An ADL-Approach to Specifying and Analyzing Centralized-Mode Architectural Connection

Guoxin Su[1], Mingsheng Ying[1,2], and Chengqi Zhang[1]

[1] Centre for Quantum Computation and Intelligent Systems, Faculty of Engineering and Information Technology, University of Technology, Sydney, NSW 2007, Australia
[2] State Key Laboratory of Intelligent Technology and Systems, Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China
{guoxin,mying,chengqi}@it.uts.edu.au

Abstract. A rigorous paradigm coordinating components is important in the design stage of large-scale software engineering. In this paper we propose a new Architecture Description Language, called ACDL, to represent the centralized-mode architectural connection in which all components are linked by a single connector. Following one usual approach to architectural description, in which component types and components are distinguished, and connectors integrate behaviors of components by specifying their coordination protocols, ACDL describes connectors in such a way that connectors are insensitive to the numbers of attached same-type components. Based on ACDL, we develop analytic techniques to facilitate the system checking of temporal properties of an architecture. In particular, our method shows to what extent one can add, delete and replace components without making the whole system lose desired temporal properties, and improves the system checking in several ways, for example enhancing the use of previous checking results to deal with new checking problems.

1 Introduction

As the complexity of software designs increases, apart from algorithmic and data-structure-related problems, attention is focused on how to compose subsystems into an overall system [1]. A rigorous paradigm coordinating components is important in the design stage of large-scale software engineering.

Many approaches exist in the literature, from application-oriented to theory-emphasized, to deal with issues related to component-based engineering [2]. Architecture Description Languages (ADLs) emerged as a promising way to formally describe some essential features of an architecture. Although the software architecture community agrees, more or less, that a description of an architecture should consist of three parts, i.e. components, connectors, and architectural configuration [3], each ADL has its own modeling focus, fleshing out features of an architecture from its own viewpoint. We consider components as interfaces performing running-time behaviors, i.e. sequences of input, output and internal actions, and connectors as a special kind of components whose functionality is to integrate components, and whose interfaces can be seen as protocols coordinating behaviors of components. Similar understanding of components and connectors can be found in ADLs such as Wright [4] and π-ADL [5].

M. Ali Babar and I. Gorton (Eds.): ECSA 2010, LNCS 6285, pp. 8–23, 2010.

In this paper we propose a new ADL, called ACDL (an acronym for Architectural Connection Description Language), in which component types and components are distinguished, and connectors are described to be insensitive to the numbers of attached same-type components. ACDL provides a suitable formal specification for both the structural and the behavioral features of centralized-model architectural connection in which components are linked by a single connector. Centralized-mode architectural connection emphasizes the central status of connectors in star-topology architectures. Advantages of such architectural topology have been recognized in the literature, such as [6] in which it was called coordinator-based architecture style.

Based on ACDL, we develop analytic techniques to facilitate the system checking of temporal properties of an architecture. The compositional analyses use a partition to divide the whole set of components in an architecture into parts, and allow to check each part against the central connector to obtain the correctness of the architecture. The type-based analyses allow to do the checking on the architecture-type level instead of the individual-architecture level, and show to what extent one can add, delete and replace components without making the whole system lose desired temporal properties. Together, these techniques can improve the system checking in four ways:

- Our method enhances the use of previous checking results to deal with new checking problems;
- It helps identify the part of an architecture leading to an undesired property;
- It reduces the complexity of checking by safely skipping over some components;
- It facilitates the reusability of ACDL-specifications by showing to what extent the system checking can be carried out in the type level.

1.1 Novelty

This paper is novel in the way that ACDL describes architectures, in particular, connectors. The idea that connectors integrate components by specifying the coordination protocols for their behaviors is not new, but connectors described in ACDL are structurally flexible in the sense that protocols implemented in them have no restriction on the numbers of attached same-type components. This structural flexibility of connectors is achieved by allowing some components to send information to inform the connector what components are involved in the interactions. Therefore ACDL need not distinguish connector types and instances as Wright does. The formal descriptions of connectors in ACDL provides the centralized-mode architectural connection a generic representation, which is important both in theory and in practice (see Sect. 2).

Another innovation is the analytic techniques of temporal properties of an architecture, which are developed based on ACDL. By employing π-calculus [7] to be its formal semantics, ACDL allows reasoning about temporal properties of the system. We show how it deals with deadlock-freedom and an important liveness property called interaction-liveness. Interaction-liveness formulates the property of a system that, at each stage during the running-time of the system, each component is able to get involved into the interaction with the rest of the architecture at some future time, or alternatively, the system will never proceed to a situation in which some of its components can no longer interact with the environment. The idea of using Process Algebras reasoning about properties of an architecture is not new and has been carried out in the previous literature, such as [4] and [8]. But the main novelty of our method is that, firstly, more general than the acyclic/cyclic sharp division in [8], it uses a partition on the whole set of components in architecture to achieve the finest-grain of the compositional analyses, and secondly, it allows to do the checking on the architecture-type level and shows to what extent one can add, delete and replace components in an architecture without making the system lose the desired properties. On the other hand, although some ADL-relevant works like [9] and [10] did indicate that their methods apply to the analysis of liveness properties, ours, which seriously deals with interaction-liveness, is still enlightening.

1.2 Other Related Works

The architectural topology that we consider, i.e. centralized-mode architectural connection, is close to the coordinator-based architecture style investigated in [6], in which the authors were motivated by the following problem: how to assemble a set of off-the-shelf software components into an overall system which enjoys desired properties. They achieved this goal by delegating the interactions of components to a single coordinator which restricts the interaction-patterns of components.

Related ADLs includes Darwin [11], which also employs π-calculus as its semantics. But Darwin considers components as interfaces for providing and requesting (references of) services, and does not explicitly model a connector as a first-class entity in an architecture. π-ADL [5] is a powerful formal specification language based on the high-order typed π-calculus, and is equipped with the analysis language π-AAL [12] which is able to express safety and other temporal properties. However, despite of their expressive and analytic power, π-ADL and π-AAL do not aim to facilitate the system checking and the reusability of specifications by providing a suitable representation of centralized-mode architectural connection, which is the primary goal of our approach.

The remainder of this paper is organized as follows: In Sect. 2 we use client-server systems as examples to motivate our modeling approach. In Sect. 3 we recall relevant definitions of π-calculus. In Sect. 4 we present the structure of our description language ACDL and a complete textual notation of a client-server system which is treated as the working example in the sequel. In Sect. 5 we describe the translation of expressions in ACDL into processes in π-calculus. In Sect. 6 we present several theorems dealing with analyses of architectural properties based on the description of ACDL and illustrate their significance by the working example. Finally, in Sect. 7 we conclude our paper and report the future work. The proofs of the theorems are provided in the Appendix.

2 Motivating Examples

We motivate our modeling approach by considering the simple client-server system shown in Fig. 1(a), which consists of two clients and one server linked by a black-box middleware embodying the functionality of procedure-callings from the clients to the server. One modeling viewpoint considers this system as a composition of two subsystems as shown in Fig. 1(b). In other words, two clients are linked to the server via two

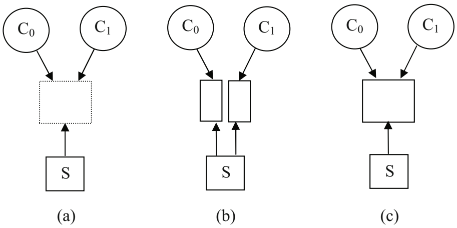

Fig. 1. Client-Server Systems

independent procedure-call connectors. We call this kind of connection the dispersed-mode connection. An obvious advantage of dispersed-mode connection is that, because each links one client and one server only, two connectors can be formally described as instances of the same connector type in ADLs such as Wright and PADL [8]. The dispersed-mode connection also applies to multi-client-server cases. However, this model disperses the connected middleware, and hence, is unable to implement within the connectors some global strategies of coordination of clients and servers, for example, a fairness strategy for access of clients to a server. The implementation of such coordination strategies in a connector is particularly desirable if we consider a connector as a first-class entity in an architecture, whose advantages have been increasingly recognized [13].

Given the above considerations, it is reasonable therefore to adopt another modeling viewpoint, i.e. the centralized-mode connection (as contrary to dispersed-mode connection), in which (take our client-server system for example) both clients are linked to the server via a single procedure-call connector, as shown in Fig. 1(c). However, one obvious difficulty for the centralized-mode connection is how to find a generic representation of connectors that are able to be attached to different numbers of clients (and servers). The significance of such a representation is two-fold. It is theoretically interesting. For example, the three connectors in Figure 1(b) and (c) can be seen as three instances of such a representation. On the other hand it favors the implementing practices. For example, the applicability of this representation in other client-server systems with different numbers of clients and servers advocates the reusability principle in software architecture [14] [15]. In this paper we offer a solution to this problem by developing ACDL to formally describe connectors in a manner where they are insensitive to the number of attached same-type components.

3 π- Calculus

In this section we summarize relevant definitions of a version of π-calculus, which is treated as the semantics of ACDL. For a reference to π-calculus we refer to [16].

We assume an infinite set of names, ranged over by a, b, c, x, y and z. The π-calculus syntax is given by the following grammar:

$$P ::= \pi.P \mid \mathbf{0} \mid P + Q \mid P \| Q \mid P \backslash N \mid I$$

Table 1. SOS of π-calculus

$a(x).P \xrightarrow{a\langle y\rangle} P\{y/x\}$	$\alpha.P \xrightarrow{\alpha} P$ if $\alpha \neq a(x)$
$\dfrac{P \xrightarrow{\alpha} P'}{P + Q \xrightarrow{\alpha} P'}$	$\dfrac{P \xrightarrow{\alpha} P'}{Q + P \xrightarrow{\alpha} P'}$
$\dfrac{P \xrightarrow{\alpha} P'}{P \parallel Q \xrightarrow{\alpha} P' \parallel Q}$	$\dfrac{P \xrightarrow{\alpha} P'}{Q \parallel P \xrightarrow{\alpha} Q \parallel P'}$
$\dfrac{P \xrightarrow{\overline{a}\langle x\rangle} P' \quad Q \xrightarrow{a\langle x\rangle} Q'}{P \parallel Q \xrightarrow{\tau} P' \parallel Q'}$	$\dfrac{P \xrightarrow{\overline{a\_c}} P' \quad Q \xrightarrow{a\_c} Q'}{P \parallel Q \xrightarrow{\tau} P' \parallel Q'}$
$\dfrac{P \xrightarrow{\alpha} P', \quad ch(\alpha) \neq N}{P\backslash N \xrightarrow{\alpha} P'\backslash N}$	$\dfrac{P \xrightarrow{\alpha} P', \quad I \overset{\mathrm{def}}{=} P}{I \xrightarrow{\alpha} P'}$

where P is called a *process*, π is called a *prefix* and ranges over $\overline{a}\langle x\rangle, a(y), \overline{a\_c}, a\_c$ and τ (the silent action), N is called a *channel* and ranges over a and a_c, and I ranges over *process identifiers*.

An *action* α ranges over π and $a\langle x\rangle$. The *structural operational semantics* (SOS) of π-calculus is a set of derivative rules defining a relation called *transition* $\mathcal{O} \subseteq$ Process $\times$ Action $\times$ Process. We write $P \xrightarrow{\alpha} Q$ if $(P, \alpha, Q) \in \mathcal{O}$. The SOS of our π-calculus is given in Table 1.

The SOS compiles a π-calculus process P into a *Labeled Transition System* (LTS) called the LTS *of* P. A *transition path* φ (of the LTS) of P is a *maximum* concatenation of transitions, i.e., either an infinite concatenation of transitions or a finite concatenation of transitions such that the last process has no more transitions. Q is *reachable* from P if there is a finite concatenation of transitions from P to Q. Furthermore, $ch(\alpha)$ refers to the channel of α. $\mathrm{Ch}(P)$ denotes the set of channels of actions labeled in the transition paths of the LTS of P. $P \xrightarrow{\alpha}$ means $P \xrightarrow{\alpha} P'$ for some P'. Given a specific occurrence of P in Q, $P \xrightarrow{\alpha}$ *in* Q means that, $P \xrightarrow{\alpha} P'$ for some P' and, in addition, this transition is a primitive of the derivation of $Q \xrightarrow{\beta} Q'$ for some Q', β according to the SOS in Table 1.[1] For example, $\bar{a}\langle b\rangle.P \xrightarrow{\bar{a}\langle b\rangle}$ in $(\bar{a}\langle b\rangle.P \| a(x).Q)\backslash a$. Let $\mathcal{I} = \{N_0, \ldots, N_n\}$, $P\backslash\mathcal{I}$ abbreviates $P\backslash N_0 \cdots \backslash N_n$; $(\pi + \pi').P$ abbreviates $\pi.P + \pi'.P$.

Definition 1. *P is* deadlock-free*, if there is no finite transition path of P.*

Definition 2. *P is* strongly deadlock-free*, if P is deadlock-free and all its transition paths contain infinite non-silent actions, i.e. α's such that $\alpha \neq \tau$.*

Strong deadlock-freedom means processes interact with the environment infinitely often.

Definition 3. *$P_1, \ldots, P_n$ are* mutually interaction-live against (the restriction of) $\mathcal{I}$, *if for each $Q = (Q_1 \| \cdots \| Q_n)\backslash\mathcal{I}$ reachable from $P = (P_1 \| \cdots \| P_n)\backslash\mathcal{I}$ and each $i \in [1, n]$, there is $R = (R_1 \| \cdots \| R_n)\backslash\mathcal{I}$ reachable from Q such that $R_i \xrightarrow{\alpha}$ in R for some $\alpha \neq \tau$ where $ch(\alpha) \in \mathcal{I}$.*

[1] Throughout this paper when $P \xrightarrow{\alpha}$ *in* Q is written, the specific occurrence of P in Q is clear in the context.

When $P_1, \ldots, P_n$ represent all components (including the connector) of a system, and $\mathcal{I}$ is the set of communication channels of the system, interaction-liveness formulates the property that, this system will not proceed to a situation in which some of its components can no longer interact with the rest of the system.[2]

4 Specifying Architectural Connection

In this section we present the architecture description language ACDL and a working example throughout the remainder of this paper. A textual notation in ACDL consists of two parts: an architecture type and an architecture. The former specifies the component types and the connector; the latter specifies the components of corresponding types. The structure of ACDL is given in the following template:

```
ArchitectureType {"name"}
  ComponentType {"name"}
    Input {...}
    Output {...}
    Control {...}
    Behavior {...} %in Process-Algebra form%
  Connector {"name"}
    Protocol {...} %in Process-Algebra form%
Architecture {"name"}
  Configuration {"Component : ComponentType"}
```

A *ComponentType* is defined as a function of *Input*, *Output*, *Control* and *Behavior*. Elements in *Input* and *Output* are indexed by their component-type name, and elements in *Control* convey component-type names (their own and others). *Behavior* of a *ComponentType* is specified in the Process-Algebra form and its prefixes are the elements in *Input*, *Output* and *Control*. A *Connector* is defined as a function of *Protocol*, which is also specified in Process-Algebra form and whose prefixes are derived from elements in *Input*, *Output* and *Control* of every *ComponentType* in this way: if *act_name* is in *Input* or *Output* of some *ComponentType*, then *act_x* is a *possible* prefix of *Protocol*; if *act*$\langle$*name*$\rangle$ is in *Control* of some *ComponentType*, then *act*(y) is a *possible* prefix of *Protocol*. Note that *act*(y) binds variable *y*. We further require that no free occurrence of variables appears in *Protocol*.

Our working example is the complete textual notation of a simple client-server system named *SimpleCS*, in which three clients, *Client*$_0$, *Client*$_1$ and *Client*$_2$, and two servers, *Server*$_0$ and *Server*$_1$, are linked by a procedure-call connector *ProCall*:

```
Architecture Type {Client-Server}
  ComponentType {C} %for Client%
    Input {result_C}
    Output {request_C}
    Control {log<C>, target<S>}
```

[2] Note that the interaction-liveness is strictly weaker than that each component will interact with other components in the system infinitely often. In our working example, the latter property is not desirable.

```
      Behavior {Client = internalCompute.log<C>.
         target<S>.request_C.result_C.Client}
    ComponentType {S} %for Server%
      Input {involve_S}
      Output {return_S}
      Behavior {Server = involve_S.internalCompute.
         return_S.Server}
    Connector {ProCall}
      Protocol {ProCall = log(x).target(y).request_x.
         involve_y.return_y.result_x.ProCall}
  Architecture {SimpleCS}
    Configuration {C0,C1,C2:C; S0,S1:S}
```

There are three important points to be observed. First, behaviors of components are derived from *Behavior* of their types, so we need not specify them in the textual notation. Secondly, the connector *ProCall* obtains its knowledge of involved components from the information conveyed in *log* and *target*, and consequently, the specification of architecture type *Client-Server* need not have any restriction on the number of components, i.e. instances of *Client* and *Server*. In this way, a component-number-insensitive connector *ProCall* is formally described. This is the main feature of ACDL. Finally, the architecture type/instance separation in ACDL implies that the specification of *Client-Server* may be reused when describing other architectures of the same type.

5 Formal Semantics

In this section, we bridge ACDL and π-calculus. For the convenience of discussion, we set down some notations in Table 2.

Table 2. Notation Convention

STRUCTURES	NOTATIONS	SEMANTICS
architecture type	$\mathbb{A}$	-
architecture	$\mathcal{A}$	$[\mathcal{A}]$
component type	$\mathbb{E}$	-
component	$\mathcal{E}$	$[\mathcal{E}]$
connector	$\mathcal{G}$	$[\mathcal{G}]$

The semantics $[\mathcal{E}]$ of component $\mathcal{E}$ is the *process identifier* whose recursive definition is naturally obtained from the behavior of $\mathcal{E}$ (not its component type) according to the input-, output- and silent-nature of the prefixes (elements in *Control* are output-nature). Similar treatment applies to the semantics $[\mathcal{G}]$ of a connector $\mathcal{G}$. But prefixes in $[\mathcal{G}]$ are dual to prefixes in some $[\mathcal{E}]$ provided that $\mathcal{E}, \mathcal{G}$ are in one architecture. As an example, Table 3 gives the semantics of the components *Client*$_0$, *Server*$_0$ and the connector *ProCall* in the *SimpleCS* system.

In the sequel, we assume that $\mathcal{E}_1, \ldots, \mathcal{E}_n$ list all components in $\mathcal{A}$, and $\mathcal{G}$ is the connector in $\mathcal{A}$. Let $\mathcal{I}_{\mathcal{E}_i}$, called *the set of channels of* $\mathcal{E}_i$, be the set of channels in *Input*,

Table 3. Semantics Samples

$$
\begin{aligned}
[\mathit{Client}_0] &= \tau.\, \overline{\mathit{log}}\langle C_0\rangle.\, (\overline{\mathit{target}}\langle S_0\rangle + \overline{\mathit{target}}\langle S_1\rangle).\, \overline{\mathit{request\_C}_0}.\, \mathit{result\_C}_0.\, [\mathit{Client}_0]\\
[\mathit{Server}_0] &= \mathit{involve\_S}_0.\, \tau.\, \overline{\mathit{return\_S}_0}.\, [\mathit{Server}_0]\\
[\mathit{ProCall}] &= \mathit{log}(x).\, \mathit{target}(y).\, \mathit{request\_x}.\, \overline{\mathit{involve\_y}}.\, \mathit{return\_y}.\, \overline{\mathit{result\_x}}.\, [\mathit{ProCall}]
\end{aligned}
$$

Output and *Control* of $\mathcal{E}_i$. For example, $\mathcal{I}_{C_0} = \{\mathit{log}, \mathit{target}, \mathit{request\_C}_0, \mathit{result\_C}_0\}$. The semantics of $\mathcal{A}$ is defined by

$$[\mathcal{A}] = ([\mathcal{E}_1]\| \cdots \|[\mathcal{E}_n]\|[\mathcal{G}])\backslash\mathcal{I}_{\mathcal{A}} \ ,$$

where $\mathcal{I}_{\mathcal{A}} = \bigcup_{i=1}^{n} \mathcal{I}_{\mathcal{E}_i}$. Note that the positions of $[\mathcal{E}_1]$, ..., $[\mathcal{E}_n]$ and $[\mathcal{G}]$ do not affect the analyses of temporal properties of $\mathcal{A}$. The following propositions formulate some neccessary properties of ACDL to formulate or prove the theorems later. Let $i, j, k \in [1, n]$.

Proposition 1. *If $\mathcal{E}_i, \mathcal{E}_j$ and $\mathcal{E}_k$ are of the same type, then (i) $\mathcal{I}_{\mathcal{E}_i} \cap \mathcal{I}_{\mathcal{E}_j} = \mathcal{I}_{\mathcal{E}_i} \cap \mathcal{I}_{\mathcal{E}_k}$, and, (ii) $\mathcal{I}_{\mathcal{E}_j} = \{N\{b/a\} : N \in \mathcal{I}_{\mathcal{E}_i}\}$ and $[\mathcal{E}_i] = [\mathcal{E}_j]\{b/a\}$, where a, b are the names of $\mathcal{E}_i, \mathcal{E}_j$, respectively.*

Proposition 1 formalizes that same-type components share the same *Control* and that their *Input* and *Output* are parameterized on their names.

Proposition 2. *(1) $\mathrm{Ch}([\mathcal{E}_i]) \subseteq \mathcal{I}_{\mathcal{E}_i}$ for each $\mathcal{E}_i$. (2) For each $P = (P_1\| \cdots \|P_n\|P_{n+1})$ $\backslash\mathcal{I}_{\mathcal{A}}$ reachable from $([\mathcal{E}_1]\| \cdots \|[\mathcal{E}_n]\ \|[\mathcal{G}])\backslash\mathcal{I}_{\mathcal{A}}$, if $P_{n+1} \xrightarrow{\alpha}$ in P, then either $\alpha = \tau$ or $ch(\alpha) \in \mathcal{I}_{\mathcal{A}}$.*

The first clause of Proposition 2 justifies the name of $\mathcal{I}_{\mathcal{E}_i}$, i.e. the set of channels of $\mathcal{E}_i$. The second clause justifies the definition of $[\mathcal{A}]$ above by showing all channels of $\mathcal{G}$ are in $\mathcal{I}_{\mathcal{A}}$, and implies that $\mathcal{G}$ indeed functions to coordinate behaviors of components in $\mathcal{A}$ only.

6 Analyzing Architectural Properties

In this section we develop formal techniques to analyze deadlock-freedom and interaction-liveness based on the framework of ACDL. To improve the readability we put all the proofs of theorems in the Appendix. The utilities of the theorems are illustrated by the working example – the *SimpleCS* system.

We say $\mathcal{A}$, $\mathcal{E}$ or $\mathcal{G}$, respectively, is *deadlock-free*, if $[\mathcal{A}]$, $[\mathcal{E}]$ or $[\mathcal{G}]$, respectively, is deadlock-free; $\mathcal{E}'_1, \ldots, \mathcal{E}'_m$ (selected from $\mathcal{E}_1, \ldots, \mathcal{E}_n$) and $\mathcal{G}$ are *mutually interaction-live against (the restriction of)* $\mathcal{I}$, if $[\mathcal{E}'_1], \ldots, [\mathcal{E}'_m]$ and $[\mathcal{G}]$ are mutually deadlock-free against $\mathcal{I}$; $\mathcal{A}$ is *interaction-live* if $\mathcal{E}_1, \ldots, \mathcal{E}_n$ and $\mathcal{G}$ are mutually interaction-live against $\mathcal{I}_{\mathcal{A}}$.

We still need to formulate one property expressing that the connector fits the components: $\mathcal{G}$ is *compatible with* $\{\mathcal{E}'_1, \ldots, \mathcal{E}'_m\}$ *against* $\mathcal{I}$, if each transition path of $([\mathcal{E}'_1]\| \cdots \|$ $[\mathcal{E}'_m]\|[\mathcal{G}])\backslash\mathcal{I}$ contains infinitely many processes $P = (P_1\| \cdots \|\ P_m\|P_{m+1})\backslash\mathcal{I}$ such that: if $P_{m+1} \xrightarrow{\alpha}$ and $ch(\alpha) \in \mathcal{I}$, then $P_{m+1} \xrightarrow{\alpha}$ in P.

6.1 Compositional Analyses

For convenience, we use the collection of the elements in the architecture $\mathcal{A}$ to denote $\mathcal{A}$ itself, i.e. $\mathcal{A} = \{\mathcal{E}_1, \ldots, \mathcal{E}_n, \mathcal{G}\}$. To carry out the compositional analyses, we need an auxiliary definition. Let $\mathcal{P}$ be the *finest* partition on $\mathcal{A} - \{\mathcal{G}\}$ such that $\mathcal{E}_j \in \mathcal{P}(\mathcal{E}_i)$ whenever $\mathcal{I}_{\mathcal{E}_j} \cap \mathcal{I}_{\mathcal{E}_i} \neq \emptyset$. We let

$$\mathcal{A} - \{\mathcal{G}\} = \{\mathcal{E}_1^1, \ldots, \mathcal{E}_1^{k_1}, \ldots, \mathcal{E}_m^1, \ldots, \mathcal{E}_m^{k_m}\} \ ,$$

where $\sum_{i=1}^m k_i = n$ and $\mathcal{P}(\mathcal{E}_i^1) = \{\mathcal{E}_i^1, \ldots, \mathcal{E}_i^{m_i}\}$, and let $\mathcal{I}_{\mathcal{P}(\mathcal{E}_i)} = \bigcup_{\mathcal{E}_j \in \mathcal{P}(\mathcal{E}_i)} \mathcal{I}_{\mathcal{E}_j}$. Note that by Proposition 1 either $\mathcal{P}(\mathcal{E}_i^1) = \{\mathcal{E}_i^1\}$ or $\mathcal{P}(\mathcal{E}_i^1)$ is the super set of the set of same-type components including $\mathcal{E}_i$. This partition sets the stage for the compositional analyses: analyses of the architecture are decomposed into analyses of parts according to the partition, while "finest" refers to the possibly finest-grained decomposition. This renders our compositional analytic method (for deadlock-freedom, i.e. Theorem 1) more general than that in [8] where components in an acyclic-topology architecture share no channels due to the definition of PADL.

Theorem 1. *If $\mathcal{G}$ is deadlock-free and compatible with $\mathcal{P}(\mathcal{E}_i^1)$ against $\mathcal{I}_{\mathcal{P}(\mathcal{E}_i^1)}$ for each $i \in [1, m]$, then $\mathcal{A}$ is deadlock-free.*

The proofs of Theorem 1 and other theorem below are given in the Appendix. Theorem 1 allows us to reduce the checking of the deadlock-freedom of $\mathcal{A}$ to the checking of the deadlock-freedom of $\mathcal{G}$ and compatibility of $\mathcal{G}$ with parts of $\mathcal{A}$, i.e. $\{\mathcal{E}_i^1, \ldots, \mathcal{E}_i^{m_i}\}$ where $i \in [1, m]$. For the *SimpleCS* system, to check the deadlock-freedom of the whole system, it suffices to check: (1) the deadlock-freedom of *ProCall*, and, (2) the compatibility of *ProCall* with $\{\mathit{Client}_0, \mathit{Client}_1, \mathit{Client}_2\}$ against $\mathcal{I}_{c0} \cup \mathcal{I}_{c1} \cup \mathcal{I}_{c2}$, with $\{\mathit{Server}_0\}$ against $\mathcal{I}_{s0}$, and with $\{\mathit{Server}_1\}$ against $\mathcal{I}_{s1}$, respectively. By decomposing the analyses, we may be able to use previous checking results to check other similar architectures, and detect which part of an architecture is responsible for deadlocks (if any) and hence, make diagnoses.

Theorem 2. *If $\mathcal{A}$ satisfies the conditions in Theorem 1, all components in $\mathcal{A}$ are strongly deadlock-free, and $\mathcal{G}$ is mutually interaction-live against $\mathcal{I}_{\mathcal{P}(\mathcal{E}_i^1)}$ for each $i \in [1, m]$, then $\mathcal{A}$ is interaction-live.*

If $\mathcal{A}$ satisfies the conditions in Theorem 1, Theorem 2 licenses us to reduce the checking of the interaction-liveness of $\mathcal{A}$ to the checking of the following two: the strong deadlock-freedom of each component in $\mathcal{A}$, and the mutual interaction-liveness of $\mathcal{E}_i^1, \ldots, \mathcal{E}_i^{k_i}, \mathcal{G}$ against $\mathcal{I}_{\mathcal{P}(\mathcal{E}_i^1)}$ for each $i \in [1, m]$. For the *SimpleCS* system, to check the mutual interaction-liveness of $\mathcal{A}$, it suffices to check: (1) the strong deadlock-freedom of all clients and servers, and, (2) the mutual interaction-liveness of Client_0, Client_1, Client_2 and *ProCall* against $\mathcal{I}_{c0} \cup \mathcal{I}_{c1} \cup \mathcal{I}_{c2}$, of Server_0 and *ProCall* against $\mathcal{I}_{s0}$, and of Server_1 and *ProCall* against $\mathcal{I}_{s1}$, respectively. The significance of Theorem 2 is similar to Theorem 1, as described above. Theorem 2 also shows that the checking of interaction-liveness can be based on the checking of deadlock-freedom according to Theorem 1.

Theorem 3. *If $\mathcal{E}_i$ and $\mathcal{E}_j$ are of the same type and $\mathcal{I}_{\mathcal{E}_i} \cap \mathcal{I}_{\mathcal{E}_j} = \emptyset$, then (1) $\mathcal{G}$ is compatible with $\{\mathcal{E}_i\}$ against $\mathcal{I}_{\mathcal{E}_i}$ if and only if $\mathcal{G}$ is compatible with $\{\mathcal{E}_j\}$ against $\mathcal{I}_{\mathcal{E}_j}$, (2) $\mathcal{E}_i$ is strongly deadlock-free if and only if $\mathcal{E}_j$ is strongly deadlock-free, and, (3) $\mathcal{E}_i$ and $\mathcal{G}$ are mutually interaction-live against $\mathcal{I}_{\mathcal{E}_i}$ if and only if $\mathcal{E}_j$ and $\mathcal{G}$ are mutually interaction-live against $\mathcal{I}_{\mathcal{E}_j}$.*

In the *SimpleCS* system, according to Theorem 3 we have that, for example, *Procall* is compatible with $\{\text{Server}_0\}$ against $\mathcal{I}_{s0}$ if and only if *Procall* is compatible with $\{\mathit{Server}_1\}$ against $\mathcal{I}_{s1}$, and *Server*$_0$ is strongly deadlock-free if and only if *Server*$_1$ is strongly deadlock-free. With this theorem we can safely skip over the checking of some parts of an architecture.

6.2 Type-Based Analyses

The compositional analyses are carried out in the level of architecture instance. We now develop analytic techniques in the level of architecture type.

Similar to an architecture, we treat an architecture type $\mathbb{A}$ as a collection of component types, together with a connector. In this section we assume $\mathbb{E} \in \mathbb{A}$. Note that $\mathbb{E}$ is disjointed if and only if $\mathcal{P}(\mathcal{E}^{\mathbb{E}}) = \{\mathcal{E}^{\mathbb{E}}\}$ for some $\mathcal{P}$. Let $\mathcal{E}^{\mathbb{E}}$ refer to a component of type $\mathbb{E}$ and $\mathcal{A}^{\mathbb{A}}$ an architecture of type $\mathbb{A}$. We say $\mathbb{A}$ is *open for deadlock-freedom* on $\mathbb{E}$, if $\mathcal{A}^{\mathbb{A}} \cup \{\mathcal{E}^{\mathbb{E}}\}$ and $\mathcal{A}^{\mathbb{A}} - \{\mathcal{E}^{\mathbb{E}}\}$ are deadlock-free whenever $\mathcal{A}^{\mathbb{A}}$ is deadlock-free;[3] $\mathbb{A}$ is *open for interaction-liveness* on $\mathbb{E}$, if the proposition of the same form holds for interaction-liveness. Informally, if $\mathbb{A}$ is open for deadlock-freedom on $\mathbb{E}$, then every new architecture obtained from a deadlock-free architecture of type $\mathbb{A}$ via adding, deleting and replacing instances of $\mathbb{E}$ is also deadlock-free, and if $\mathbb{A}$ is open for interaction-liveness on $\mathbb{E}$, then every new architecture obtained from an interaction-live architecture of type $\mathbb{A}$ via adding, deleting and replacing instances of $\mathbb{E}$ is also interaction-live. We are going to set down some reasonable conditions on component types to ensure these two properties hold.

We say $\mathbb{E}$ is *disjointed*, if $\mathcal{I}_{\mathcal{E}_1^{\mathbb{E}}} \cap \mathcal{I}_{\mathcal{E}_2^{\mathbb{E}}} = \emptyset$ for any components $\mathcal{E}_1^{\mathbb{E}}, \mathcal{E}_2^{\mathbb{E}}$; $\mathbb{E}$ is *excluded*, if for any component $\mathcal{E}_1^{\mathbb{E}}, \mathcal{E}_2^{\mathbb{E}}$ the following holds: for each $P = (P_1 \| P_2 \| P_3) \backslash \mathcal{I}_{\mathcal{E}_1^{\mathbb{E}}} \cup \mathcal{I}_{\mathcal{E}_2^{\mathbb{E}}}$ reachable from $([\mathcal{E}_1^{\mathbb{E}}] \| [\mathcal{E}_2^{\mathbb{E}}] \| [\mathcal{G}]) \backslash\, \mathcal{I}_{\mathcal{E}_1^{\mathbb{E}}} \cup \mathcal{I}_{\mathcal{E}_2^{\mathbb{E}}}$, there is $\alpha \neq \tau$ such that $P_i \xrightarrow{\alpha}$ in P for each $i \in \{1, 2\}$, only if P_j where $j = 3 - i$ is reachable from $[\mathcal{E}_j^{\mathbb{E}}]$ via a finite concatenation of transitions labeled by τ only. Informally, a component type is disjointed if and only if its *Control* is empty. The definition of “excludedness” implies the following lemma which says, informally, that each component of that type must not start its interaction if any other component of the same type is in the middle of interaction, and whose proof follows immediately from the definition of excludedness.

Lemma 1. *Suppose $\mathcal{E}'_1, \ldots, \mathcal{E}'_m$ $(m \geq 2)$ are all instances of $\mathbb{E}$ in $\mathcal{A}^{\mathbb{A}}$, and $\mathbb{E}$ is excluded, then: For each $P = (P_1 \| \cdots \| P_m \| P_{m+1}) \backslash \bigcup_{i=1}^{m} \mathcal{I}_{\mathcal{P}(\mathcal{E}'_i)}$ reachable from $([\mathcal{E}'_1] \| \cdots \| [\mathcal{E}'_m] \| [\mathcal{G}]) \backslash \bigcup_{i=1}^{m} \mathcal{I}_{\mathcal{P}(\mathcal{E}'_i)}$, $P_i \xrightarrow{\alpha}$ in P where $\alpha \neq \tau$ for each $i \in [1, m]$, only if P_j where $j \in [1, m] - \{i\}$ is reachable from $[\mathcal{E}'_j]$ via a finite concatenation of transitions labeled by τ only.*

[3] For $\mathcal{A}^{\mathbb{A}} - \{\mathcal{E}^{\mathbb{E}}\}$ we have to suppose $\mathcal{A}^{\mathbb{A}}$ has more than one instances of type $\mathbb{E}$, for there must be at least one instance for each component type in an architecture.

We now demonstrate the relationship between disjointedness, excludedness, deadlock-freedom and interaction-liveness.

Theorem 4. *If $\mathbb{E}$ is disjointed, then $\mathbb{A}$ is open both for deadlock-freedom and for interaction-liveness on $\mathbb{E}$.*

Theorem 4 allows us to add and delete components of disjointed types without making the architecture lose deadlock-freedom and interaction-liveness, if the architecture enjoyed these two properties originally. We illustrate the application of Theorem 4 by our working example – the *SimpleCS* system. Since the the component type *Server* does not have any *Control* actions, it is not hard to prove that *Server* is disjointed. If we have already obtained the result that *SimpleCS* is deadlock-free (resp. interaction-live), then we can build new systems based on *SimpleCS* that are also deadlock-free (resp. interaction-live), such as:

$$\mathit{MultiServerCS} = \mathit{SimpleCS} \cup \{\mathit{Server}_2, \ldots, \mathit{Server}_n\} - \{\mathit{Server}_0, \mathit{Server}_1\} \ .$$

If we do not know the deadlock-freedom and interaction-liveness of *SimpleCS*, we check the following simpler system with one server only:

$$\mathit{SingleServerCS} = \mathit{SimpleCS} - \{\mathit{Server}_1\} \ .$$

Theorem 5. *If $\mathbb{E}$ is excluded, then $\mathbb{A}$ is open both for deadlock-freedom and for interaction-liveness on $\mathbb{E}$.*

The significance of Theorem 5 is just like Theorem 4 in that it allows us to add and delete components of excluded types without making the architecture lose deadlock-freedom and interaction-liveness, if the architecture enjoyed these two properties originally. In the *SimpleCS* system, since the $\overline{\mathit{log}}$ actions in each *Client* instance has to synchronize with the *log* actions in the connector *ProCall*, it is not hard to verify that component type *Client* is excluded. As above, if we already have the result that *SimpleCS* is deadlock-free (resp. interaction-live), then we can build new systems based on *SimpleCS* that are also deadlock-free (resp. interaction-live) (combining Theorem 4), such as:

$$\mathit{MultiCS} = \mathit{MultiSeverCS} \cup \{\mathit{Client}_3, \ldots, \mathit{Client}_m\} - \{\mathit{Client}_0, \mathit{Client}_1, \mathit{Client}_2\} \ .$$

If we do not know the deadlock-freedom or interaction-liveness of *SimpleCS*, we check the following elementary system with one client and one server only (combining Theorem 4):

$$\mathit{SingleCS} = \mathit{SingleServerCS} - \{\mathit{Client}_1, \mathit{Client}_2\} = \{\mathit{Client}_0, \mathit{Server}_0, \mathit{ProCall}\} \ .$$

In total, what Theorem 4 and Theorem 5 tell us is when and to what extent the checking of deadlock-freedom and interaction-liveness of an architecture can be carried out in the type level. A final point worthy to be noticed is that, combining with the compositional analytic techniques (Theorem 1 and 2), the checking of *SingleCS* can be splitted into the checking of the following two subsystems:

$$\mathit{SingleCS}_C = \{\mathit{Client}_0, \mathit{ProCall}\}, \quad \mathit{SingleCS}_S = \{\mathit{Server}_0, \mathit{ProCall}\} \ .$$

6.3 Discussions

We have shown how our analytic techniques, i.e. the compositional analyses and the type-based analyses, deal with the deadlock-freedom and the interaction-liveness of an architecture. We now summarize how these techniques improve the system checking in the following four ways and explain them by examples.

- First, our method enhances the use of previous checking results to deal with new checking problems. For example, if we already know that *ProCall* is compatible with Server_0 against $\mathcal{I}_{s_0}$, then we can deduce that *ProCall* is compatible with Server_1 against $\mathcal{I}_{s_1}$.
- Secondly, our method helps make diagnoses of those architectures failing to satisfy a desired property. This is due to the compositional nature of our first analyses. Suppose a client-server architecture *BadCS* contains a deadlock and we detect that the *ProCall* is not compatible with the client type, say, *BadClient*. By fixing *BadClient* we may obtain a deadlock-free architecture.
- Thirdly, our method reduces the complexity of system checking. For example, we can reduce the checking of deadlock-freedom and interaction-liveness of the system *SimpleCS* to the checking of those properties of the system *SingleCS*.
- Finally, while the architecture-type/instance distinction in ACDL makes the reusability of architecture-type specifications to describe new architectures possible, our type-based analyses facilitate this reusability by showing when and to what extent the system checking of some properties can be undertaken in the type level.

7 Conclusions and Future Work

In this paper we consider components in software-intensive systems as interfaces performing behaviors of input, output, and internal actions, and connectors as a special kind of components that glue components by specifying coordination protocols for their behaviors. Our focus is the centralized-mode architectural connection in which all components are linked by a single connector. We have proposed a new ADL called ACDL, the key feature of which is that it describes connectors in such a way that they are insensitive to the numbers of attached same-type components. We develop two kinds of analytic techniques customizing ACDL, i.e. compositional analyses and type-based analyses, to improve the system checking of temporal properties, such as deadlock-freedom and interaction-liveness, of an architecture. The latter property is a liveness property formulating that, during the running-time of the system, each component will never be trapped in a situation where no future interactions with the rest of the system is possible.

Our future work will follow two directions. First, the interaction-liveness is only one kind of liveness properties, but we foresee that our method applies to other liveness properties. Therefore one challenging problem is to find out what range of liveness properties can be dealt with by our method, and to give them a formal definition.

The other important direction is the tool support for ACDL. A tool-set accompanying an ADL is, strictly speaking, not part of the language itself, but the purpose of developing formal languages for architectural description is because their formality implies their suitability to be manipulated by software tools [3]. However, until now we have not offered (in this paper or elsewhere) any tool support for ACDL, such as a parser which analyzes the syntactic correctness of a piece of written ACDL textual notation, and this renders ACDL rather conceptual. A "shortcut" to overcome this shortcoming is mapping a conceptual language like ACDL to a standard language equipped with a well-developed toolkit such as UML (currently UML2.0 [17]), so an ACDL user can leverage the tools customizing UML like a code-generator and be favor of the theoretic merits of ACDL in practice. Nonetheless, the applicability of the mapping depends on whether and to what extent UML supports modeling the abstractions formally described by ACDL, especially given the fact that UML is a semi-formal language. Optimistically, UML has an extension mechanism permitting one use Object Constraint Language (OCL) [18], which is based on set theory and predicate logic, to provide a precise description of the information unable to be expressed in standard UML diagrams. The general applicability of using UML to model software architectures as several representatives of ADLs do has been evaluated in the literature [19]. In our case, however, a thorough examination is needed.

Acknowledgements

The authors would like to thank the anonymous referees for their helpful comments to improve the draft of this paper.

References

1. Garlan, D., Shaw, M.: An introduction to software architecture. Technical report, Pittsburgh, PA, USA (1994)
2. Sifakis, J.: A framework for component-based construction. In: Proceedings of the 3rd IEEE International Conference on Software Engineering and Formal Methods (2005)
3. Medvidovic, N., Taylor, R.: A classification and comparison framework for software architecture description languages. IEEE Transactions on Software Engineering 26(1), 70–93 (2000)
4. Allen, R., Garlan, D.: A formal basis for architectural connection. ACM Transactions on Software Engineering and Methodology 6(3), 213–249 (1997)
5. Oquendo, F.: π-ADL: an architecture description language based on the higher-order typed π-calculus for specifying dynamic and mobile software architectures. ACM SIGSOFT Software Engineering Notes 29(3), 1–14 (2004)
6. Tivoli, M., Inverardi, P.: Failure-free coordinators synthesis for component-based architectures. Science of Compututer Programming 71(3), 181–212 (2008)
7. Milner, R., Parrow, J., Walker, D.: A calculus of mobile processes. Information and Computation 100(1), 1–77 (1992)
8. Bernardo, M., Ciancarini, P., Donatiello, L.: Architecting families of software systems with process algebras. ACM Transactions on Software Engineering and Methodology 11(4), 386–426 (2002)
9. Inverardi, P., Wolf, A.L., Yankelevich, D.: Static checking of system behaviors using derived component assumptions. ACM Transactions on Software Engineering and Methodology 9(3), 239–272 (2000)

10. Aldini, A., Bernardo, M.: On the usability of process algebra: An architectural view. Theoretical Computer Science 335(2-3), 281–329 (2005)
11. Magee, J., Dulay, N., Eisenbach, S., Kramer, J.: Specifying distributed software architectures. In: Proceedings of the 5th European Software Engineering Conference, pp. 137–153 (1995)
12. Mateescu, R., Oquendo, F.: π-AAL: an architecture analysis language for formally specifying and verifying structural and behavioural properties of software architectures. ACM SIGSOFT Software Engineering Notes 31(2), 1–19 (2006)
13. Shaw, M., Garlan, D.: Software Architecture: Perspectives on an Emerging Discipline. Prentice-Hall, NJ (1996)
14. Spitznagel, B., Garlan, D.: A compositional formalization of connector wrappers. In: Proceedings of the 25th International Conference on Software Engineering (2003)
15. Giesecke, S.: Taxonomy of architectural style usage. In: Proceedings of the 2006 Conference on Pattern Languages of Programs (2006)
16. Sangiorgi, D., Walker, D.: π-calculus: A Theory of Mobile Processes. Cambridge University Press, NY (2001)
17. Booch, G., Rumbaugh, J., Jacobson, I.: Unified Modeling Language User Guide, 2nd edn. Addison-Wesley Professional, Reading (2005)
18. Warmer, J., Kleppe, A.: The Object Constraint Language: Getting Your Models Ready for MDA. Addison-Wesley, Boston (2003)
19. Medvidovic, N., Rosenblum, D.S., Redmiles, D.F., Robbins, J.E.: Modeling software architectures in the unified modeling language. ACM Transaction on Software Engineering Methodology 11(1), 2–57 (2002)

Appendix

Proof of Theorem 1. To improve readability we use $\{[\mathcal{E}_i^1]\}$ refering to $[\mathcal{E}_i^1]\|\cdots\|[\mathcal{E}_i^{k_i}]$, for each $i \in [1,m]$. Hence $[\mathcal{A}] = (\{[\mathcal{E}_1^1]\}\|\cdots\|\{[\mathcal{E}_m^1]\}\|[\mathcal{G}])\backslash\mathcal{I}_\mathcal{A}$. We decompose the proof of Theorem 1 into the following three lemmas.

Lemma 2. *If $P = (P_1\|\cdots\|P_m\|P_{m+1})\backslash\mathcal{I}_\mathcal{A}$ is reachable from $[\mathcal{A}] = (\{[\mathcal{E}_1^1]\}\|\cdots\|\{[\mathcal{E}_m^1]\}\|[\mathcal{G}])\backslash\mathcal{I}_\mathcal{A}$, then $(P_i\|P_{m+1})\backslash\mathcal{I}_{\mathcal{P}(\mathcal{E}_i^1)}$ is reachable from $(\{[\mathcal{E}_i^1]\}\|[\mathcal{G}])\backslash\mathcal{I}_{\mathcal{P}(\mathcal{E}_i^1)}$ for each $i \in [1,m]$.*

Proof. The proof is by induction on the number of transitions from $[\mathcal{A}]$ to P. Suppose $Q = (Q_1\|\cdots\|Q_m\|Q_{m+1})\backslash\mathcal{I}_\mathcal{A}$, $Q \xrightarrow{\alpha} P$, and Q is reachable from $[\mathcal{A}]$. W.r.t. Proposition 2 and the partition $\mathcal{P}$, there are only two cases on the derivation of $Q \xrightarrow{\alpha} P$. (1) Suppose $Q \xrightarrow{\alpha} P$ is derived from $Q_i \xrightarrow{\tau} P_i$ for some $i \in [1, m+1]$. Then clearly $(Q_j\|Q_{m+1})\backslash\mathcal{I}_{\mathcal{P}(\mathcal{E}_j^1)}$ for each $j \in [1,m]$. By induction hypothesis we are done. (2) Suppose $Q \xrightarrow{\alpha} P$ is derived from $Q_j \xrightarrow{\beta} P_j$ for some $j \in [1,m]$ and $Q_{m+1} \xrightarrow{\beta'} P_{m+1}$ where β' is dual to β. Then $(Q_j\|Q_{m+1})\backslash\mathcal{I}_{\mathcal{P}(\mathcal{E}_j^1)} \xrightarrow{\beta} (P_j\|P_{m+1})\backslash\mathcal{I}_{\mathcal{P}(\mathcal{E}_j^1)}$, $(Q_k\|Q_{m+1})\backslash\mathcal{I}_{\mathcal{P}(\mathcal{E}_k^1)} \xrightarrow{\beta} (Q_k\|P_{m+1})\backslash\mathcal{I}_{\mathcal{P}(\mathcal{E}_k^1)}$, and $Q_k = P_k$ where $k \in [1,m] - \{j\}$. By induction hypothesis, we have the result.

Lemma 3. *If $\mathcal{G}$ is compatible with $\mathcal{P}(\mathcal{E}_i^1)$ for each $i \in [1,m]$, then $\mathcal{G}$ is compatible with $\mathcal{A} - \{\mathcal{G}\}$ against $\mathcal{I}_\mathcal{A}$.*

Proof. Suppose $P = (P_1 \| \cdots \| P_m \| P_{m+1}) \backslash \mathcal{I}_{\mathcal{A}}$ is reachable from $[\mathcal{A}]$, and $P_{m+1} \xrightarrow{\alpha}$ for some α. By Proposition 2, $ch(\alpha) \in \mathcal{I}_{\mathcal{P}(\mathcal{E}_i^1)}$ for some $i \in [1, m]$. By Lemma 2, $(P_i \| P_{m+1}) \backslash \mathcal{I}_{\mathcal{P}(\mathcal{E}_i^1)}$ is reachable from $([\mathcal{E}_i] \| [\mathcal{G}]) \backslash \mathcal{I}_{\mathcal{P}(\mathcal{E}_i^1)}$. Since $\mathcal{G}$ is compatible with $\mathcal{P}(\mathcal{E}_i^1)$, $P_{m+1} \xrightarrow{\alpha}$ in $(\{[\mathcal{E}_i^1]\} \| [\mathcal{G}]) \backslash \mathcal{I}_{\mathcal{P}(\mathcal{E}_i^1)}$. Therefore $P_{m+1} \xrightarrow{\alpha}$ in $[\mathcal{A}]$.

Lemma 4. *If $\mathcal{G}$ is deadlock-free and compatible with $\mathcal{A} - \{\mathcal{G}\}$ against $\mathcal{I}_{\mathcal{A}}$, $\mathcal{A}$ is deadlock-free.*

Proof. The result is obvious by the definitions.

Proof of Theorem 2. We first show a small lemma:

Lemma 5. *$\mathcal{E}_i$ and $\mathcal{E}_j$ are strongly deadlock-free, the $[\mathcal{E}_i] \| [\mathcal{E}_j]$ is strongly deadlock-free; and this can be generalized to any number of components.*

Proof. This lemma can be proved by show that if $[\mathcal{E}_i] \| [\mathcal{E}_j]$ has a finite transition path or a infinite path failing the satisfying the non-silent-label requirement, then one of $\mathcal{E}_i$ and $\mathcal{E}_j$ must be failed; this can be easily generalized to any finite number of components.

Then we prove Theorem 2:

Proof. Suppose $\mathcal{A}$ satisfies the conditions in Theorem 1, and $\{[\mathcal{E}_j]\}$ is strongly deadlock-free for each $j \in [0, m]$, we show that if for some $i \in [1, m]$ there is an infinite transition path φ of $[\mathcal{A}]$ containing only finitely many P such that there is Q reachable from P and $Q_i \xrightarrow{\alpha}$ in Q for some $\alpha \neq \tau$, then there is an infinite transition path ψ of $(\{[\mathcal{E}_i^1]\} \| \mathcal{G}) \backslash \mathcal{I}_{\mathcal{P}(\mathcal{E}_i^1)}$ containing only finitely many P' such that there is Q' reachable from P' and $Q'_i \xrightarrow{\alpha}$ in Q' for some $\alpha \neq \tau$, and hence prove the theorem. More specifically, ψ is constructed in the following procedure: Suppose P is reachable from $[\mathcal{A}]$ via n transitions in φ and also $P \xrightarrow{\alpha} Q$ is in φ, and let $\widehat{\psi_n}$ be a finite concatenation of transitions starting at $(\{[\mathcal{E}_i^1]\} \| \mathcal{G}) \backslash \mathcal{I}_{\mathcal{P}(\mathcal{E}_i^1)}$ and ending at $(P_i \| P_{m+1}) \backslash \mathcal{I}_{\mathcal{P}(\mathcal{E}_i^1)}$,

1. $\widehat{\psi_0} = (\{[\mathcal{E}_i^1]\} \| \mathcal{G}) \backslash \mathcal{I}_{\mathcal{P}(\mathcal{E}_i^1)}$;
2. If $P \xrightarrow{\alpha} Q$ is derived from $P_i \xrightarrow{\tau} Q_i$ (thus $\alpha = \tau$), then $\widehat{\psi_{n+1}} = \widehat{\psi_n} \xrightarrow{\tau} (Q_i \| Q_{m+1}) \backslash \mathcal{I}_{\mathcal{P}(\mathcal{E}_i^1)}$ where $Q_{m+1} = P_{m+1}$;
3. If $P \xrightarrow{\alpha} Q$ is derived from $P_i \xrightarrow{\beta} Q_i$ and $P_{m+1} \xrightarrow{\beta'} Q_{m+1}$ where β' are dual to β (thus $\alpha = \tau$), then $\widehat{\psi_{n+1}} = \widehat{\psi_n} \xrightarrow{\tau} (Q_i \| Q_{m+1}) \backslash \mathcal{I}_{\mathcal{P}(\mathcal{E}_i^1)}$;
4. If $P \xrightarrow{\alpha} Q$ is derived from $P_j \xrightarrow{\beta} Q_j$ where $j \neq i \in [1, m]$ and $P_{m+1} \xrightarrow{\beta'} Q_{m+1}$ where β' are dual to β, then $\widehat{\psi_{n+1}} = \widehat{\psi_n} \xrightarrow{\beta'} (Q_i \| Q_{m+1}) \backslash \mathcal{I}_{\mathcal{P}(\mathcal{E}_i^1)}$ where $Q_i = P_i$;
5. If $P \xrightarrow{\alpha} Q$ is derived from $P_j \xrightarrow{\tau} Q_j$ where $j \neq i \in [1, m]$ (thus $\alpha = \tau$), then $\widehat{\psi_{n+1}} = \widehat{\psi_n}$.

The strong deadlock-freedom of each $\{[\mathcal{E}_j]\}$ guarantees that there are only finitely many k_1's and k_2's such that $k_1 \neq k_2$ and $\widehat{\psi_{k_1}} = \widehat{\psi_{k_2}}$. Therefore it not hard to verify that $\lim_{n \to \infty} \widehat{\psi_n}$ is the ψ we want.

Proof of Theorem 3. Clause (1) and (3) in Theorem 3 immediately follow from Theorem 4 (see below) and Clause (2) is obvious.

Proof of Theorem 4. We decompose the proof of Theorem 4 into the following three lemmas.

Lemma 6. *Provided $\mathbb{E}$ is disjointed, $P = (P_1 \| P_2)\backslash\mathcal{I}_{\mathcal{E}_1^{\mathbb{E}}}$ is reachable from $R = ([\mathcal{E}_1^{\mathbb{E}}] \| [\mathcal{G}])\backslash\mathcal{I}_{\mathcal{E}_1^{\mathbb{E}}}$ if and only if $P' = (P_1\sigma \| P_2\sigma)\backslash\mathcal{I}_{\mathcal{E}_2^{\mathbb{E}}}$ is reachable from $R' = ([\mathcal{E}_2^{\mathbb{E}}] \| [\mathcal{G}])\backslash\mathcal{I}_{\mathcal{E}_2^{\mathbb{E}}}$, where $\sigma = \{p, q/q, p\}$ and p, q are the IDs of $\mathcal{E}_1, \mathcal{E}_2$, respectively.*

Proof. Note that by Proposition 1, we have that $P' = P\sigma$ and $R' = R\sigma$, and that $\alpha \in \mathcal{I}_{\mathcal{E}_1^{\mathbb{E}}}$ iff $\alpha\sigma \in \mathcal{I}_{\mathcal{E}_2^{\mathbb{E}}}$. We firstly consider the direction from left to right. The proof is by induction on the number of transitions from R to P. Suppose $Q = (Q_1 \| Q_2)\backslash\mathcal{I}_{\mathcal{E}_1^{\mathbb{E}}}$ is reachable from R and $Q \xrightarrow{\alpha} P$. By induction hypothesis $Q' = Q\sigma = (Q_1\sigma \| Q_2\sigma)\backslash\mathcal{I}_{\mathcal{E}_2^{\mathbb{E}}}$ is reachable from R'. Similar to the proof of Lemma 2, we proceed by two cases on the derivation of $Q \xrightarrow{\alpha} P$. (1) Suppose $Q \xrightarrow{\alpha} P$ is derived from $Q_2 \xrightarrow{\alpha} P_2$ and $\alpha \notin \mathcal{I}_{\mathcal{E}_1^{\mathbb{E}}}$, then $Q\sigma \xrightarrow{\alpha\sigma} P\sigma$ is derived from $Q_2\sigma \xrightarrow{\alpha\sigma} P_2\sigma$ for $\alpha\sigma \notin \mathcal{I}_{\mathcal{E}_2^{\mathbb{E}}}$. (2) Suppose $Q \xrightarrow{\alpha} P$ is derived from $Q_1 \xrightarrow{\tau} P_1$ (thus $\alpha = \tau$), then $Q\sigma \xrightarrow{\tau} P\sigma$ is derived from $Q_1\sigma \xrightarrow{\tau} P_1\sigma$. (3) Suppose $Q \xrightarrow{\alpha} P$ is derived from $Q_1 \xrightarrow{\beta} P_1$ and $Q_2 \xrightarrow{\beta'} P_2$ where β' is dual to β (thus $\alpha = \tau$), similarly we have the same result. Hence $P' = P\sigma$ is reachable from R'. By symmetric of substitution we complete the proof.

Lemma 7. *Provided $\mathbb{E}$ is disjointed, $\mathcal{G}$ is compatible with $\{\mathcal{E}_1^{\mathbb{E}}\}$ against $\mathcal{I}_{\mathcal{E}_1^{\mathbb{E}}}$ if and only if $\mathcal{G}$ is compatible with $\{\mathcal{E}_2^{\mathbb{E}}\}$ against $\mathcal{I}_{\mathcal{E}_2^{\mathbb{E}}}$.*

Proof. Lemma 7 is based on Lemma 6 in the same vein that Lemma 3 is based on Lemma 2.

Lemma 8. *Provided $\mathbb{E}$ is disjointed, $\mathcal{E}_1^{\mathbb{E}}$ and $\mathcal{G}$ are mutually interaction-live against $\mathcal{I}_{\mathcal{E}_1^{\mathbb{E}}}$ if and only if $\mathcal{E}_2^{\mathbb{E}}$ and $\mathcal{G}$ are mutually interaction-live against $\mathcal{I}_{\mathcal{E}_2^{\mathbb{E}}}$.*

Proof. Let $\sigma = \{p, q/q, p\}$ where p, q are the IDs of $\mathcal{E}_1, \mathcal{E}_2$, respectively. We show that if $P = (P_1 \| P_2)\backslash\mathcal{I}_{\mathcal{E}_1^{\mathbb{E}}}$ is reachable from $([\mathcal{E}_1]^{\mathbb{E}} \| [\mathcal{G}])\backslash\mathcal{I}_{\mathcal{E}_1^{\mathbb{E}}}$ and $P_i \xrightarrow{\alpha}$ in P $(i = 1, 2)$, then $P\sigma = (P_1\sigma \| P_2\sigma)\backslash\mathcal{I}_{\mathcal{E}_2^{\mathbb{E}}}$ is reachable from $([\mathcal{E}_2]^{\mathbb{E}} \| [\mathcal{G}])\backslash\mathcal{I}_{\mathcal{E}_2^{\mathbb{E}}}$ and $P_i\sigma \xrightarrow{\alpha\sigma}$ in $P\sigma$. The proof is similar to the proof of Lemma 6.

Proof of Theorem 5. The proof of Theorem 5 is decomposed into three lemmas below whose proofs are similar to those of previous lemmas.

Lemma 9. *Provided $\mathbb{E}$ is excluded, if $\mathcal{G}$ is compatible with $\{\mathcal{E}_i^{\mathbb{E}}\}$ against $\mathcal{I}_{\mathcal{E}_i^{\mathbb{E}}}$ for some $i \in [1, m]$, then $\mathcal{G}$ is compatible with $\{\mathcal{E}_1^{\mathbb{E}}, \ldots, \mathcal{E}_m^{\mathbb{E}}\}$ against $\bigcup_{i=1}^{m} \mathcal{I}_{\mathcal{E}_i^{\mathbb{E}}}$.*

Lemma 10. *If $[\mathcal{E}_1^{\mathbb{E}}]$ is strongly deadlock-free, then $[\mathcal{E}_1^{\mathbb{E}}] \| \cdots \| [\mathcal{E}_m^{\mathbb{E}}]$ is strongly deadlock-free.*

Lemma 11. *Provided $\mathbb{E}$ is excluded, if $\mathcal{E}_i^{\mathbb{E}}$ and $\mathcal{G}$ are mutually interaction-live against $\mathcal{I}_{\mathcal{E}_i^{\mathbb{E}}}$ for some $i \in [1, m]$, then $\mathcal{E}_1^{\mathbb{E}}, \ldots, \mathcal{E}_m^{\mathbb{E}}$ and $\mathcal{G}$ are mutually interaction-live against $\bigcup_{i=1}^{m} \mathcal{I}_{\mathcal{E}_i^{\mathbb{E}}}$.*