On the Compositional Analysis of Hierarchical Components with Explicit Ports

Rolf Hennicker, Stephan Janisch, Alexander Knapp

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

Abstract

We propose an analysis method for the behaviour of large scale components following their modular and hierarchical structure. The method mirrors the different verification tasks concerning the developer of a single component and the system architect who puts components together. Our main results provide, first, criteria under which the deadlock freeness of hierarchical components can be checked and, secondly, criteria which allow for checking the correctness of components w.r.t. the observable behaviour at their explicit ports. For the static structure of components we use UML 2.0 notation and for the description of behaviours we use I/O-transition systems.

Keywords: component composition, verification methods, behaviour specifications, input/output transition systems

1 Introduction

Components, being based on the notion of strong encapsulation, foster a compositional approach to behavioural verification. Port-based component models, as e.g. the ROOM model [17], stress this encapsulation aspect by taking ports to form the exclusive and explicit communication windows of components; additionally, ports clarify the different views on a component and its possible uses. The behavioural properties of a component should thus be governed only by the observable behaviour at the component ports. In particular, when composing components hierarchically all sub-component implementation details become hidden behind the port facade of the super-component.

The compositional analysis of component properties is, in fact, a long-standing topic and several proposals have been provided regarding the analysis of different architectural styles [4,1,2,5], global deadlock, liveness, and progress properties [10,11] and protocol correctness [19] — to mention just a few. However, the study of explicit ports in hierarchical components has received little attention.

[1] This research has been partially supported by the GLOWA-Danube project (01LW0303A) sponsored by the German Federal Ministry of Education and Research.

In our analysis approach we are interested in checking the deadlock-freeness and correctness of components. The basic idea is to proceed in a hierarchical way, starting from the analysis of the (local) properties of simple components and their ports from which we can then derive, by a compositionality result for particular acyclic topologies, properties of composite components. Thus, following the hierarchical construction of components, we finally obtain results for the behaviour of the global system. Our analysis process consists of the following steps:

(i) For each simple component we analyse
  - the behaviour specification of each of its ports,
  - the behaviour specification of the component itself, and
  - the relationships between the component behaviour and the behaviour specified for each of its ports.

(ii) For each composite component we analyse
  - the interaction behaviour of connected ports,
  - the behaviour of the composite component which can be inferred from the behaviours of its constituent parts, and
  - the relationships between the behaviour of the composite component and the behaviour of each of its relay ports.

The semantic basis of our study are labelled I/O-transition systems which are used to formally represent the behaviour specifications of ports and components. The static structure of components is specified in UML 2.0.

The remainder of this paper is structured as follows: In Sect. 2 we discuss related work. Sect. 3 summarises the foundations of our formal model, Sect. 4 reviews our component model by means of an example. In Sect. 5, we start with the analysis of simple components and pursue our method in Sect. 6 with analysing composite components. Finally, Sect. 7 summarises our approach and outlines some future work.

2 Related Work

Most closely related to our study is the work of Bernardo et al. [4,1,2] which is also motivated by the derivation of global properties from local ones, structured along architectural types. Though [4,2] considers more general architectural topologies (including cyclic structures) the conditions for modular verification are significantly stronger requiring what we would call "behaviour reflection" for all connections between architectural elements, i.e. components. In contrast to [4,2] we are strongly in favour of using explicit port protocols with a preferably simple presentation on which the system architect can rely when connecting components.

In [14] the behaviour of components within hierarchical component architectures is specified in terms of behaviour protocols [15]. The approach focuses on local analysis of components under a particular (preferably small) environment which guarantees the functioning of the component when plugged into the whole system. Instead of computing environments for checking local components the use of explicit port protocols which define the assumptions and guarantees w.r.t. any compatible environment looks beneficial.

Gössler et al. [10,11] analyse global deadlock-freedom, liveness and local progress of component-based systems building on a theoretical framework for component-based

modelling, called interaction systems [12]. Systems may be constructed in a hierarchical way, but the individual components remain visible after composition. This is in contrast to our approach with explicitly specified relay ports of composite components.

In [18] a general framework for the modelling and analysis of component-based systems is developed, specifically considering the problem of component substitutability. Properties such as deadlock-freedom are not discussed explicitly, but may carry over if the parameterised formalism of component interaction automata is applied to our port-based component model.

Carrez et al. [5] define a notion of sound assemblies for which (external) deadlock-freeness is shown. An assembly is a configuration of components and contracts. It is sound if all components are correct w.r.t. to their interface specifications and all interconnections are compatible. For sound assemblies deadlock-freeness is shown by checking a dependency relation for cycles. The relation is constructed along the specified communications. Aspects such as observational equivalence, minimisation or hierarchical system construction are not considered.

Tracta [9] defines a notion of composite components where the behaviour is, similar to our approach, computed from the behaviour of the particular subcomponents, themselves represented by labelled transition systems. Hierarchical composition is efficiently supported by employing transition systems after minimisation w.r.t. observational equivalence. However, explicit (relay) ports are not considered, hence the correctness of composite components is not of concern. Furthermore, behaviours, be they specified or computed, are checked against user-defined properties instead of deriving global properties from local behaviours.

In Wright [3] the focus is on the formalisation and the analysis of local component properties and connectors without drawing global conclusions like deadlock-freedom in hierarchically constructed systems.

Yellin and Strom [19] define a notion of protocol compatibility between two communication partners based on so-called collaboration specifications, essentially defining the legal ordering of interface messages by means of finite state machines. Concentrating on single interactions, neither do they consider composite components, nor global properties of composed systems.

3 Preliminaries: I/O-Transition Systems

In the following we summarise the basic definitions, operators and facts for I/O-transition systems that will be needed hereafter for the behaviour specifications and behavioural analysis of components and ports. Our definitions are inspired by the interface automata approach of [8]. From the theoretical point of view it is not necessary to restrict here to finitary transition systems but in the concrete cases we will only consider finitary transition systems which allow for model checking.

An *I/O-labelling* (I, O, T) consists of three mutually disjoint sets of *input* (or *provided*) labels I, *output* (or *required*) labels O, and *internal* labels T. An *I/O-transition system* $A = (Q, q_0, \Delta)$ over an I/O-labelling (I, O, T) is given by a set of *states* Q, an *initial state* $q_0 \in Q$ and a *transition relation* $\Delta \subseteq Q \times (I \cup O \cup T \cup \{\tau\}) \times Q$; the distinguished action τ is called *invisible* (or *silent*) action. The set of *labels* of A is given by $\mathit{Label}(A) = I \cup O \cup T$; the set of *actions* of A is given by $\mathit{Action}(A) = \mathit{Label}(A) \cup \{\tau\}$. We define the τ-closure

of Δ as $\hat{\Delta} \subseteq Q \times (I \cup O \cup T \cup \{\tau\}) \times Q$ as follows: For an $l \in \mathit{Label}(A)$, $(q, l, q') \in \hat{\Delta}$, if there are $q = q_0, q_1, \ldots, q_m \in Q$ and $q'_0, \ldots, q'_n = q' \in Q$ such that $(q_i, \tau, q_{i+1}) \in \Delta$ for $0 \leq i < m$, $(q_m, l, q'_0) \in \Delta$, and $(q'_j, \tau, q'_{j+1}) \in \Delta$ for $0 \leq j < n$; and $(q, \tau, q') \in \hat{\Delta}$ if there are $q = q_0, q_1, \ldots, q_n = q' \in Q$ such that $(q_i, \tau, q_{i+1}) \in \Delta$ for $0 \leq i < n$.

An *I/O-transition system* $A = (Q, q_0, \Delta)$ is *deadlock-free*, if for all states $q \in Q$ reachable from q_0 by transitions of Δ there exists a transition $(q, l, q') \in \hat{\Delta}$ with $l \in \mathit{Label}(A)$.

Two I/O-transition systems $A = (Q_A, q_{0,A}, \Delta_A)$ and $B = (Q_B, q_{0,B}, \Delta_B)$ over the same I/O-labelling (I, O, T) are *observationally equivalent* [13], denoted by $A \approx B$, if there exists a weak bisimulation relation R between A and B with $(q_{0,A}, q_{0,B}) \in R$. A relation $R \subseteq Q_A \times Q_B$ is a *weak bisimulation relation* between A and B, if for all $(q_A, q_B) \in R$ and all $l \in \mathit{Action}(A)$ the following holds:

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

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

Observational equivalence is compatible with deadlock-freeness.

A *relabelling* $\lambda : L \to L'$ from an I/O-labelling $L = (I, O, T)$ to an I/O-labelling $L' = (I', O', T')$ consists of three functions $\lambda_I : I \to I'$, $\lambda_O : O \to O'$, and $\lambda_T : T \to T'$. For simplicity, we write $\lambda(l)$ instead of $\lambda_I(l)$ if $l \in I$, and similarly if $l \in O$ or $l \in T$. Let $A = (Q, q_0, \Delta)$ be an I/O-transition system over L and let $\lambda : L \to L'$ be a relabelling. The *relabelling* of A w.r.t. λ is the I/O-transition system $A\lambda$ over L' defined by $A\lambda = (Q, q_0, \Delta\lambda)$ where $\Delta\lambda = \{(q, \lambda(l), q') \mid (q, l, q') \in \Delta \wedge l \neq \tau\} \cup \{(q, \tau, q') \mid (q, \tau, q') \in \Delta\}$. In various cases we will need a simple form of relabelling where the labels of A are just prefixed by a given name, say n. Then we write $n.A$ for the relabelling $A\lambda_n$ with $\lambda_n(l) = n.l$ for all $l \in I$, and similarly for $l \in O$ or $l \in T$ where λ_n is assumed to preserve the kinds of the labels.

The *hiding* of an I/O-labelling $L = (I, O, T)$ w.r.t. a subset $H \subseteq I \cup O \cup T$ is the I/O-labelling $L \setminus H = (I \setminus H, O \setminus H, T \setminus H)$. Let $A = (Q, q_0, \Delta)$ be an I/O-transition system over L and let $H \subseteq \mathit{Label}(A)$. The *hiding* of A w.r.t. H is the I/O-transition system $A \setminus H$ over $L \setminus H$ defined by $A \setminus H = (Q, q_0, \Delta \setminus H)$ where $\Delta \setminus H = \{(q, \tau, q') \mid (q, l, q') \in \Delta \wedge l \in H\} \cup \{(q, l, q') \mid (q, l, q') \in \Delta \wedge l \notin H\}$.

Two I/O-labellings $L_1 = (I_1, O_1, T_1)$ 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 L_1 and L_2, written $L_1 \cap L_2$, are given by $(I_1 \cup O_1 \cup T_1) \cap (I_2 \cup O_2 \cup T_2)$ which, if L_1 and L_2 are composable, is just $(I_1 \cap O_2) \cup (O_1 \cap I_2)$. The *product* of two composable I/O-labellings L_1 and L_2 is the I/O-labelling $L_1 \otimes L_2 = ((I_1 \cup I_2) \setminus (L_1 \cap L_2), (O_1 \cup O_2) \setminus (L_1 \cap L_2), T_1 \cup T_2 \cup (L_1 \cap L_2))$. Let $A_1 = (Q_1, q_1, \Delta_1)$ and $A_2 = (Q_2, q_2, \Delta_2)$ be two I/O-transition systems over composable I/O-labellings L_1, L_2, respectively. The *product* of A_1 and A_2 is the I/O-transition system $A_1 \otimes A_2 = (Q, q_0, \Delta)$ over $L_1 \otimes L_2$ defined as follows: [2]

(i) $Q = Q_1 \times Q_2$;

(ii) $q_0 = (q_1, q_2)$;

(iii) $\Delta = \{((q_1, q_2), l, (q'_1, q_2)) \mid (q_1, l, q'_1) \in \Delta_1 \wedge l \notin L_1 \cap L_2 \wedge q_2 \in Q_2\} \cup$

[2] Intuitively, the product of A_1 and A_2 describes the parallel composition of A_1 and A_2 where coinciding input and output labels are synchronised and then become internal labels.

$$\{((q_1,q_2),l,(q_1,q_2')) \mid (q_2,l,q_2') \in \Delta_2 \wedge l \notin L_1 \cap L_2 \wedge q_1 \in Q_1\} \cup$$
$$\{((q_1,q_2),l,(q_1',q_2')) \mid (q_1,l,q_1') \in \Delta_1 \wedge (q_2,l,q_2') \in \Delta_2 \wedge l \in L_1 \cap L_2\}.$$

In several cases, for synchronisation of A_1 and A_2, some labels of A_1 and of A_2 must first be identified before the product operator is applied. For this purpose we use the synchronised product defined in the following way: Let n_1, n_2 be two names. Then $A_1 \underset{(n_1,n_2)}{\otimes} A_2$ denotes the product $A_1\lambda_1 \otimes A_2\lambda_2$ where the relabelling λ_1 maps each label of A_1 of the form $n_1.l$ to the label $(n_1,n_2).l$ and, similarly, λ_2 maps each label of A_2 of the form $n_2.l$ to the label $(n_1,n_2).l$. Both relabellings are assumed to preserve the I/O nature of the labels, i.e. their inclusion in I, O and T respectively.

All the above operators on I/O-transition systems preserve observational equivalence.

4 Ports, Components and their Behaviours

Building upon ideas from [17], we consider components to be strongly encapsulated behaviours. Encapsulation is achieved by ports which regulate any interaction of components with their environment. Within this section we illustrate our underlying component model by an example. First, we consider the static and then the dynamic aspects of the system architecture. The notation follows our component metamodel defined in [16].

4.1 Static Component Structure

The following example is an extension of the compressing proxy system presented in [4], partly following the proposal to the design and implementation of a proxy server with hybrid data compression facilities in [6]. An HTTP proxy server mediates connections between a web server and clients, like web browsers, which are executing on different machines within a local area network. In order to increase network bandwidth, the proxy server may apply different compression techniques depending on the kind of information transfered. In our example the proxy distinguishes textual (`txt`) from graphical data (`gif`) and applies different compression tools before sending the data further downstream. We model this scenario by a composite component CompressingProxy which is essentially a composition of three simple (i.e. not further decomposed) components Adaptor, GZip and GifToJpg. These components are connected via ports according to the given port declarations as shown in the composite structure diagram in Fig. 1.

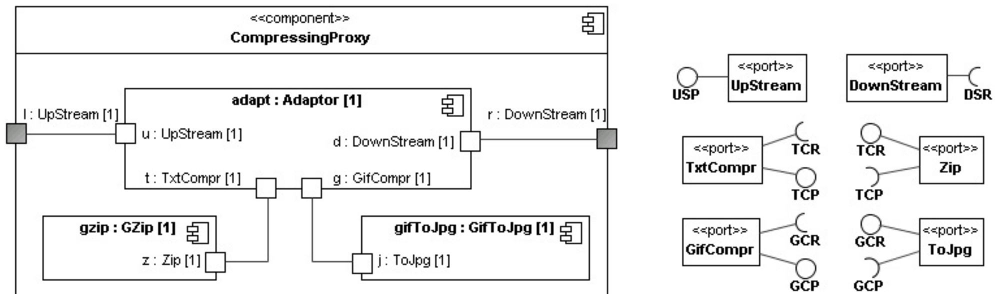


Figure 1. Static structure of a compression proxy server

Port declarations have the form $p : P[m]$ where p is a port name, P a port type and m the port multiplicity indicating how many instances of that port a component (instance)

can have. Port names are locally unique in the component where they are declared. For example, the component Adaptor declares four ports each of them to be instantiated exactly once. These ports can be referred to by their names u, t, g and d. In case of composite components only so-called relay ports are declared which mirror the non-connected ports of subcomponents. The ports l and r, shown grey in Fig. 1, are relay ports of the composite component CompressingProxy. Port types have a provided and a required interface which are depicted in a condensed form, using the ball-and-socket notation of the UML, on the right-hand side of Fig. 1. A provided interface declares a set of (provided) operations that can be called by the user of a component; a required interface consists of a set of (required) operations that are needed for the functioning of a component. In our example the interfaces consist of the following operations: USP (openTxt, openGif, data, close), DSR (compressed), TCR (txt, endTxt), TCP (bufFul,zip), GCR (gif) and finally GCP (jpg).

A composite component comprises of a set of component declarations, a set of (binary) connectors, which connect ports of subcomponents, and a set of relay port declarations. Component declarations have the form $c : C[m]$ where c is a component name, C a component type and m a multiplicity indicating how many instances of that component a composite component (instance) can have. Component names are locally unique in the composite component where they are declared. For instance, the component CompressingProxy declares three subcomponents each of them to be instantiated exactly once. These components can be referred to by their names adapt, gzip and gifToJpg.

Intuitively a proxy of type CompressingProxy receives stream-based data on its port l which is relayed to the port u of the contained component adapt. The adaptor distinguishes the compression of the formats `txt` and `gif`. In the former case adapt directly forwards the stream to be compressed via t to gzip, whereas in the latter case the component waits until it has received all upstream data, and only then it sends the complete image to be compressed via g to gifToJpg. After having received the compression result, adapt sends the data further downstream using port d which is relayed to the port r of the proxy.

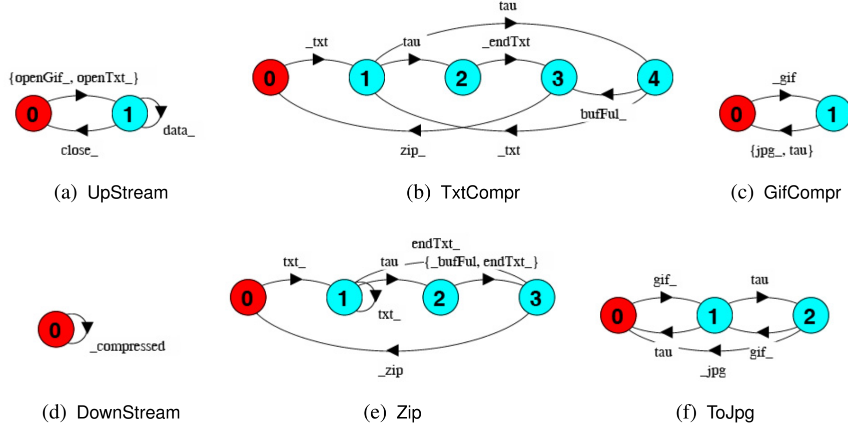
4.2 Port and Component Behaviours

According to our metamodel (see [16]), for each port and for each simple component a behaviour specification must be provided by the component developer. For the formal representation of behaviours we use I/O-transition systems; cf. Sect. 3. The distinction between input, output and internal labels allows, amongst others, for (syntactical) consistency checks during component composition. Due to the abstract nature of transition systems our approach can be used for any concrete syntax of behaviour specifications, like e.g. process algebras or UML state machines, as long as a semantic translation into I/O-transition systems is provided. The graphs, subsequently representing behaviours in our example have been produced with the LTSA tool [13]. We indicate that a label i is an input label by the visual representation $i_$ and, symmetrically, that a label o is an output label by the visual representation $_o$.[3] Furthermore, several transitions between the same two states are shown as a single transition together with the corresponding set of labels.

First, we consider behaviour specifications of ports. As pointed out in Sect. 4.1 a port has a provided and a required interface. Calls of operations of the provided interface of

[3] The LTSA tool supports the process algebra FSP [13] and for the visual representation of the I/O-transition systems we defined appropriate FSP processes. For the reasons explained above the concrete FSP processes are, however, irrelevant here.

a port are represented by input labels; for sending an operation request according to the required interface of a port we use output labels. [4] Since a port represents a window of a component to the outside, a port has no internal labels. Note, however, that transitions with the invisible action τ can still occur to model a possible internal choice (of the port's owner component) which is not visible at the port but may have an impact on the future behaviour of the port. As a concrete example we consider the behaviour specification of the port TxtCompr of the component Adaptor; see Fig. 1. It is given by the I/O-transition system shown in Fig. 2(b) over the I/O-labelling (I, O, T) with $I = \{\text{bufFul}, \text{zip}\}$, $O = \{\text{txt}, \text{endTxt}\}$ and $T = \emptyset$. After having sent an initial part of textual data using the ac-



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

(d) DownStream (e) Zip (f) ToJpg

Figure 2. I/O-transition systems for the ports of the static structure in Fig. 1

tion txt, there is an internal choice between completion and continuation of the current communication. In the former case the signal endTxt is sent as a notification about the completed data stream. Thereafter, the port waits to receive zip to retrieve the particular zip archive. If the internal decision was to continue data compression, the possibility of a buffer overflow needs to be taken into account. In case the connected compression tool signals bufFul, the port is forced to retrieve the current archive via zip. After that, in order to continue the compression of the current data stream, the communication starts again with the output action txt. Otherwise the communication continues directly with sending txt until the data stream is finished or the buffer capacity has been reached.

Next, we consider behaviour specifications of simple components. Since a component can only communicate with its environment via its ports, any input label of a component has the form $p.i$ where p is a port name and i is an input label of the port. Similarly, the output labels of a component have the form $p.o$. [5] In contrast to ports, components can have internal labels which model internal actions of the component. For instance, the behaviour specification of the simple component Adaptor is given by the I/O-transition system shown in Fig. 3. Its I/O-labelling (I, O, T) consists of $I = \{\text{u.openTxt}, \text{u.openGif}, \text{u.close}, \text{u.data}, \text{t.zip}, \text{t.bufFul}, \text{g.jpg}\}$, $O = \{\text{d.compressed}, \text{t.txt}, \text{t.endTxt}, \text{g.gif}\}$ and $T = \{\text{nojpg}\}$.

[4] In our examples the interface operations have no parameters and hence we can use their plain names as labels. In general, for operations with parameters, we can again use their plain names if the actual arguments are not relevant for the behaviour. In the other cases we must assume that the impact of the arguments can be expressed by an appropriate partition of the argument types which then will be reflected by the chosen labels. Of course, for applying model checking techniques this partition must be finitary [7].

[5] For the definition of input and output labels of components we do not take into account here the multiplicities of ports. This is possible if ports are declared with multiplicity 1 (as in our example) or if we assume that actions of different port instances of the same port are independent from each other, i.e. can be arbitrarily interleaved.

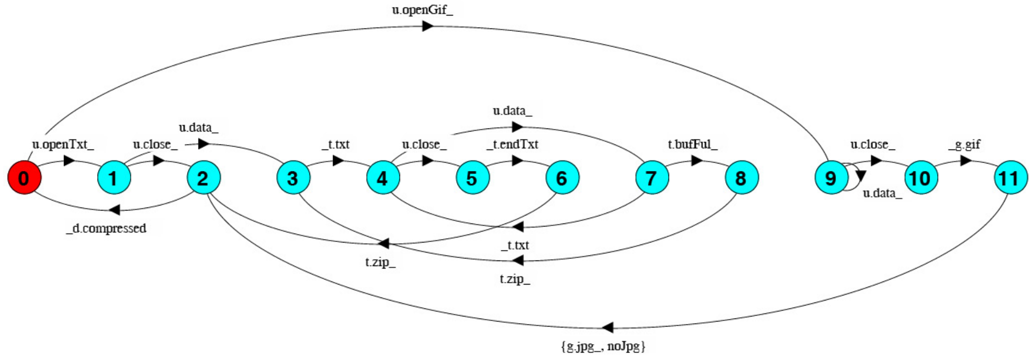


Figure 3. I/O-transition system for component Adaptor

The behaviour specification introduces an internal transition (11, noJpg, 2), refining the τ-transition of the port behaviour in Fig. 2(c). Notice that the transition's trigger is not further specified here. One may think, for instance, of a loop in state 11 such that the component waits only a fixed amount of time for the compression of `gif` data. If the compression takes too long, the adaptor sends the original data further downstream instead of waiting for the compression result. In this case the internal transition noJpg would be fired. In fact, internal decisions like this are often forgotten to be specified in the external behaviour of a component, yielding unexpected interaction behaviour of this component. Our notion of component correctness together with an interaction analysis aims to reveal erroneous situations of this kind.

For lack of space we do not show behaviour specifications of the simple components GZip and GifToJpg; since these components are equipped with one port only, their behaviour is not too much different from the behaviour of their ports (Fig. 2(e), 2(f)).

5 Analysis of Simple Components

In the first step of our model analysis we consider simple components which are the basic building blocks of our system model. For each simple component we check the deadlock-freeness of the behaviour specification of each of its ports and the deadlock-freeness of the behaviour specification of the component itself. Obviously, this condition is satisfied for all simple components and ports of our example.

A more subtle point concerns the relationships between the behaviour of a component and the behaviour specified for each of its ports which must in some sense fit together. To consider this issue more closely, let C be a component with associated behaviour represented by the I/O-transition system A_C and let $p : P$ be a port declaration [6] of C such that the behaviour specification associated to P is represented by the I/O-transition system A_P. Intuitively, the component C is correct w.r.t. its port declaration $p : P$ if the component behaviour supports the behaviour specified for that port. Apparently this is the case if the component observable at port p is observationally equivalent to the behaviour specification of P (up to an appropriate relabelling).

Formally, the *observable behaviour of C at port p*, denoted by $obs_p(C)$, can be constructed by hiding all labels of A_C which do not refer to p. Using the hiding operator (see Sect. 3), $obs_p(C) = A_C \setminus H$ where H is the set of internal labels of A_C together with all

[6] As explained in Sect. 4.2 we disregard port multiplicities here.

input or output labels $q.op$ of A_C such that $q \neq p$. Since the transition system $obs_p(C)$ has no internal labels and, up to the prefix p, the same input and output labels as A_P we can now require that it is observationally equivalent to $p.A_P$ (which is the copy of A_P, see Sect. 3). These considerations lead to the following definition of component correctness.

Definition 5.1 Let C be a component and A_C the I/O-transition system representing the behaviour of C. Let $p : P$ be a port declaration of C and A_P the I/O-transition system representing the behaviour specification of P. The component C is *correct w.r.t. its port declaration* $p : P$ if $obs_p(C) \approx p.A_P$. The component C is *correct*, if it is correct w.r.t. all its port declarations.

Let us illustrate how we can check the correctness of the component Adaptor w.r.t. its port t : TxtCompr. First we consider the observable behaviour $obs_{\mathsf{t}}(\mathsf{Adaptor})$ of the Adaptor at port t, which is just the transition system shown in Fig. 3 where all labels which are not prefixed by t are replaced by τ. If we minimise this transition system w.r.t. observational equivalence then we obtain (up to the prefix t) the transition system in Fig. 2(b) which represents the behaviour of the port type TxtCompr. This shows the correctness of the Adaptor component w.r.t. its port t : TxtCompr. In fact, using for instance the LTSA tool one can check that the component Adaptor is correct w.r.t. all of its ports.

The definition of the observable behaviour of a component at a particular port can be generalised in a straightforward way to arbitrary subsets of the port declarations of a component. Given a component C and some port declarations $p_1 : P_1, \ldots, p_k : P_k$ of C, the observable behaviour of C at ports $p_1, \ldots, p_k$ is denoted by $obs_{p_1,\ldots,p_k}(C)$. In the special case where $p_1 : P_1, \ldots, p_k : P_k$ consists of all port declarations of C, $obs_{p_1,\ldots,p_k}(C)$ is called the *observable behaviour* of C and simply denoted by $obs(C)$. Obviously, observable behaviours of components at particular ports can be constructed by successively reducing the observed ports, e.g., $obs_p(C) \approx obs_p(obs(C))$. This can be particularly useful for verification, since after each reduction step a minimisation of the resulting observable behaviour can be computed which may drastically reduce the state space. In particular, for proving the correctness of a component at some port it may be more efficient to prove $obs_p(obs(C)) \approx p.A_P$ (with A_P as in Def. 5.1).

Note that the above definitions of correctness and observable behaviour apply not only to simple components but also to composite components considered in the next section.

6 Analysis of Composite Components

The analysis of composite components is related to the task of a system architect who puts components together to build larger ones. A crucial aspect of our method is that before analysing the behaviour of a composite component we first examine the connections that have been established between the ports of their subcomponents.

6.1 Analysis of Connectors

For the analysis of connectors one has first to check whether the connections between the ports of components are syntactically well-defined. After that we can analyse the interaction behaviour of two connected ports.

In the following let us consider a connection between two port declarations $p_1 : P_1$ and

$p_2 : P_2$ occurring in components C_1 and C_2 resp. The connection is syntactically well-defined, if the operations of the required interface of P_1 coincide with the operations of the provided interface of P_2 and conversely.[7] To study the interaction behaviour of the two ports, let A_{P_1} be the I/O-transition system over the I/O-labelling $(I_1, O_1, \emptyset)$ representing the behaviour of P_1 and let A_{P_2} be the I/O-transition system over the I/O-labelling $(I_2, O_2, \emptyset)$ representing the behaviour of P_2. According to the syntactic well-formedness condition, $O_1 = I_2$ and $O_2 = I_1$. Any communication between the connected ports is expressed by synchronising output labels of one port with the corresponding input label of the other port according to the possible transitions of A_{P_1} and A_{P_2}. Hence, the interaction behaviour of A_{P_1} and A_{P_2} can be formally represented by the *port product* $A_{P_1} \otimes A_{P_2}$ where "$\otimes$" is the product operator defined in Sect. 3. Note that the transitions of the port product are marked only by internal labels (representing interactions) or by the invisible τ-action.

A first semantic condition which should be required for a port connection is that any two port instances can communicate with each other without the possibility to run into a deadlock. Since the interaction behaviour of two ports is represented by the transition system of the port product $A_{P_1} \otimes A_{P_2}$ this condition can be formalised as follows leading to the notion of behavioural port compatibility.

Definition 6.1 Two ports P_1 and P_2 with behaviours represented by the I/O-transition systems A_{P_1}, A_{P_2} resp. are *behaviourally compatible* if $A_{P_1} \otimes A_{P_2}$ is deadlock-free.

Figure 4 shows the port product of the ports TxtCompr and Zip which does not deadlock. Hence both ports are behaviourally compatible.

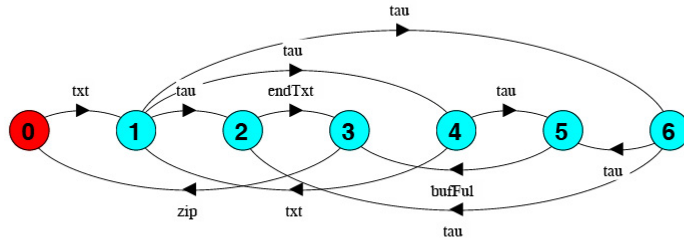


Figure 4. Port product of TxtCompr and Zip

In general, the potential capabilities for interaction of a port will not be used when the port is connected to another port. In this case the behaviour specified for that port is restricted by the interaction with another port. It is, however, often the case that this restriction applies only to one side of a connection while the behaviour of the port on the other side is not restricted and hence fully reflected by the interaction. This property plays an essential role for the compositionality of behaviours that will be studied below to analyse behaviours of composite components. It can be formalised in the following way.

Definition 6.2 Let $A_{P_1} \otimes A_{P_2}$ be the port product representing the interaction behaviour of two ports P_1 and P_2 (whose single behaviours are represented by the I/O-transition systems A_{P_1}, A_{P_2} resp.). The interaction behaviour of P_1 and P_2 *reflects* the behaviour of P_1, if $A_{P_1} \approx A_{P_1} \otimes A_{P_2}$ where A_{P_1} on the lefthand-side is considered as an I/O-transition system with all labels being internal (in order to fit to the labelling of the port product).

[7] In general, one could use a more flexible condition such that the required operations of one port are included in the provided operations of the other one. However, it is technically more convenient and also sufficient for the example to use the more restrictive condition from above.

Considering our example, we have verified, using LTSA, that the interaction behaviour of the two ports GifCompr and ToJpg reflects the behaviour of GifCompr. Obviously, this is not the case for the ports TxtCompr and Zip.

6.2 *Behaviour of Composite Components*

In contrast to simple components the behaviour of a composite component is not explicitly specified by the developer but is derived from the behaviours of the single parts of the composite component. For the behaviour of a composite component we construct the product of the observable behaviours of all of its subcomponents which must be synchronised according to the given port connectors. That is, we focus on the interactions between the subcomponents (via their connected ports) and on the actions on the relay ports. The internal behaviour of the subcomponents is not relevant. How this construction is technically performed for the case of a composite component with two connected subcomponents and with one relay port is discussed in the following. The construction can be generalised in a straightforward way to arbitrary many subcomponents, connectors and relay ports.

Let CC be a composite component which contains two component declarations $c_1 : C_1$ and $c_2 : C_2$. As for ports we omit multiplicities of component declarations and therefore assume that subcomponents are either declared with multiplicity 1 (as in our example) or the actions of different component instances of the same component declaration are independent from each other.[8] Assume that C_1 contains two port declarations $r : R$, $p_1 : P_1$ and C_2 contains one port declaration $p_2 : P_2$ such that $p_1 : P_1$ and $p_2 : P_2$ are connected and the connection is syntactically well-defined (i.e. the output labels of P_1 correspond to input labels of P_2 and vice versa). Moreover, assume that CC has one relay port declaration $rp : RP$ which refers to the port declaration $r : R$ of C_1. We assume that the relay port reference is syntactically well-defined, i.e. that the port type RP has the same provided and required interfaces as the port type R. Note, however, that in general RP and R may have different behaviour specifications as discussed when we analyse the correctness of composite components below.

Let $obs(C_1)$ and $obs(C_2)$ be the I/O-transition systems representing the observable behaviours of C_1 and C_2 resp.; cf. Sect. 4.2. We will construct an I/O-transition system A_{CC} representing the behaviour of the composite component CC. First, let $c_1.obs(C_1)$ be a copy of $obs(C_1)$ obtained by prefixing each label of $obs(C_1)$ with c_1 (cf. Sect. 3) and, similarly, let $c_2.obs(C_2)$ be a copy of $obs(C_2)$ using prefix c_2. Obviously, since component names are locally unique, the labels of $c_1.obs(C_1)$ and $c_2.obs(C_2)$ are disjoint. To construct A_{CC} we have to synchronise the given transition systems according to the connection of their ports which is easily achieved by using the construct for synchronised products (cf. Sect. 3) such that, for any label op of the port P_1 (and hence of P_2), the labels $c_1.p_1.op$ and $c_2.p_2.op$ are identified via the shared label $(c_1.p_1, c_2.p_2).op$. Finally, the labels expressing actions of c_1 on the non-connected port r must be renamed to actions on the relay port using the relabelling $\lambda_{relay}(c_1.r.op) = rp.op$ for all labels op of R (and hence of RP). The I/O-transition system A_{CC} representing the behaviour of the composite component CC is then given by $A_{CC} = (c_1.obs(C_1) \underset{(c_1.p_1, c_2.p_2)}{\otimes} c_2.obs(C_2))\lambda_r$.

[8] If they would be dependent they could be simulated by introducing as many additional component declarations of the same type as instances of the original declaration are involved, provided that no reconfiguration occurs.

Of course, one may again construct the observable behaviour $obs(A_{CC})$ of a composite component which then could be used for analysis, on the one hand, and for the construction of the behaviour of another composite component on the next hierarchy level on the other hand. Considering our example, the I/O-transition system representing the behaviour of the composite component CompressingProxy has 14 states and 21 transitions which are not shown here. As Fig. 5 illustrates, constructing the observable behaviour of CompressingProxy allows for an efficient minimisation w.r.t observational equivalence.

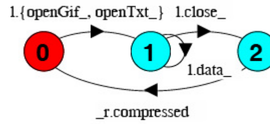


Figure 5. Observable behaviour of CompressingProxy

In the following we focus on checking the deadlock-freeness of the behaviour of a composite component. It is well-known that, in general, the deadlock-freeness of subcomponents does not guarantee the deadlock-freeness of a global system (as nicely illustrated by Dijkstra's philosophers example). Indeed this is unfortunately still the case if all components are correct w.r.t. their ports (in the sense from above) and if all ports are connected in a behaviourally compatible way, as soon as more than two subcomponents are involved. Hence, we are looking for particular topologies of component structures where deadlock-freeness is preserved. An appropriate candidate are (acyclic) star topologies as shown in Fig. 6 containing one central component C with n ports such that each port is connected to the port of one of the components C_i for $i = 1, \ldots, n$.

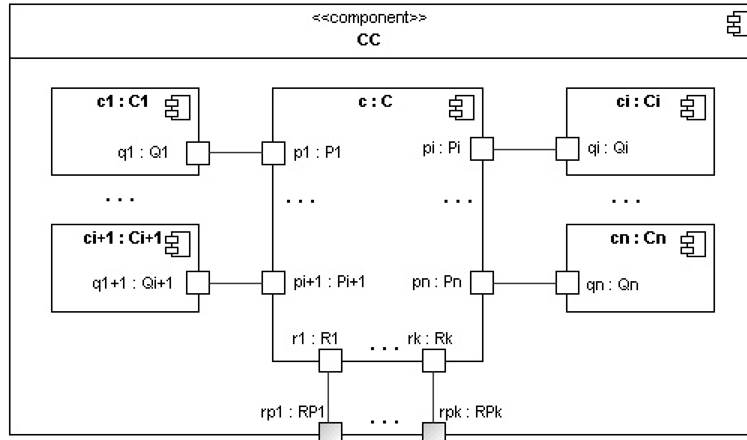


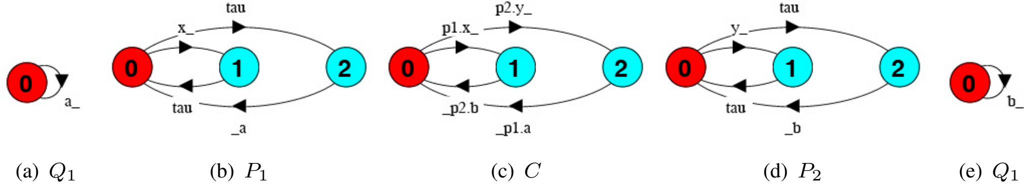
Figure 6. Star topology of composite component

We assume that all subcomponents are correct (w.r.t. their ports) and that their local behaviours are deadlock-free. Then, if all connected ports are behaviourally compatible, the composite component CC can only deadlock if at least two ports $p_\alpha : P_\alpha$, $p_\beta : P_\beta$ of the central component C are connected to ports $q_\alpha : Q_\alpha$, $q_\beta : Q_\beta$ of components C_α and C_β resp. such that the behaviour specifications of both port types Q_α and Q_β properly restrict the behaviours specified for P_α, P_β in a way which is incompatible with the behaviour of C.[9] This may happen, if C introduces a dependency between P_α and P_β

[9] Note that it is sufficient to consider the behaviours of the *ports* of C_α and C_β instead of considering the observable behaviour of the full components C_α and C_β since both components are assumed to be correct w.r.t. their respective ports.

that is incompatible with the simultaneous restrictions imposed by the connections with Q_α and Q_β on both sides of C. The following example illustrates this situation.

Example 6.3 We consider the component structure of Fig. 6 for the case $n = 2$, i.e. there are subcomponents C, C_1 and C_2. The behaviour specifications of C and of the port types P_1, P_2, Q_1 and Q_2 are shown in Fig. 7.



(a) Q_1 (b) P_1 (c) C (d) P_2 (e) Q_1

Figure 7. Behaviour specifications for component C with ports P_1, P_2, Q_1 and Q_2.

Obviously, all components are correct and all behaviours are deadlock-free. To study the interaction behaviour of the two ports P_1 and Q_1 we assume that Q_1 has the input label a and the output label x though there is never a transition with x. Then their interaction behaviour is represented by the port product $P_1 \otimes Q_1$ which properly restricts the behaviour of P_1 but still performs the infinite sequence of "τ a" actions and hence does not deadlock. Similarly, the interaction behaviour of the ports P_2 and Q_2 properly restricts the behaviour of P_2 but still performs the infinite sequence of "τ b" actions and hence does also not deadlock. However, the behaviour of the central component C intertwines the actions of its ports P_1 and P_2 in such a way that after the connection to Q_1 and Q_2 a deadlock occurs immediately (since Q_1 never sends x and Q_2 never sends y).

In order to exclude situations as illustrated by the example, we assume that for at most one port of the central component C it may happen that its behaviour specification is restricted by the actual connection (but the port interaction is still deadlock-free), and that for all other ports of C their behaviour specifications are reflected by the actual connection. Then local deadlock-freeness is preserved by the component composition. This fact is expressed by the corollary given below which shows that indeed for the global deadlock check it is enough if the subcomponents are locally checked for deadlock-freeness and correctness and if the architect of the composite component checks each single port connection on the basis of the interaction behaviour of the connected ports. The corollary is a consequence of the following theorem which considers the general case, where an arbitrary number of connections can exist such that the behaviour specifications of the ports of the central component are not reflected by the port connections (but the port interaction is still deadlock-free).In this case, the difficulty is that it is no more sufficient to consider only the behaviour specifications of ports but one has to consider the observable behaviour of the whole central component C at all ports that are not connected in a behaviour reflecting way.

Theorem 6.4 *Let CC be a composite component with component structure as shown in Fig. 6. Let the following conditions be satisfied:*

(i) *All components $C_1, \ldots, C_n$ are correct and C is deadlock-free and correct with respect to the port declarations $p_1 : P_1, \ldots, p_n : P_n$.* [10]

[10] We have not explicitly required that all $C_1, \ldots, C_n$ are deadlock-free, because, for $j = i + 1, \ldots, n$ this follows from their correctness and from condition (iii) and for $j = 0, \ldots, i$ this is not even necessary.

(ii) *For all* $j \in \{1, \dots, i\}$ *the interaction behaviour of* P_j *and* Q_j *reflects the behaviour of* P_j.

(iii) $c.obs_{p_{i+1},\dots,p_n}(C) \underset{(c.p_{i+1},c_{i+1}.q_{i+1})}{\otimes} Q^{(i+1)}$ *with*

$$Q^{(i+1)} = c_{i+1}.q_{i+1}.Q_{i+1} \underset{(c.p_{i+2},c_{i+2}.q_{i+2})}{\otimes} \cdots \underset{(c.p_n,c_n.q_n)}{\otimes} c_n.q_n.Q_n \ ,$$

is deadlock-free, where the synchronised product is computed from left to right and, for all j, Q_j *denotes, by abuse of notation, the I/O-transition system representing the behaviour specification of the port type* Q_j.

Then the I/O-transition system representing the behaviour of CC *is deadlock-free.*

Proof We assume, w.l.o.g, that CC has exactly one relay port $rp : RP$ referring to port $r : R$ of C. By condition (ii) we know that for $j = 1, \dots, i$, $P_j \approx P_j \otimes Q_j$ where, for simplicity, P_j, Q_j denote the I/O-transition systems representing the behaviour specifications of the respective ports. Obviously, since C is correct,

$$c.obs_{p_{i+1},\dots,p_n}(C) \approx c.obs(C) \underset{(c,c_1)}{\otimes} c_1.P_1 \underset{(c,c_2)}{\otimes} \cdots \underset{(c,c_i)}{\otimes} c_i.P_i \ .$$

Then, by condition (ii) and since the product preserves "$\approx$", P_j can be replaced by $P_j \otimes Q_j$, which, within the context of $obs(C)$, can in turn be replaced by Q_j for $j = 1, \dots, i$:

$$(*) \quad c.obs_{r,p_{i+1},\dots,p_n}(C) \approx c.obs(C) \underset{(c.p_1,c_1.q_1)}{\otimes} c_1.q_1.Q_1 \underset{(c.p_2,c_2.q_2)}{\otimes} \cdots \underset{(c.p_i,c_i.q_i)}{\otimes} c_i.q_i.Q_i \ .$$

Again by the compatibility of the product with "$\approx$", we then obtain from (*)

$$c.obs_{r,p_{i+1},\dots,p_n}(C) \underset{(c.p_{i+1},c_{i+1}.q_{i+1})}{\otimes} Q^{(i+1)} \approx$$
$$c.obs(C) \underset{(c.p_1,c_1.q_1)}{\otimes} c_1.q_1.Q_1 \underset{(c.p_2,c_2.q_2)}{\otimes} \cdots \underset{(c.p_n,c_n.q_n)}{\otimes} c_n.q_n.Q_n \ .$$

By condition (iii), the lefthand side of the equivalence is deadlock-free and hence, since "$\approx$" is compatible with deadlock-freeness, also the righthand side is deadlock-free. Then, since by condition (i) all components $C_1, \dots, C_n$ are correct, the righthand side from above is observationally equivalent to

$$c.obs(C) \underset{(c.p_1,c_1.q_1)}{\otimes} c_1.obs(C_1) \underset{(c.p_2,c_2.q_2)}{\otimes} \cdots \underset{(c.p_n,c_n.q_n)}{\otimes} c_n.obs(C_n) \ .$$

which then in turn is deadlock-free. If we finally apply the relabelling $\lambda_r(c.r.op) = rp.op$ for all labels op of R (and hence of RP) then we obtain an I/O-transition system which just represents the behaviour of CC and, since relabelling preserves deadlock-freeness, the result is also deadlock-free. □

Corollary 6.5 *Let* CC *be a composite component as in Theorem 6.4 such that the conditions (i) of Theorem 6.4 is satisfied.*

(a) If the interaction behaviour of P_j *and* Q_j *reflects the behaviour of* P_j *for all* $j \in \{1, \dots, n\}$, *then the I/O-transition system representing the behaviour of* CC *is deadlock-free.*

(b) If there exists $k \in \{1, \dots, n\}$ such that P_j and Q_j reflects the behaviour of P_j for all $j \in \{1, \dots, n\}$ with $j \neq k$ and the ports P_k and Q_k are behaviourally compatible, then the I/O-transition system representing the behaviour of CC is deadlock-free.

Proof For (a), condition (ii) of Thm. 6.4 is satisfied for $i = n$ and condition (iii) is trivially true. The result follows then from Thm. 6.4.

For (b), let w.l.o.g. $k = n$. Then condition (ii) of Thm. 6.4 is satisfied for $i = n - 1$. Since, by assumption, the port product $P_n \otimes Q_n$ is deadlock-free and since, by correctness of C, $obs_{p_n}(C) \approx p_n.P_n$, $c.obs_{p_n}(C) \underset{(c.p_n, c_n.q_n)}{\otimes} q_n.Q_n$ is deadlock-free as well. Hence condition (iii) of Thm. 6.4 is satisfied and the desired result follows from Thm. 6.4. □

Part (a) of the corollary is related to a similar result in [4] where, however, no explicit port concept is considered. Part (b) can directly be applied to our example since all subcomponents are correct and the behaviour of the Adaptor component (and also of the other subcomponents) is deadlock-free. Moreover, we know from Sect. 6.1 that the interaction behaviour of the two ports GifCompr and ToJpg reflects the behaviour of GifCompr and that the ports TxtCompr and Zip are behaviourally compatible. Hence, the behaviour of the composite component CompressingProxy is deadlock-free.

6.3 Correctness of Composite Components

An important aspect of the analysis of composite components concerns their correctness which is needed to apply iteratively our results when climbing up the component hierarchy. In this case the question is whether the given behaviour specification of each relay port coincides, up to observational equivalence, with the behaviour of the composite component observable at the relay port. Considering again the component structure of Fig. 6, one may even expect that for the behaviour specification of a relay port RP_j of CC one could reuse the behaviour specification of the referenced port R_j of the subcomponent C as it is indeed the case in our example. In general, this is however only true if all port connections inside the composite component reflect the behaviour of the ports as assumed in Corollary 6.5(a). In all other cases it may happen that a non-behaviour reflecting (but still behaviourally compatible) port connection restricts the behaviour of the central component C in a way which influences also the actual behaviour visible at some port $r_j : R_j$ referenced by a relay port $rp_j : RP_j$. Hence, for the composite component to be correct, the behaviour specification of the relay port (type) RP_j of CC may be a restriction of the behaviour specification of the referenced port (type) R_j of C. Unfortunately, for lack of space, we cannot provide an example here which illustrates this situation. The following theorem provides general conditions for the correctness of composite components whose structure conforms to the star topology.

Theorem 6.6 *Let CC be a composite component with component structure as shown in Fig. 6 and let $rp : RP$ be a relay port of CC (referring to port $r : R$ of C). Let the following conditions be satisfied:*

(i) *All components $C_1, \dots, C_n$ are correct.*

(ii) *For all $j \in \{1, \dots, i\}$ the interaction behaviour of P_j and Q_j reflects the behaviour of P_j.*

(iii) $obs_{rp}((c.obs_{r,p_{i+1},\dots,p_n}(C) \underset{(c.p_{i+1},c_{i+1}.q_{i+1})}{\otimes} Q^{(i+1)})\lambda_r) \approx rp.RP$ *with*

$$Q^{(i+1)} = c_{i+1}.q_{i+1}.Q_{i+1} \underset{(c.p_{i+2},c_{i+2}.q_{i+2})}{\otimes} \cdots \underset{(c.p_n,c_n.q_n)}{\otimes} c_n.q_n.Q_n\ ,$$

where RP denotes, by abuse of notation, the I/O-transition system representing the behaviour specification of the port type RP and $\lambda_r(c.r.op) = rp.op$ for all labels op of R (and hence of RP).

Then CC is correct w.r.t. its relay port $rp : RP$.

Proof We assume, w.l.o.g, that $rp : RP$ is the only relay port of CC. From the proof of Thm. 6.4 we know that

$$c.obs_{r,p_{i+1},\dots,p_n}(C) \underset{(c.p_{i+1},c_{i+1}.q_{i+1})}{\otimes} Q^{(i+1)} \approx$$
$$c.obs(C) \underset{(c.p_1,c_1.q_1)}{\otimes} c_1.obs(C_1) \underset{(c.p_2,c_2.q_2)}{\otimes} \cdots \underset{(c.p_n,c_n.q_n)}{\otimes} c_n.obs(C_n)\ .$$

We apply now on both sides of the equivalence the relabelling λ_r which preserves "$\approx$". The right hand side is then the I/O-transition system representing the behaviour of CC which, for simplicity, will be also denoted by CC. In a second step we construct on both sides the observable behaviour at port rp by applying obs_{rp}. Then the lefthand side coincides with the transition system in condition (iii) and the transition system on the righthand side is $obs_{rp}(CC)$. Hence, by condition (iii) and transitivity of "$\approx$", we obtain $obs_{rp}(CC) \approx rp.RP$. Thus CC is correct w.r.t. $rp : RP$. □

Coming back to our example we know that condition (i) is satisfied and, concerning condition (ii), that the interaction behaviour of the two ports GifCompr and ToJpg reflects the behaviour of GifCompr. Hence, the condition (iii) to be proved for the correctness of CompressingProxy w.r.t. the relay port l : UpStream is that

$$obs_l((obs_{\mathsf{u,t}}(\mathsf{adapt.Adaptor}) \underset{(\mathsf{adapt.t,gzip.z})}{\otimes} \mathsf{gzip.z.Zip})\lambda_u) \approx \mathsf{l.Up}\ .$$

This condition can again be verified with the LTSA tool. Similarly, we can show that CompressingProxy is correct w.r.t. its relay port r : Down.

7 Conclusions

We have studied in detail the tasks to be carried out when we want to prove deadlock-freeness and correctness of large scale components. From the methodological point of view our main principle is to split the verification tasks along the modular and hierarchical structure of a component deriving global properties from local ones. Technically this approach is supported by the introduction of explicit ports with user-defined protocols, hierarchical structuring operators, and by exploiting the observational equivalence (weak bisimulation) concept thus revealing how powerful this concept can be in more practical applications. Our results provide the source for the investigation of several open issues concerning e.g. run-time reconfigurations of networks of component instances and extensions to more general system topologies.

Acknowledgement

This work is motivated by questions that appeared in the context of the common component modelling example CoCoME [16]. We would like to thank the organisers of this initiative for providing the motivation and Hubert Baumeister, Florian Hacklinger and Martin Wirsing for their collaboration. Special thanks to Mila Majster-Cederbaum for many valuable comments and remarks during the exploration phase of the present work.

References

[1] Alessandro Aldini and Marco Bernardo. A General Approach to Deadlock Freedom Verification for Software Architectures. In Keijiro Araki, Stefania Gnesi, and Dino Mandrioli, editors, *Proc. 12th Int. Symp. Formal Methods Europe (FME'03)*, volume 2805 of *Lect. Notes Comp. Sci.*, pages 658–677. Springer, 2003.

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

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

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

[5] Cyril Carrez, Alessandro Fantechi, and Elie Najm. Behavioural Contracts for a Sound Composition of Components. In Hartmut König, Monika Heiner, and Adam Wolisz, editors, *Proc. 23rd IFIP Int. Conf. Formal Techniques for Networked and Distributed Systems (FORTE'03)*, volume 2767 of *Lect. Notes Comp. Sci.*, pages 111–126. Springer, 2003.

[6] Chi-Hung Chi, Jing Deng, and Yan-Hong Lim. Compression Proxy Server: Design and Implementation. In *USENIX Symp. Internet Technologies and Systems*, 1999.

[7] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, 2000.

[8] Luca de Alfaro and Thomas A. Henzinger. Interface Automata. In *Proc. 9th ACM SIGSOFT Ann. Symp. Foundations of Software Engineering (FSE'01)*, pages 109–120, Wien, 2001. ACM Press.

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

[10] Gregor Gössler, Susanne Graf, Mila Majster-Cederbaum, M. Martens, and Joseph Sifakis. An approach to modeling and verification of component based systems. In *Current Trends in Theory and Practice of Computer Science, SOFSEM'07*, number 4362 in LNCS, 2007.

[11] Gregor Gössler, Susanne Graf, Mila Majster-Cederbaum, Moritz Martens, and Joseph Sifakis. Ensuring Properties of Interaction Systems by Construction. In *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.

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

[13] Jeff Magee and Jeff Kramer. *Concurrency — State Models and Java Programs*. John Wiley & Sons, 1999.

[14] Pavel Parízek and František Plášil. Modeling environment for component model checking from hierarchical architecture. In *Third International Workshop on Formal Aspects of Component Software (FACS 2006)*, Electronic Notes in Theoretical Computer Science, 2006.

[15] Frantisek Plasil and Stanislav Visnovsky. Behavior protocols for software components. *IEEE Transactions on Software Engineering*, 28(11):1056–1076, 2002.

[16] Ralf Reussner, Klaus Krogmann, Heiko Koziolek, Andreas Rausch, Sebastian Herold, Holger Klus, Yannick Welsch, Benjamin Hummel, Michael Meisinger, Christian Pfaller, and Raffaela Mirandola. *CoCoME Book*, chapter 3, CoCoME — The Common Component Modelling Example. Springer, 2007.

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

[18] Ivana Černá, Pavlína Vařeková, and Barbora Zimmerova. Component Substitutability via Equivalencies of Component-Interaction Automata. In *Proc. Wsh. Formal Aspects of Component Software (FACS'06)*, pages 115–130, Praha, 2006. UNU-IIST Technical Report 344.

[19] Daniel M. Yellin and Robert E. Strom. Protocol Specifications and Component Adaptors. *ACM Trans. Program. Lang. Syst.*, 19(2):292–333, 1997.