster-Cederbaum, Moritz Martens, and Joseph Sifakis. Ensuring Properties of Interaction Systems by Construction. In Thomas Reps, Mooly Sagiv, and Jörg Bauer, editors, *Program Analysis and Compilation, Theory and Practice — Essays Dedicated to Reinhard Wilhelm on the Occasion of His 60th Birthday*, volume 4444 of *Lect. Notes Comp. Sci.*, pages 201–224. Springer, 2007.

[18] Gregor Gössler and Joseph Sifakis. Composition for Component-based Modeling. *Sci. Comp. Prog.*, 55(1–3):161–183, 2005.

[19] Alexander Knapp, Stephan Janisch, Rolf Hennicker, Allan Clark, Stephen Gilmore, Florian Hacklinger, Hubert Baumeister, and Martin Wirsing. Modelling the CoCoME with the Java/A Component Model. In Rausch et al. [28], pages 207–237.

[20] Jeff Kramer and Jeff Magee. Exposing the Skeleton in the Coordination Closet. In *Proc. COORDINATION'97*, volume 1282 of *Lect. Notes Comp. Sci.*, pages 18–31. Springer, 1997.

[21] Kung-Kiu Lau and Zheng Wang. Software Component Models. *IEEE Trans. Softw. Eng.*, 33(10):709–724, 2007.

[22] Jeff Magee, Naranker Dulay, Susan Eisenbach, and Jeff Kramer. Specifying Distributed Software Architectures. In *Proc. ESEC'95*, volume 989 of *Lect. Notes Comp. Sci.*, pages 137–153. Springer, 1995.

[23] Jeff Magee and Jeff Kramer. *Concurrency — State Models and Java Programs*. Wiley, 1999.

[24] Jeff Magee, Jeff Kramer, and Dimitra Giannakopoulou. Analysing the Behaviour of Distributed Software Architectures: A Case Study. In *Proc. FTDCS'97*, pages 240–247. IEEE, 1997.

[25] Moritz Martens and Mila Majster-Cederbaum. Compositional Analysis of Deadlock-Freedom of Tree-Like Component Architectures. In *Proc. EMSOFT'08*, pages 199–206. ACM, 2008.

[26] Robin Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[27] Object Management Group. Unified Modeling Language: Superstructure, Vers. 2.1.2. Specification, OMG, 2007.

[28] Andreas Rausch, Ralf Reussner, Raffaela Mirandola, and František Plášil, editors. *The Common Component Modeling Example: Comparing Software Component Models*, volume 5153 of *Lect. Notes Comp. Sci.* Springer, 2008.

[29] João Costa Seco and Luis Caires. A Basic Model of Typed Components. In *Proc. ECOOP'00*, volume 1850 of *Lect. Notes Comp. Sci.*, pages 108–128. Springer, 2000.

[30] Bran Selic, Garth Gullekson, and Paul T. Ward. *Real-Time Object-Oriented Modeling.* Wiley, 1994.

[31] Bran Selic and Jim Rumbaugh. Using UML for Modeling Complex Real-Time Systems. Technical report, ObjectTime Ltd., Rational Software Corp., 1998.

A Minimal Weakly Deterministic I/O-Transition Systems

An iots $A = (L, Q, q_0, \Delta)$ is called *minimal*, if (i) for all $q \in Q$ there is a $\lambda \in \mathscr{T}(A)$ with $(q_0, \lambda, q) \in \hat{\Delta}^*$, i.e., all states of A are reachable from q_0; and (ii) for all $q_1, q_2 \in Q$, if $q_1 \approx q_2$, then $q_1 = q_2$. Note that if $A \approx B$ and A is minimal, then the cardinality of the states of A is bounded by the cardinality of the states of B.

Proposition A.1 *If A is a weakly deterministic iots, then there is a minimal iots B without τ-transitions such that $A \approx B$.*

Proof Let $A = (L, Q_A, q_{0,A}, \Delta_A)$ be a weakly deterministic iots. Define (by abuse of notation) $\approx \subseteq \mathscr{T}(A) \times \mathscr{T}(A)$ by

$$\lambda_1 \approx \lambda_2 \iff (\forall q_1, q_2 \in Q_A \,.\, (q_{0,A}, \lambda_1, q_1) \in \hat{\Delta}_A^* \wedge (q_{0,A}, \lambda_2, q_2) \in \hat{\Delta}_A^* \supset q_1 \approx q_2) \,.$$

Then $\approx$ is an equivalence relation on $\mathscr{T}(A)$: $\approx$ is reflexive as A is weakly deterministic, it is symmetric by definition, and it is transitive as only weak traces are considered. Furthermore, if $\lambda_1 \approx \lambda_2$ and $l \in \mathscr{L}(A)$, then $\lambda_1 l \approx \lambda_2 l$: If $(q_{0,A}, \lambda_1, q_1) \in \hat{\Delta}_A^*$ and $(q_1, l, q_1') \in \hat{\Delta}_A$, and $(q_{0,A}, \lambda_2, q_2) \in \hat{\Delta}_A^*$ and $(q_2, l, q_2') \in \hat{\Delta}_A$, then $q_1 \approx q_2$; thus there is a $q_2'' \in Q_A$ with $(q_2, l, q_2'') \in \hat{\Delta}_A$ and $q_1' \approx q_2''$; but then $(q_{0,A}, \lambda_2 l, q_2'') \in \hat{\Delta}_A^*$ and $(q_{0,A}, \lambda_2 l, q_2') \in \hat{\Delta}_A^*$ and thus $q_2'' \approx q_2'$, as A is weakly deterministic. We write $[\lambda]_\approx$ for the equivalence class of $\lambda \in \mathscr{T}(A)$ in the quotient $\mathscr{T}(A)/\approx$ of $\mathscr{T}(A)$ w.r.t. $\approx$.

Define an iots $B = (L, Q_B, q_{0,B}, \Delta_B)$ by $Q_B = \{[\lambda]_\approx \mid \lambda \in \mathscr{T}(A)\}$, $q_{0,B} = [\varepsilon]_\approx$, and $([\lambda]_\approx, l, [\lambda']_\approx) \in \Delta_B$ if $\lambda l \in [\lambda']_\approx$. Then B has no τ-transitions; and $A \approx B$ by using $R = \{(q, [\lambda]_\approx) \subseteq Q_A \times \mathscr{T}(A)/\approx \mid \exists \lambda' \in [\lambda]_\approx \,.\, (q_{0,A}, \lambda', q) \in \hat{\Delta}_A^*\}$ as weak bisimulation between A and B with $(q_{0,A}, [\varepsilon]_\approx) \in R$. B is also minimal: All $[\lambda]_\approx \in Q_B$ are reachable by definition. If $[\lambda_1]_\approx, [\lambda_2]_\approx \in Q_B$ with $[\lambda_1]_\approx \approx [\lambda_2]_\approx$ in B, let $q_1, q_2 \in Q_A$ with $(q_{0,A}, \lambda_1, q_1), (q_{0,A}, \lambda_2, q_2) \in \hat{\Delta}_A^*$; then $q_1 \approx q_2$ in A, as $R = \{(q_1', q_2') \mid \exists \lambda' \in \bigcup L^* \,.\, (q_{0,A}, \lambda_1 \lambda', q_1') \in \hat{\Delta}_A^* \wedge (q_{0,A}, \lambda_2 \lambda', q_2') \in \hat{\Delta}_A^*\}$ is a weak bisimulation on A with $(q_1, q_2) \in R$. Thus $\lambda_1 \approx \lambda_2$ and hence $[\lambda_1]_\approx = [\lambda_2]_\approx$. □