Software Architecture Modelling And Analysis In Process Algebra

Guoxin Su

Centre of Quantum Computation & Intelligent Systems
Faculty of Engineering and Information Technology
University of Technology, Sydney

Doctor of Philosophy

2013

Certificate of Authorship/Originality

I certify that the work in this thesis has not previously been submitted for a degree nor has it been submitted as part of requirements for a degree except as fully acknowledged within the text.

I also certify that the thesis has been written by me. Any help that I have received in my research work and the preparation of the thesis itself has been acknowledged.

In addition, I certify that all information sources and literature used are indicated in the thesis.

Signature of Student

Acknowledgements

I am greatly indebted to my principal supervisor Professor Mingsheng Ying, for bringing me from logic to computer science and for creating a versatile academic environment, in which I have undertaken rigorous doctoral training whilst have been able to freely pursue research problems that interested me.

I sincerely appreciate my alternative supervisor Professor Chengqi Zhang, who founded QCIS that benefits every of its members including me and who constantly gave me important advices in person.

I am enormously grateful to my families, especially my wife, who always supports me unconditionally, and my parents, who offer me more than what I can ask for but demand little from me.

I am greatly thankful to the providers of my doctoral scholarships, i.e. the university and my faculty FEIT.

Many thanks to all other people who ever discussed with and enlightened me in person or in one of their seminars, in particular, Prof. Runyao Duan, A/Prof. Sanjiang Li, A/Prof. Yuan Feng, Dr Weiming Liu, Nengku Yu, Yangjia Li, Cheng Guo, and Shenggang Ying.

Contents

Abstract

Description of the principal design of software-intense systems is fundamental in the diversified areas of software architecture. In the past two decades, a variety of Architecture Description Languages (ADLs) emerged and competed to be a suitable modelling medium for software architecture. The high-level syntax of these languages expresses the coarse-grained specifications and properties of interest for the systems, and their formal semantics enables a rigorous analysis of these properties. Notations from the dialect of Process Algebra (PA) are frequently employed by ADLs. The variety of ADLs and the popularity of PA in architectural description motivate the research of PA as formal notations of architectural description, independent of the high-level syntax of ADLs. One benefit of this study is to shed light on some verification techniques for the system properties that ADLs aim to deal with.

This thesis investigates two basic architectural models, which are the *connector-based* and *peer-to-peer* architectures, respectively. This thesis presents a PA-based ADL-like language and a PA-based semantic approach to describe and analyse the first architectural model, and employs session types for π-calculus to deal with the second model. The main contributions are as follows. First, this thesis conducts an *in-depth* study on the use of PA as formal notations for modelling these architectures, and presents various analytic techniques to facilitate the verification of the system properties of interest. Second, this thesis introduces the session type theory into the field of software architecture research and improves the state of the art of the theory by structuring the session description in the communication and integration levels. Last, to enhance the adequacy of PA as *ad hoc* architectural modelling notations, this thesis presents an algorithm to merge a modal extension of PA and the constraint-style specifications.

Nomenclature

Term/Symbol	**Full Spelling/Meaning**
ADL	Architecture Description Language
PA	Process Algebra
LTS	Labelled Transition System
MTS	Modal Transition System
$\mathcal{PA}_1, \mathcal{PA}_2, \mathcal{PA}_s$	PA notations
P, Q, R	processes in PA
α, β, γ	actions in PA
$\tilde{\alpha}.\tilde{\beta}$	sequences of actions
X, Y	process variables
$\longrightarrow, \Longrightarrow$	transitions in PA
A, A'	integrating session types for $\mathcal{PA}_s$
B, B'	communicating session types for $\mathcal{PA}_s$
$\mathfrak{A}$	architecture type
$\mathcal{A}$	architecture instance
$\mathcal{G}$	connector
$\mathfrak{E}$	component type
$\mathcal{E}\langle a \rangle$	component instance
$P^*_{c,i}$	canonical component instance
$\mathcal{F}$	configuration
$\mathcal{F}^*_c$	canonical configuration
$\mathcal{S}, \mathcal{S}_*$	LTS
$\mathcal{M}$	MTS
$S_{\mathcal{S}}, S_{\mathcal{S}*}, S_{\mathcal{M}}$	the set of states of $\mathcal{S}, \mathcal{S}_*, \mathcal{M}$ (resp.)
$ini_{\mathcal{S}}, ini_{\mathcal{S}*}, ini_{\mathcal{M}}$	the initial state of $\mathcal{S}, \mathcal{S}_*, \mathcal{M}$ (resp.)
$A_{\mathcal{S}}, A_{\mathcal{S}*}, A_{\mathcal{M}}$	the set of actions of $\mathcal{S}, \mathcal{S}_*, \mathcal{M}$ (resp.)
$\longrightarrow_{\mathcal{S}}, \longrightarrow_{\mathcal{S}*}, \longrightarrow_{\mathcal{M}}$	transition relations of $\mathcal{S}, \mathcal{S}_*, \mathcal{M}$ (resp.)

Chapter 1

Introduction

Software architecture as developed up to date assembles areas including architectural description[1], architecture decision making, product lines, documentation, industrial case studies, ect. [see WICSA/ECSA'12]. Among others, architectural description or, equivalently, architectural modelling is the cornerstone of software architecture, in the sense that all activities in software architecture cannot be carried out without a suitable method to describe what is considered as *the* architecture of a software system. This is justified historically: the most important pioneer work that raised software architecture as a concept usually came with prototypes of specification languages called Architecture Description Languages (ADLs) later [Shaw and Clements, 2006].

However, as a matter of fact, there is little consensus on the criteria for a language to be a suitable descriptive method of architectures. For example, each ADL is based on its own conceptual architectural framework and captures the more-or-less unique aspect(s) in the framework [Medvidovic and Taylor, 2000]. Fortunately, there is at least a rough, informal picture regarding software architecture, which is generally adopted in the community: the basic building blocks in architectural description are components, which interact with the environment via a set of interfaces or ports and whose internal computations are usually hidden; the architecture of a system is obtained by *binding* the interfaces or ports

[1] In the broad sense, architectural description spans over all aspects of software architecture [IEEE'00]; in the narrow sense, which is adopted by the presented thesis, architectural description focuses on the *principal* design of the system [Medvidovic and Taylor, 2010].

of components in the expected way. In many ADLs, a second group of entities, called connectors, are separated from components and built for purely interaction purposes. Also, for reusability and better documentation, components and connectors (if any) are specified in the instance and type levels. Moreover, the architectures of many, if not all, software systems are supposed to enjoy some important features, such as dynamism (which means the runtime re-configuration of the component binding) and scalability (which refers to the ability to enhance its size by loading up more components). Therefore, in the least sense, a language or a notation is for architectural description if it describes the system in a way that is consistent to the generally accepted understanding on software architecture.

Two of the most important features of ADLs are their coarse-grained syntax and formal semantics. ADLs provide adequate abstractions for modelling large-scale systems whilst reveal sufficient details for the analysis of properties of interest. The formal semantics of ADLs enables an automated way to carry out such analysis (i.e. formal verification). Among other formalisms, the dialect of Process Algebra (PA) [Baeten, 2005; Bergstra et al., 2001] is frequently employed by ADLs as their underpinning notations. The popularity of PA in architectural description comes from several reasons: both the protocol-based behavioural specifications and the parallel composition of components have direct correspondence in the lightweight algebraic syntax of PA; the operation semantics of PA is suitable for reasoning about the runtime behaviours of the system; commercial and non-commercial verification tools using PA as system modelling notations make the PA-based verification accessible.

The diversity of ADLs and the popularity of PA in architectural description motivate the following research approach: we informally isolate a basic software architecture model, the interest in which may be due to its pervasive status in the world of software systems or its advantages over others for overcoming some practical restrictions, and deal with it directly using a notation from the PA family. This research approach is the main thread for the subsequent parts of this thesis. Therefore, the aim of this thesis is not to propose any ready-to-use ADL or similar specification language; instead, the subsequent parts of the thesis study the adequacy of various PA notations (and a type theory in Chapter 5) for architectural description and analysis from the theoretic point of view. The

syntax of each of these PA notations is tailored for either capturing specific aspects of the component behaviours or establishing specific verification techniques for the properties of interests.

It has to be pointed out that, as a specification formalism, the relatively small set of syntactic primitives and rules of PA is a double-edged sword. On the one hand, the PA-based manipulation by human or computers is kept as simple as possible; on the other hand, many coarse-grained data-structural and algorithmic specifications cannot be expressed in PA in a direct and convenient way. This thesis also presents a method to complement PA as *ad hoc* notations in architectural modelling.

Thesis overview Literature review is presented in the next section (Section 2). Some introductory information about PA is provided and followed by three categories of publications corresponding to the work in Chapters 3 and 4, Chapter 5, and Chapter 6, respectively.

Chapter 3 deals with the *connector-based* architecture, in which communications between the components are via the unique connector. This chapter presents a process algebraic ADL-like language, denoted ACDL, to describe this architectural model. ACDL separates the data and control ports in the components, the former of which are for sending data to and receiving data from other components via the central connector, whilst the latter of which are for configurating the communication topology. In ACDL, the central connector in the system architecture is regardless of the number of the components. Various analytic techniques are developed for the compositional and type-based checking of properties such as deadlock-freedom and non-starvation for the modelled system.

Chapter 4 tackles the same architecture but adopts a purely process algebraic semantic approach, which is independent of the coarse-grained ADL-like syntax and extends the previous approach in Chapter 2 in both the modelling and the analytic aspects. The semantic model in this chapter does not distinguish the ports of components, but allows components to instantiate component types by not only parameterisation but also behavioural conformance. The developed analytic techniques mainly demonstrate that the verification state space of the system properties, such as deadlock-freedom, non-starvation, conservation, and

completeness, can be reduced from all possible system configurations to specific ones that are fixed according to the type-level architectural description.

Chapter 5 introduces session types into the modelling and analysis of the *peer-to-peer* architecture. Session types are employed based on two reasons: the global descriptive method of session types facilitates the description of this architectural model; the projection of sessions into private channels in session types ensures the correct port binding between components. This chapter presents a PA notation with a two-level session type system, in which the session types are structured into the communication and integration levels. It demonstrates that, if the system formulated in the PA notation is well-typed then the channel communication is secured and its behaviours follow the session type-based specifications. Moreover, a process-slicing method is constructed to help debug the system quickly.

Chapter 6 complements the previous three chapters, a common goal of which is to employ PA notations as *ad hoc* architectural description mediums. More specifically, this chapter aims to bridge the gap between the low-level algebraic syntax of PA and the coarse-grained specifications in architectural description. The main technical contribution is an algorithm to merge a modal extension of PA, which is equivalent to Modal Transition Systems (MTSs), and the constraint-style specification language.

Chapter 7 concludes the thesis with a summary of the main work in the thesis and discussions on its limitations and problems for further research.

The thesis is partly based on several publications of the author and his supervisors. Chapter 3 is based on [Su et al., 2010]. Chapter 4 is revised from [Su et al., 2012a]. The techniques and results in Chapter 5 are found in [Su et al., 2012b].

Chapter 2

Literature Review

Process Algebra Process Algebra (PA), sometimes called Process Calculi, are a family of algebraic notations originally invented as *the* basic formalism for concurrent computation, just as Church's λ-calculus [Barendregt, 1984] for sequential computation. Around thirty years after the inception of PA witness that the application of PA spans over areas including programming languages, system modelling and analysis, communication protocol verification, and computer security. A comprehensive survey of the theories and applications of PA is beyond the scope of this thesis. A brief history of PA is found in [Baeten, 2005]. Studies on different theoretic aspects of PA are collected in [Bergstra et al., 2001], in which, in terms of relevance to this thesis, of particular interest are chapters by van Glabbeek [2001] and Cleavelandy and Sokolsky [2001]. Two of the representatives in the PA family, i.e. Milner's Calculus of Communicating Systems (CCS) [Milner, 1989] and his π-calculus [Milner et al., 1992a,b; Sangiorgi and Walker, 2001] are most relevant to the PA notations presented in this thesis. Other leading examples of PA include Hoare's Communicating Sequential Processes (CSP) [Hoare, 1985], the Algebra of Communicating Processes (ACP) [Bergstra and Klop, 1985], and the Language Of Temporal Ordering Specification (LOTOS) [Bolognesi and Brinksma, 1987], to name a few.

Process algebraic ADLs and architectural analysis There are a large group of ADLs that can be coined as process algebraic ADLs. This part of the literature review summarises the most typical ones. The emphasis is given to the

role of PA in these ADLs and, in particular, the use of PA in the architectural analysis based on these ADLs. Tool supports are an important aspect to evaluate an ADL; however, this review focuses on the theoretic techniques of an ADL built on the formal notation of a particular PA.

Wright [Allen and Garlan, 1997] is one of the earliest ADLs that extensively deal with the connection mechanism between components from the protocol point of view. Wright emphasises the advantage of separating connectors from components in the realm of entities in software architecture. Connectors specify the interactions of components, whilst components deal with the different aspects of actual computation which together implement the functionality of the system. A Wright description is separated into two levels. In the type level, component and connector types are defined or described. The ingredients of a component type are a set of ports, which are described by protocols, and a behavioural specification, which is usually left as implicit. A connector type consists of a set of roles and a glue, all of which are described by protocols. Roles are corresponded to ports and the glue is the specification of the integration of roles. In the instance level, the instantiation of components and connectors and the attachment of component ports to connector roles are defined.

Protocols are written in the formal notation CSP. Based on the aforementioned model of architecture description and CSP, the important property of deadlock-freedom of the system can be analysed in the following steps. First, the property is formulated. Then, two properties of the connector are formulated: the roles are consistent against the glue and the behaviour of the connector is conservative. These two properties can be verified in an automated way. Thirdly, the compatibility of ports and their attached roles is also formulated and can be checked automatically. Finally, based on a theory, which says that, for a consistent and conservative connector, if all ports are compatible to their roles then the system is deadlock-freedom, the verification of deadlock-freedom of the system (with a consistent and conservative connector) can be reduced to that of the port-role compatibility.

Compared with Wright, PADL [Bernardo et al., 2002] stands closer to the algebraic syntax of PA. The behavioural specification of the component is explicitly defined as processes in CCS, and ports are degenerated from protocols

(as in Wright) to channel names. In the system architecture described in PADL, the end-point channel-based communications between components are realised by attachment, and modular encapsulation is obtained by the restriction operator in CCS. Another feature of PADL, which is different from Wright, is that PADL does not explicitly model connectors. Besides parallel composition of components, PADL also supports hierarchical modelling of nested architectures.

In the analytic aspect, PADL is designed to tackle three kinds of deadlocks caused in the end-point interactions within the system. The first one is due to the incompatibility between two components in a single interaction, which is corresponded to between the port-role compatibility in Wright; the second is due to the incompatibility between two components in multiple interactions [Inverardi et al., 2000]; and the last one is due to the mismatch between a set of components whose interactions form a cyclic topology (and hence cannot be reduced to the second case). For the first and second kind of deadlocks, PADL is able to perform composition compatibility checking, whilst for the third kind, it is able to perform inter-operability. Both notions of compatibility and inter-operability are based on the standard definition observational equivalence in CCS. An important advantage of both checking methods is their diagnostic utility or the detection of the component(s) that lead to the deadlock. Moreover, these two methods can be extended to analyse hierarchical architecture description based on the hiding operator and observational equivalence.

LEAD [Canal et al., 1999] is motivated by the following two considerations: firstly, mobility, i.e. channel-based runtime reconfiguration, is an important property shared by any informal description of software architecture; secondly, although mobility can be captured by π-calculus, π-calculus in general is too low-level to express the essential architectural blocks in a user-friendly way for architectural documentation. In this sense, LEAD is taking the approach opposite to PADL and more close to Wright. Besides runtime reconfiguration, LEAD also supports hierarchical modelling (like PADL) and inheritance, another two user-friendly features. Like other ADLs, in general LEAD describes the architecture of a system in two aspects: in the structural aspect, the modular structure of the system is described; in the behavioural aspect, interfaces of the component are described in a syntactic extension of the standard π-calculus. Also, similar

to Wright, roles are defined in the protocol point of view, and similar to PADL, attachments rather connectors capture the interconnection of interfaces of components.

The architectural analysis based on LEAD [Canal et al., 2001] roughly locates at the second level, i.e. the component level, with respect to the distinction by Bernardo et al. [2002]. However, as the specification of a component in LEAD is not given explicitly (as in PADL) but (partially) 'correctly specified' by a set of roles (protocols, the compatibility issue is between roles of different components. Based on the formality of π-calculus, a compatible definition is formulated and several of its desired properties, such as that an architecture obtained by attaching compatible roles is deadlock-free, are proved. Further, inheritance is also formulated and shown to preserve compatibility.

PiLar [Cuesta et al., 2005] adopts another specific approach: it follows the algebraic syntax of architectural description, but enriches the set of syntactic primitives in the π-calculus to capture architecture-level abstractions and interprets the new primitives according to the informal understanding of the behaviours of these abstractions, such as ports, components, and connection. Another feature of PiLar is that, besides ports, components, and connection, PiLar has another kind of constructs called reifications, which enable a component to monitor other components and change their attachments. In the semantic aspect, all constructs in PiLar into π-calculus by various rules, and therefore, formal architectural analysis can be carried out. Like Pilar, π-ADL [Oquendo, 2004] extends the syntax of π-calculus by various operations and data-structures. Besides the operational semantics, π-ADL also has also a type discipline.

The correspondence between Darwin and PA is two-fold. Firstly, Darwin [Magee et al., 1995] employs π-calculus as its semantics. Compared with the previous ADLs, Darwin describes the architectural of a system from the structural point of view only. Originally, the only behavioural aspect analysed in π-calculus is the binding, which involves several name passing events to model the architecture-level abstract service request and provision. Later in [Magee et al., 1999], an additional behavioural specification, i.e. an extra abstraction called portals, is associated with each component, and the semantic of the portal is given in FSP, a PA notation that combines features of Hoare's CSP and

Milner's CCS.

In some languages usually not classified as ADLs, PA also plays a role. Andova et al. [2011] presented a translation from the coordination language Paradigm [Groenewegen and de Vink, 2002] into ACP so that properties of Paradigm models can be rigorously analysed in the mCRL2 toolset [Groote et al., 2006]. As another example, the observable part of a Web service described in the language BPEL is defined as an abstract (timed) processes. Moreover, PA as a formal tool for system modelling independent of concrete languages. For example, Shen et al. [2005] proposes a semantic model called Dynamic Update Connector (DUC) based CSP for modelling connectors that support dynamic connection of components without causing the running system to halt.

As an important property of ADLs, dynamism or, precisely speaking, various aspects of dynamism are supported by different ADLs. Except PADL, the other four ADLs support some aspects of dynamism. For example, the dynamic version of Wright makes a distinction between the communication and control parts of a connector and models anticipated dynamism via the interactions between the two parts of the connector. Darwin, on the other hand, specifies dynamic systems from a structural point of view. New members of a component type can be created in the runtime and their binding with other components are declared in their component type.

Session types Honda et al. [1998] argued that concurrent programming should be structured in the same way as imperative programming and introduced (bi-party) sessions as the underlying structuring concept. Informally, (biparty) sessions document conversations between two participants. The authors proposed a programming formalism, which is a variant of π-calculus with communication primitives for session establishments, and a type discipline for the formalism. Sessions are interpreted as types in their type discipline, called session types, for a specific group of channels, called session channels and possed by processes in their programming formalism. Session types are behavioural extensions of the conventional IO types for π-calculus and each of them specifies a sequence of message (including channel) types instead of a single one. Hence, the conveyed channels by session channels, if well-typed, are used to convey the prescribed

message types in the prescribed order. Gay and Hole [1999] added sub-typing to session types, and showed that it strengthens the application of session types in specifying a group of client-server protocols.

Carbone et al. [2007] proposed a theoretic basis for bridging global and language descriptive notations for business protocols in web services. They presented two calculi with session types to describe the protocols from the global and local perspectives, respectively, and explored a theory of end-point projection which defines three principles for well-structured global description and a global-local translation with properties of type-preservation, soundness, and completeness. Honda et al. [2008] extended the session types from the biparty setting to the multiparty setting, in which sessions can be established between more than two participants, and showed that various conventional session type-based analytic techniques, such as linearity analysis, can be generalised for π-calculus with multiparty session types.

The subsequent work on session types witnesses a trend of increment on the expressive power to characterise richer conversation structures. For example, Deniélou and Yoshida [2011] extended the multiparty session types to accommodate the runtime change of session participants, i.e. the joining or leaving of participants, after a session is initiated. Yoshida et al. [2010] introduced a finite recursive type constructors into the multiparty session types to express a wide range of processes whose specification structures are parameterised whilst keeping the type checking for the resulting type system decidable. To improve protocol modularisation of session types, Demangeon and Honda [2012] introduced a way to define abstract nested protocols independent of their host protocols such that the host protocols can call the nested ones by passing them arguments such as values, roles, and even (names of) other protocols. In these studies, the enrichment of the session type construction leads to the increment of syntactic primitives in the session calculi. Padovani [2012] proposed an interesting backward approach to session types, in which session types are defined as projected fragments of processes. More specifically, a process is sliced as per channels it uses and session types are a type approximation of the channel-sliced fragments of the process.

MTSs and constraints MTSs were introduced by Larsen and Thomsen [1988] to overcome a restriction of PA or LTSs as a specification formalism. The standard semantic equivalences of PA or LTSs as the potential formulations of the specification-implementation correctness restrict the implementations of a specification as an equivalent (in various senses) class. In contrast, one may want an initial specification to allow non-equivalent implementations and choose some of those implementations based considerations at the later stage. This informal 'refinement' relation can be well-defined between MTSs, which classify all transitions as the necessary and possible ones (the necessary transitions are also possible ones, but not vice versa). If the possible transitions of an MTS identify with the necessary ones, then it is an LTS. A special modal refinement, which is between an MTS and an LTS, is called implementation. Originally, the modal refinement formulated by Larsen and Thomsen [1988] is a strong one, in the sense that the internal transitions are also distinguished. But the relation can be naturally generalised to an observational one, in which the internal transitions are absorbed by the observational ones, as first done by Hüttel and Larsen [1989]. Since software behavioural modelling is an incremental process, it is unrealistic to complete and fix the alphabet of the behavioural model (i.e. its collection of actions) once for all time. Based on this consideration, the weak modal refinement is further relaxed to capture MTSs with different alphabets and called the weak-alphabet modal refinement [Fischbein et al., 2012].[1]

A related formal notion is (operational) model merging, which, informally speaking, constructs a new model by combining features of two models. Model merging was raised in the context of MTSs first by Uchitel and Chechik [2004]. They argued that model merging should result in a minimal modal refinement (MMR) model for two MTSs, and proposed algorithms checking the model consistency (i.e. whether an MMR model exists for two MTSs) and computing the least MMR model (if any) or one of the MMR models. The model merging techniques were generalised by Fischbein et al. [2012] to handle the weak-alphabet modal refinement. The same authors also studied the parallel composition of MTSs and demonstrated that the sets of implementations of two MTSs are subsets of those

[1]The refinement relation adopted in Chapter 6 is actually a special weak-alphabet modal refinement.

of their parallel composition.

Recently, various techniques for synthesising behavioural models (including MTSs) from scenario-based specification (e.g. Message Sequence Charts [ITU, 2000] and UML Sequence Diagrams [Booch et al., 2005]). The theoretic feasibility of this kind of model synthesis comes from the fact that both behavioural models and scenarios (and also standard PA) can be viewed as expressions of traces, that is, sequences of actions. Constraints, such as pre- and post-conditions and initial values, on the other hand make explicitly the values of the variables that defines the state of the system and their dynamics with the actions, and hence, are an important complement for behavioural models. Of course, some high-level behavioural models (statecharts, for example) incorporate the semantic information about states and conditions of actions. But in practice, due to various reasons (such as the separation of concerns and the easiness of formal manipulations) behavioural models usually come without those information. Also, an interesting point is that Object Constraint Language (OCL) [Warmer and Kleppe, 2003], the most notable industrial constraint-style specification language, is invented to supplement UML, which contains statecharts as one kind of behavioural diagrams. The following reports three works along with the research line of behavioural model synthesis from scenarios and addressing the combination of behavioural models and constraints.

Whittle and Schumann [2000] proposed an algorithm to generate statecharts from sequence diagrams. Their algorithm solves three problems: firstly, conflicts of the sequences diagrams in the generation process can be detected and resolved as much as possible; secondly, overlapped fragments of the sequence diagrams are identified and merged; thirdly, the algorithm returns a hierarchical statechart that facilitates human reading and understanding. They argued that, although pure sequence diagrams are intuitive enough, the lack of semantic information makes the interpretation of them difficult, and therefore, they inserted annotations, which are values of state variables and pre- and post-conditions in the OCL style, into the sequence diagrams. Regardless of their rational informal argument, technically, the algorithm depends on the semantic annotation. In other words, PPCs enable conflict detection, state identification, and hierarchical statechart construction for scenarios. For example, as a technical detail, two or above of

the sequence diagrams under consideration are merged at their nodes where the annotated values of state variables are identical and there is at least one incoming transition for each of these nodes labelled by the same action.

Uchitel et al. [2003] pointed out a mismatch between LTSs (and thus statecharts) as complete behavioural models and scenarios as partial specifications, by enhancing the ATM example from [Whittle and Schumann, 2000]. The nature of scenario-based specifications is to depict how the system should work on the instance level instead of its complete behaviours. Therefore, they suggested synthesise the scenarios as a partial LTS (which, in fact, is equivalent to an MTS) instead of a complete one. Krka et al. [2009] proposed an algorithm generating MTSs from scenarios and pre- and post-conditions. Their algorithm accepts the similar inputs as those of Whittle and Schumann [2000] but works in the modal setting. In their algorithm, the pre- and post-conditions form an initial MTS and, by adding scenarios, the final MTS is obtained by a stepwise refining process. Moreover, by specifying the ownership of the state variables in the pre- and post-conditions, the generated MTSs are for the components rather than the whole system.

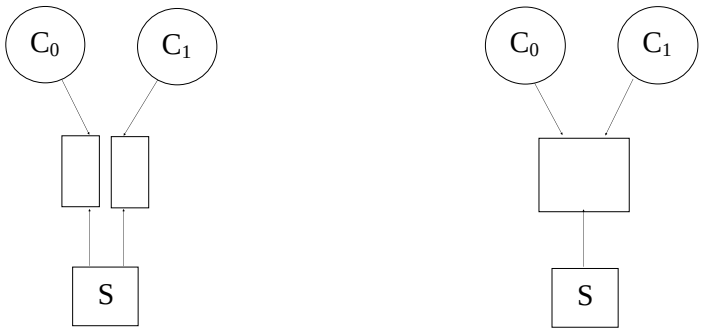
Chapter 3

Connector-based Architecture: Scalability

3.1 Introduction

An architecture is connector-based if its components communicate with each other via its central connector [Shaw, 1993]. This chapter presents a PA-based ADL-like language, called ACDL, to describe systems in this architecture. By describing a system in a way such that the connector is insensitive to the number of components, ACDL facilitates the reusability and scalability of the specifications. Moreover, this chapter provides formal analytic techniques to facilitate the deadlock-freedom and non-starvation checking for connector-based systems described in ACDL. These analytic techniques demonstrate several methods to facilitate the verification of the above two temporal properties. The main work of this chapter is found in [Su et al., 2010].

We begin by providing a connector-based system modelled in Wright to illustrate the ADL approach to architectural description (Section 3.1.1), and then explain a modelling problem if the system is modified (Section 3.1.2). Section 3.2 presents a PA notation and its operational semantics. The textual notation of ACDL and an example are provided in Sections 3.3. The translation of ACDL into PA is found in 3.4. Section 3.5 deploys the formulation of deadlock-freedom and non-starvation, and develops various analytic techniques for the two prop-



(a) dispersed mode (b) centralised mode

Figure 3.1: Procedure-call connectors

erties and explain significance of these techniques by examples. For readability, proof details of the theorems in Section 3.5 are found in Section 3.6. In Section 3.7, the chapter is concluded with discussions on the limitations of the approach and further work.

3.1.1 Background

To illustrate an ADL description of connector-based systems, a simple client-server system described in Wright is presented. In general, Wright has the following three features: firstly, Wright separates connectors as interaction specifications from components as computation specifications; secondly, Wright specifies the component and connector behaviours in the protocol style based on CSP; thridly, Wright describes components and connectors in the levels of types and instances. The client-server system has two clients and one server, as shown in Figure 3.1a. The blocks connecting clients and the server refer to two procedure-call connectors.

The textual description of the system in Wright is given in Figure 3.2, which consists of three parts. The first part of the description defines the type-level contents, i.e. component and connector types. A component type is described as a set of ports (although both component types in the example contain only one port). Ports are interfaces of components' interactions with the environment. A connector type is described as a set of roles and a glue. Roles are connectors' interactions with the environment. The glue specifies how the various roles are

```
Architecture Type {CStype}
  ComponentType {Client}
    Port {clientReq = let X = internalCompute.
          request.result.X in X }
  ComponentType {Server}
    Port {serverPrv = let Y = involve.
          internalCompute.return.Y in Y}
  Connector {ProCall}
    Role {clientRole = let X = request.result.X in X}
    Role {serverRole = let Y = request.result.Y in Y}
    Glue {csGlue = let Z = request.involve.return.
          result.Z in Z}
Architecture {CSinstance}
  Instances
    c1, c_2: Client
    s: Server
    prc1, prc2: ProCall
  Attachments
    c1.clientReq as prc1.clientRole
    c2.clientReq as prc2.clientRole
    s.serverPrv as prc1.serverRole
```

Figure 3.2: A client-server system in Wright

integrated. The second part defines a set of component and connector instances which are entities in the actual configuration of the system. In the third part of the description, the attachment (or binding) of interfaces of components and connectors is defined by linking their ports and roles.

In the semantic aspect, ports of the component and roles and the glue of the connector are translated into processes of CSP. Thus, formal analysis of port-role compatibility and deadlock-freedom of connectors are enabled. If the deadlock-free connector is further shown to be 'conservative', then the deadlock-freedom of the actual system configuration can be reduced to the compatibility checking between each port and its attached role. The practical significance is to facilitate the reuse of connector (type) specifications.

3.1.2 Problem: Reusability and Scalability

Architectural description of connectors in ADLs should allow one to define not only the most basic cases of interactions but also more advanced cases, such as a procedure call together with initialisation to avoid simultaneous calls. Such complex connectors are suitable to be specified in the centralised mode of architectural description, as depicted in Figure 3.1b, in contrast to the dispersed mode, as depicted in Figure 3.1a. Although Wright is expressive enough to define this connector, a careful inspection on Wright reveals that the port-role attachment in Wright is one-one and, therefore, a Wright description of this connector (type) has at least three roles, two of which are for the two clients. In the n-client case, the connector (type) description has $n + 1$ roles, n of which are for the clients. This observation implies a shortcoming of Wright: the connector specification is not reusable when the numbers of components change and it is hard to scale the Wright architectural description for large systems. A solution to the problem is to specify the connector in a way such that the connector is not sensitive to the numbers of the same-type (attached) components. Moreover, it is desirable to establish some analytic techniques that 'leverage' the verification of properties dealt by Wright and other ADLs from the instance-level description to the type-level one. These two considerations are the motivation of this chapter.

3.2 Process Algebra $\mathcal{PA}_1$

Before the presentation of ACDL, a light-weight ADL-like specification language for connector-based systems, a version of PA, denoted $\mathcal{PA}_1$, is firstly defined and will be used as the semantics of ACDL later.[1]

The basic set is an infinite set of names, ranged over by a, b, c, x, y and z. The syntax of $\mathcal{PA}_1$ is provided in Figure 3.3. We distinguish two kinds of channels, i.e. control channels and communication channels. Hence, a channel h is has two forms: it is either a (a control channel) or $a{:}b$ (a communication channel). Communication channels are for data transport whilst control channels are for

[1] As a foretaste, we hints that the subscription of $\mathcal{PA}_1$ implies more versions of PA to be defined in the later chapters.

communication channel passing. Actions (α, β, γ) have five forms: $a!\langle b\rangle$ and $a{:}b!$ are output actions, $a?\langle b\rangle$ and $a{:}b?$ input actions, and τ an internal one. $a!\langle b\rangle.P$ is the process that sends b along with the channel a and then proceeds as P. $a{:}b!.P$ is the process that sends an empty message along with the channel $a{:}b!$ and then proceeds as P. $a?(x).P$ and $a : b?.P$ are correspondent respectively to $a!\langle b\rangle.P$ and $a{:}b!.P$. $a?(b).P$ is the process that receives some b along with a and proceeds as $P\{b/x\}$, which informally means the substitution of b in P by b. $a{:}b?.P$ is the process that receives some empty message along with $a : b$ and proceeds as $P\{b/x\}$. $\tau.P$ is the process that performs an internal action and then proceeds as P. $P + Q$ is the sum of P and Q, and $P \mid Q$ their parallel composition. $\mathbf{0}$ is a nil process and X is a process variable or, simple, a variable. We associate a recursive equation $X \stackrel{\text{def}}{=} P$ to each process variable X.

We use $Proc$ and Act to denote the sets of processes and actions, respectively. We use $\sharp\alpha$ to denote the dual of α, which means that (i) if α is $a?\langle\rangle$ or $a : b?$ then $\sharp\alpha$ is $a!\langle b\rangle$ or $a{:}b!$, respectively, and vice versa, and (2) if $\alpha = \tau$ then $\sharp\alpha = \tau$.

The binders are $a?(x).P$ and $P\backslash h$. The free and bound occurrences of names are standard. For convenience, we assume that in a given process the free and bound names never overlap. Rigorously speaking, $P\{a/x\}$ means the substitution of all free occurrences of x in P by a. The free and bound occurrences of names are standard. We apply the left-associative law to multiple cases of $+$ and $|$. Let $N = \{h_0, \ldots, h_n\}$; $P\backslash N$ abbreviate $P\backslash h_0 \cdots \backslash h_n$. If for the process $h!\langle a\rangle.P$ or $h?(a).P$, if a is not a free name of P, then we just write $h!.P$ or $h?P$.

The *(structural) operational semantics* (SOS) of $\mathcal{PA}_1$ is a set of derivative rules defining a transition relation $\longrightarrow \subseteq Proc \times Act \times Proc$. Following the convention, we write $P \stackrel{\alpha}{\longrightarrow} Q$ if $(P, \alpha, Q) \in \longrightarrow$. The SOS of $\mathcal{PA}_1$ is given in Figure 3.4, together with the symmetric forms of [SUM-1], [PARA-1] and [COM-1]. The SOS compiles a process P in $\mathcal{PA}_1$ into a *labelled transition system* (LTS).

We use $\tilde{\alpha} = \alpha_1, \ldots, \alpha_n$ $(n \leq 0)$ to denote a finite and possibly empty sequence of actions; $\stackrel{\tilde{\alpha}}{\longrightarrow}$ is the *concatenation* of the transitions $\stackrel{\alpha_1}{\longrightarrow}, \ldots, \stackrel{\alpha_n}{\longrightarrow}$. Let $P \Longrightarrow Q$ if $P \stackrel{\tilde{\alpha}}{\longrightarrow} Q$ for any trace $\tilde{\alpha}$. We also say Q is *reachable* from P if $P \Longrightarrow Q$. We write $P \stackrel{\alpha}{\longrightarrow}$ if $P \stackrel{\alpha}{\longrightarrow} Q$ for some Q, write $P \longrightarrow$ if $P \stackrel{\alpha}{\longrightarrow}$ for some α, and write $P \stackrel{*}{\longrightarrow}$ if $P \stackrel{\alpha}{\longrightarrow}$ for some $\alpha \neq \tau$. For convenience of discussions, we also define

process	$P, Q, R ::=$	$a!\langle b\rangle.P$	sending
	$\mid$	$a{:}b!.P$	$-$
	$\mid$	$a?(x).P$	receiving
	$\mid$	$a{:}b?.P$	$-$
	$\mid$	$\tau.P$	silent
	$\mid$	$P + Q$	sum
	$\mid$	$P \mid Q$	parallel
	$\mid$	$\mathbf{0}$	terminal
	$\mid$	X	variable
	$\mid$	$P\backslash h$	hiding
action	$\alpha, \beta, \gamma ::=$	$a!\langle b\rangle$	output
	$\mid$	$a{:}b!$	$-$
	$\mid$	$a?\langle b\rangle$	input
	$\mid$	$a{:}b?$	$-$
	$\mid$	τ	internal
channel	$h ::=$	a	control
	$\mid$	$a{:}b$	communication

Figure 3.3: Syntax of $\mathcal{PA}_1$

the following sets:

$$\begin{aligned} Proc(P) &= \{Q \mid P \Longrightarrow Q\} \\ Act(P) &= \{\alpha \mid Q \xrightarrow{\alpha}, Q \in Proc(P)\} \\ Ch(P) &= \{ch(\alpha) \mid \alpha \in Act(P)\} \end{aligned}$$

We say $\tilde{\alpha}$ is a (finite) *trace* of P if $P \xrightarrow{\tilde{\alpha}}$. An infinite sequence ω of actions is a trace of P if any finite initial fragment of ω is a (finite) trace of P.

Two properties are defined for processes in $\mathcal{PA}_1$ below.

Definition 3.2.1. *P is* deadlock-free*, if $Q \longrightarrow$ for each $Q \in Proc(P)$.*

Let $Q = (P_1 \mid \ldots \mid P_n)\backslash N$; we write $P_i \xrightarrow{\alpha}\!\downarrow Q$ (where $i \in [1, n]$) if $P_i \xrightarrow{\alpha} P_i'$ and $Q \xrightarrow{\beta} (P_1 \mid \ldots P_{i-1} \mid P_i' \mid P_{i+1} \ldots \mid P_n)\backslash N$. For example, $a!\langle b\rangle.P \xrightarrow{a!\langle b\rangle}\!\downarrow (a!\langle b\rangle.P \mid a?(x).Q)\backslash a$. $ch(\alpha)$ is the channel of α.

$$a?(x).P \xrightarrow{a?\langle y\rangle} P\{y/x\} \quad \text{[IN-1]}$$

$$\alpha.P \xrightarrow{\alpha} P \quad \text{if } \alpha \neq a?(x) \quad \text{[ELSE-1]}$$

$$\frac{P \xrightarrow{\alpha} P'}{P + Q \xrightarrow{\alpha} P'} \quad \text{[SUM-1]}$$

$$\frac{P \xrightarrow{\alpha} P'}{P \mid Q \xrightarrow{\alpha} P' \mid Q} \quad \text{[PARA-1]}$$

$$\frac{P \xrightarrow{\alpha} P' \quad Q \xrightarrow{\sharp\alpha} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'} \quad \text{[COM-1]}$$

$$\frac{P \xrightarrow{\alpha} P' \quad ch(\alpha) \neq h}{P\backslash h \xrightarrow{\alpha} P'\backslash h} \quad \text{[HID-1]}$$

$$\frac{P \xrightarrow{\alpha} P' \quad X \stackrel{\text{def}}{=} P}{X \xrightarrow{\alpha} P'} \quad \text{[VAR-1]}$$

Figure 3.4: Operational semantics of $\mathcal{PA}_1$

Definition 3.2.2. *$P_1, \ldots,$ and P_n are* mutually non-starving against (the restriction of) *N, if the following holds: for each $Q = (Q_1 \mid \ldots \mid Q_n)\backslash N \in Proc(P = P_1 \mid \ldots \mid P_n)\backslash N$, if $Q_i \xrightarrow{*}$ for some $i \in [1, n]$, then there is $R = (R_1 \mid \ldots R_{i-1} \mid Q_i \mid R_{i+1} \ldots \mid R_n)\backslash N$ such that $Q \Longrightarrow R$ and $Q_i \xrightarrow{*}\!\!\downarrow R$.*

When $P_1, \ldots, P_n$ represent all components (including the connector) of a system, and N is the set of communication and control channels of the system, non-starvation formulates the property that, this system will not proceed to a situation in which some of its components, if not terminted, can no longer interact with the rest of the system.[1]

3.3 ACDL Language

In this section we present the 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

[1]Note that the non-starvation is strictly weaker than that each component will interact with other components in the system infinitely often.

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

Figure 3.5: ACDL template

types and the connector; the latter specifies the components of corresponding types. The structure of ACDL is given in Figure 3.5.

A *ComponentType* is defined as a function of *Input*, *Output*, *Control* and *Behaviour*. Elements in *Input* and *Output* are indexed by component-type names, and elements in *Control* convey component-type names. *Behaviour* of a *ComponentType* is specified in the process algebraic 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 algebraic 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*. The ACDL description of the system is given in Figure 3.6.

There are three important points to be observed. First, behaviours of components are derived from *Behaviour* 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

```
Architecture Type {CS-TYPE}
  ComponentType {C} %for Client%
    Input {result:C}
    Output {request:C}
    Control {log!<C>, target!<S>}
    Behaviour {CLIENT = internalCompute. log!<C>.
       target!<S>. request:C. result:C. CLIENT}
  ComponentType {S} %for Server%
    Input {involve:S}
    Output {return:S}
    Behaviour {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}
```

Figure 3.6: ACDL description of *CSTYPE* and *SimpleCS*

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.

3.4 Architectural Semantics and Properties

In this section, we bridge ACDL and $\mathcal{PA}_1$. For the convenience of discussions, we set down some notations in Table 3.1.

The semantics $[\mathcal{E}]$ of component $\mathcal{E}$ is the *process variable* whose recursive definition is naturally obtained from the behaviour 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 3.2 gives the semantics of the

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

Table 3.1: Notation Convention

$$
\begin{aligned}
[\mathit{Client}_0] = \;& \tau.\ \mathit{log}!\langle c_0\rangle.\ (\mathit{target}!\langle s_0\rangle.\mathit{request}{:}c_0!.\ \mathit{result}{:}c_0.\ [\mathit{Client}_0] + \\
& \mathit{target}!\langle s_1\rangle.\ \mathit{request}{:}c_0!.\ \mathit{result}{:}c_0.\ [\mathit{Client}_0]) \\
[\mathit{Server}_0] = \;& \mathit{involve}{:}s_0.\ \tau.\ \mathit{return}{:}s_0!.\ [\mathit{Server}_0] \\
[\mathit{ProCall}] = \;& \mathit{log}(x).\ \mathit{target}(y).\ \mathit{request}!{:}x.\ \mathit{involve}{:}y!. \\
& \mathit{return}{:}y.\ \mathit{result}{:}x!.\ [\mathit{ProCall}]
\end{aligned}
$$

Table 3.2: Component and Connector Semantics

components Client_0, Server_0 and the connector *ProCall* in the *SimpleCS* system.

Throughout the remainder of this chapter, 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 $N_{\mathcal{E}_i}$, called *the set of channels of* $\mathcal{E}_i$, be the set of channels in *Input*, *Output* and *Control* of $\mathcal{E}_i$. For example, $N_{c_0} = \{\mathit{log}, \mathit{target}, \mathit{request}{:}c_0, \mathit{result}{:}c_0\}$. The semantics of $\mathcal{A}$ is defined by

$$[\mathcal{A}] = ([\mathcal{E}_1] \mid \ldots \mid [\mathcal{E}_n] \mid [\mathcal{G}])\backslash N_{\mathcal{A}}\ ,$$

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

Property 3.4.1. *(1)* $Ch([\mathcal{E}_i]) \subseteq N_{\mathcal{E}_i}$ *for each* $\mathcal{E}_i$*. (2) If* $\mathcal{E}_i, \mathcal{E}_j$ *and* $\mathcal{E}_k$ *are different components but of the same type, then (i)* $N_{\mathcal{E}_i} \cap N_{\mathcal{E}_j} = N_{\mathcal{E}_i} \cap N_{\mathcal{E}_k}$*, and, (ii)* $N_{\mathcal{E}_j} = \{h\{b/a\} \mid h \in N_{\mathcal{E}_i}\}$ *and* $[\mathcal{E}_i] = [\mathcal{E}_j]\{b/a\}$*, where* a, b *are the names of* $\mathcal{E}_i, \mathcal{E}_j$*, respectively.*

The first clause of Property 3.4.1 justifies that the channels specified in the ports of $\mathcal{E}_i$ cover the channels of the behavioural specification of $\mathcal{E}_i$. The second clause formalises that same-type components share the same *Control* and that

their *Input* and *Output* are parameterised on their names.

Property 3.4.2. *(1) if* $\alpha \in Act([\mathcal{E}_i])$ *then* $\sharp\alpha \notin Act([\mathcal{E}_j])$. *(2) For each* $P = (P_1 \mid \ldots \mid P_n \mid P_{n+1})\backslash N_{\mathcal{A}} \in Proc(([\mathcal{E}_1] \mid \ldots \mid [\mathcal{E}_n] \mid [\mathcal{G}])\backslash N_{\mathcal{A}})$, *if* $P_{n+1} \xrightarrow{\alpha}\!\!\downarrow P$, *then either* $\alpha = \tau$ *or* $ch(\alpha) \in N_{\mathcal{A}}$.

The first clause of Property 3.4.2 confirms that components will not interact themselves. The second clause justifies the definition of $[\mathcal{A}]$ above by showing all channels of $\mathcal{G}$ are in $N_{\mathcal{A}}$, and implies that $\mathcal{G}$ indeed functions to coordinate behaviours of components in $\mathcal{A}$ only. Note that the second clause is weaker than $Ch([\mathcal{G}]) \subseteq N_{\mathcal{A}}$ (because of arbitrary values 'inserted' by input actions).

3.5 Formal Analysis

In this section we develop formal techniques to analyse deadlock-freedom and non-starvation based on the framework of ACDL. To improve the readability we put all the proofs of theorems in Section 3.6. The utilities of the theorems are illustrated by the working example – the *SimpleCS* system.

The following definitions formulate the main properties of architecture instances that are dealt with.

Definition 3.5.1 (Deadlock-freedom)**.** $\mathcal{A}$, $\mathcal{E}$, *or* $\mathcal{G}$, *respectively, is deadlock-free, if* $[\mathcal{A}]$, $[\mathcal{E}]$ *or* $[\mathcal{G}]$, *respectively, is deadlock-free.*

Definition 3.5.2 (Non-starvation)**.** $\mathcal{E}'_1, \ldots, \mathcal{E}'_m$ *(selected from* $\mathcal{E}_1, \ldots, \mathcal{E}_n$*) and* $\mathcal{G}$ *are* mutually non-starving against (the restriction of) N, *if* $[\mathcal{E}'_1], \ldots, [\mathcal{E}'_m]$ *and* $[\mathcal{G}]$ *are mutually non-starving against* N. $\mathcal{A}$ *is* non-starving *if* $\mathcal{E}_1, \ldots, \mathcal{E}_n$ *and* $\mathcal{G}$ *are mutually non-starving against* $N_{\mathcal{A}}$.

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* N, if the following holds: for each $P = (P_1 \mid \cdots \mid P_m \mid P_{m+1})\backslash N \in Proc(([\mathcal{E}'_1] \mid \cdots \mid [\mathcal{E}'_m] \mid [\mathcal{G}])\backslash N)$, if $P_{m+1} \xrightarrow{\alpha}$ and $ch(\alpha) \in N$ then $P_{m+1} \xrightarrow{\alpha}\!\!\downarrow P$.

3.5.1 Compositional Analysis

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 analysis, 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 $N_{\mathcal{E}_j} \cap N_{\mathcal{E}_i} \neq \emptyset$. 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 $N_{\mathcal{P}(\mathcal{E}_i)} = \bigcup_{\mathcal{E}_j \in \mathcal{P}(\mathcal{E}_i)} N_{\mathcal{E}_j}$. Note that by Property 3.4.1 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 analysis: analysis of the architecture are decomposed into analysis 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 3.5.3) more general than that in [Bernardo et al., 2002] where components in an acyclic-topology architecture share no channels due to the definition of PADL.

Theorem 3.5.3. *If $\mathcal{G}$ is deadlock-free and compatible with $\mathcal{P}(\mathcal{E}_i^1)$ against $N_{\mathcal{P}(\mathcal{E}_i^1)}$ for each $i \in [1, m]$, then $\mathcal{A}$ is deadlock-free.*

The proofs of Theorem 3.5.3 and other theorems below are given in the Appendix. Theorem 3.5.3 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 $\{Client_0, Client_1, Client_2\}$ against $N_{c0} \cup N_{c1} \cup N_{c2}$, with $\{Server_0\}$ against N_{s0}, and with $\{Server_1\}$ against N_{s1}, respectively. By decomposing the analysis, 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 3.5.4. *If $\mathcal{G}$ is compatible with $\{\mathcal{E}_i^1, \ldots, \mathcal{E}_i^{k_i}\}$ and $\mathcal{E}_i^1, \ldots, \mathcal{E}_i^{k_i}, \mathcal{G}$ are mutually non-starving for each $i \in [1, m]$, then $\mathcal{A}$ is non-starving.*

Theorem 3.5.4 licenses us to reduce the checking of the non-starvation of $\mathcal{A}$ to the checking of the compatibility and mutual non-starvation of G with respect to the set of components in each partition. For the *SimpleCS* system, to check the non-starvation of $\mathcal{A}$, it suffices to check the compatibility and mutual non-starvation of *ProCall* with respect to (1) $Client_0$, $Client_1, Client_2$ against $N_{c0} \cup N_{c1} \cup N_{c2}$, (2) $Server_0$ against N_{s0}, and (3) $Server_1$ against N_{s1}, respectively. The significance of Theorem 3.5.4 is similar to Theorem 3.5.3, as described above. Theorem 3.5.4 also shows that the checking of non-starvation can be based on the checking of deadlock-freedom according to Theorem 3.5.3.

Theorem 3.5.5. *If $\mathcal{E}_i$ and $\mathcal{E}_j$ are of the same type and $N_{\mathcal{E}_i} \cap N_{\mathcal{E}_j} = \emptyset$, then (1) $\mathcal{G}$ is compatible with $\{\mathcal{E}_i\}$ against $N_{\mathcal{E}_i}$ if and only if $\mathcal{G}$ is compatible with $\{\mathcal{E}_j\}$ against $N_{\mathcal{E}_j}$, and (2) $\mathcal{E}_i$ and $\mathcal{G}$ are mutually non-starving against $N_{\mathcal{E}_i}$ if and only if $\mathcal{E}_j$ and $\mathcal{G}$ are mutually non-starving against $N_{\mathcal{E}_j}$.*

In the *SimpleCS* system, according to Theorem 3.5.5 we have that, for example, *Procall* is compatible with $\{\text{Server}_0\}$ against N_{s0} if and only if *Procall* is compatible with $\{Server_1\}$ against N_{s1}, and $Server_0$ and *ProCall* are mutually non-starving if and only if $Server_1$ and *ProCall* are mutually non-starving. With this theorem we can safely skip over the checking of some parts of an architecture.

3.5.2 Type-based Analysis

The compositional analysis 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 $\mathfrak{A}$ as a collection of component types, together with a connector. In this section we assume $\mathfrak{E} \in \mathfrak{A}$. Note that $\mathfrak{E}$ is disjointed if and only if $\mathcal{P}(\mathcal{E}^{\mathfrak{E}}) = \{\mathcal{E}^{\mathfrak{E}}\}$ for some $\mathcal{P}$. Let $\mathcal{E}^{\mathfrak{E}}$ refer to a component of type $\mathfrak{E}$ and $\mathcal{A}^{\mathfrak{A}}$ an architecture of type $\mathfrak{A}$. We say $\mathfrak{A}$ is *open for deadlock-freedom* on $\mathfrak{E}$, if $\mathcal{A}^{\mathfrak{A}} \cup \{\mathcal{E}^{\mathfrak{E}}\}$ and $\mathcal{A}^{\mathfrak{A}} - \{\mathcal{E}^{\mathfrak{E}}\}$ are deadlock-free whenever $\mathcal{A}^{\mathfrak{A}}$ is deadlock-free;[1] $\mathfrak{A}$ is *open for non-starvation* on $\mathfrak{E}$, if the proposition of the same form holds for non-starvation . Informally, if $\mathfrak{A}$ is open for deadlock-freedom on $\mathfrak{E}$, then every new architecture obtained from a deadlock-free architecture of

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

type $\mathfrak{A}$ via adding, deleting and replacing instances of $\mathfrak{E}$ is also deadlock-free, and if $\mathfrak{A}$ is open for non-starvation on $\mathfrak{E}$, then every new architecture obtained from an non-starving architecture of type $\mathfrak{A}$ via adding, deleting and replacing instances of $\mathfrak{E}$ is also non-starving. We are going to set down some reasonable conditions on component types to ensure these two properties hold.

We say $\mathfrak{E}$ is *disjointed*, if $N_{\mathcal{E}_1^{\mathfrak{E}}} \cap N_{\mathcal{E}_2^{\mathfrak{E}}} = \emptyset$ for any components $\mathcal{E}_1^{\mathfrak{E}}, \mathcal{E}_2^{\mathfrak{E}}$; $\mathfrak{E}$ is *excluded*, if for any component $\mathcal{E}_1^{\mathfrak{E}}, \mathcal{E}_2^{\mathfrak{E}}$ the following holds: for each $P = (P_1 \mid P_2 \mid P_3) \backslash N_{\mathcal{E}_1^{\mathfrak{E}}} \cup N_{\mathcal{E}_2^{\mathfrak{E}}}$ reachable from $([\mathcal{E}_1^{\mathfrak{E}}] \mid [\mathcal{E}_2^{\mathfrak{E}}] \mid [\mathcal{G}]) \backslash\ N_{\mathcal{E}_1^{\mathfrak{E}}} \cup N_{\mathcal{E}_2^{\mathfrak{E}}}$, 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 $[\mathcal{E}_j^{\mathfrak{E}}]$ 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 3.5.6. *Suppose $\mathcal{E}'_1, \ldots, \mathcal{E}'_m$ ($m \geq 2$) are all instances of $\mathfrak{E}$ in $\mathcal{A}^{\mathfrak{A}}$, and $\mathfrak{E}$ is excluded, then: for each $P = (P_1 \mid \cdots \mid P_m \mid P_{m+1}) \backslash \bigcup_{i=1}^{m} N_{\mathcal{P}(\mathcal{E}'_i)}$ reachable from $([\mathcal{E}'_1] \mid \cdots \mid [\mathcal{E}'_m] \mid [\mathcal{G}]) \backslash \bigcup_{i=1}^{m} N_{\mathcal{P}(\mathcal{E}'_i)}$, $P_i \xrightarrow{\alpha}$ 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 labelled by τ only.*

We now demonstrate the relationship between disjointedness, excludedness, deadlock-freedom and non-starvation .

Theorem 3.5.7. *If $\mathfrak{E}$ is disjointed, then $\mathfrak{A}$ is open both for deadlock-freedom and for interaction-liveness on $\mathfrak{E}$.*

Theorem 3.5.7 allows us to add and delete components of disjointed types without making the architecture lose deadlock-freedom and non-starvation , if the architecture enjoyed these two properties originally. We illustrate the application of Theorem 3.5.7 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. non-starving), then we can build new systems based on

SimpleCS that are also deadlock-free (resp. non-starving), such as:

$$MultiServerCS = SimpleCS \cup \{Server_2, \ldots, Server_n\} - \{Server_0, Server_1\} \ .$$

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

$$SingleServerCS = SimpleCS - \{Server_1\} \ .$$

Theorem 3.5.8. *If $\mathfrak{E}$ is excluded, then $\mathfrak{A}$ is open both for deadlock-freedom and for interaction-liveness on $\mathfrak{E}$.*

The significance of Theorem 3.5.8 is just like Theorem 3.5.7 in that it allows us to add and delete components of excluded types without making the architecture lose deadlock-freedom and non-starvation , if the architecture enjoyed these two properties originally. In the *SimpleCS* system, since the *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. non-starving), then we can build new systems based on *SimpleCS* that are also deadlock-free (resp. non-starving) (combining Theorem 3.5.7), such as:

$$MultiCS = MultiSeverCS \cup \{Client_3, \ldots, Client_m\} - \{Client_0, Client_1, Client_2\} \ .$$

If we do not know the deadlock-freedom or non-starvation of *SimpleCS*, we check the following elementary system with one client and one server only (combining Theorem 3.5.7):

$$SingleCS = SingleServerCS - \{Client_1, Client_2\} = \{Client_0, Server_0, ProCall\} \ .$$

In total, what Theorem 3.5.7 and Theorem 3.5.8 tell us is when and to what extent the checking of deadlock-freedom and non-starvation 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 3.5.3 and 3.5.4), the checking of *SingleCS* can be splitted into the checking of the following two

subsystems:

$$SingleCS_C = \{Client_0, ProCall\}, \quad SingleCS_S = \{Server_0, ProCall\} \ .$$

3.5.3 Significance

We have shown how our analytic techniques, i.e. the compositional analysis and the type-based analysis, deal with the deadlock-freedom and the non-starvation 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 N_{s_0}, then we can deduce that *ProCall* is compatible with $Server_1$ against N_{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 analysis. 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 non-starvation 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 analysis facilitate this reusability by showing when and to what extent the system checking of some properties can be undertaken in the type level.

3.6 Proof Details

Proof of Theorem 3.5.3 To improve readability we use $\{[\mathcal{E}_i^1]\}$ refering to $[\mathcal{E}_i^1] \mid \cdots \mid [\mathcal{E}_i^{k_i}]$, for each $i \in [1, m]$. Hence $[\mathcal{A}] = (\{[\mathcal{E}_1^1]\} \mid \cdots \mid \{[\mathcal{E}_m^1]\} \mid [\mathcal{G}])\backslash N_{\mathcal{A}}$. We decompose the proof of Theorem 3.5.3 into the following three lemmas.

Lemma 3.6.1. *If* $P = (P_1 \mid \cdots \mid P_m \mid P_{m+1})\backslash N_{\mathcal{A}}$ *is reachable from* $[\mathcal{A}] = (\{[\mathcal{E}_1^1]\} \mid \cdots \mid \{[\mathcal{E}_m^1]\} \mid [\mathcal{G}])\backslash N_{\mathcal{A}}$, *then* $(P_i \mid P_{m+1})\backslash N_{\mathcal{P}(\mathcal{E}_i^1)}$ *is reachable from* $(\{[\mathcal{E}_i^1]\} \mid [\mathcal{G}])\backslash N_{\mathcal{P}(\mathcal{E}_i^1)}$ *for each* $i \in [1, m]$.

Proof. The proof is by induction on the number of transitions from $[\mathcal{A}]$ to P. Suppose $Q = (Q_1 \mid \cdots \mid Q_m \mid Q_{m+1})\backslash N_{\mathcal{A}}$, $Q \xrightarrow{\alpha} P$, and Q is reachable from $[\mathcal{A}]$. By the first clause of Property 3.4.2 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 $i \in [1, m+1]$ (thus $\alpha = \tau$). Then clearly $(Q_j \mid Q_{m+1})\backslash N_{\mathcal{P}(\mathcal{E}_j^1)}$ 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 $(Q_j \mid Q_{m+1})\backslash N_{\mathcal{P}(\mathcal{E}_j^1)} \xrightarrow{\beta} (P_j \mid P_{m+1})\backslash N_{\mathcal{P}(\mathcal{E}_j^1)}$, $(Q_k \mid Q_{m+1})\backslash N_{\mathcal{P}(\mathcal{E}_k^1)} \xrightarrow{\beta} (Q_k \mid P_{m+1})\backslash N_{\mathcal{P}(\mathcal{E}_k^1)}$, and $Q_k = P_k$ where $k \in [1, m] - \{j\}$. By induction hypothesis, we have the result. □

Lemma 3.6.2. *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* $N_{\mathcal{A}}$.

Proof. Suppose $P = (P_1 \mid \cdots \mid P_m \mid P_{m+1})\backslash N_{\mathcal{A}}$ is reachable from $[\mathcal{A}]$, and $P_{m+1} \xrightarrow{\alpha}$ for some α. By the second clause of Property 3.4.2, $ch(\alpha) \in N_{\mathcal{P}(\mathcal{E}_i^1)}$ for some $i \in [1, m]$. By Lemma 3.6.1, $(P_i \mid P_{m+1})\backslash N_{\mathcal{P}(\mathcal{E}_i^1)}$ is reachable from $([\mathcal{E}_i] \mid [\mathcal{G}])\backslash N_{\mathcal{P}(\mathcal{E}_i^1)}$. Since $\mathcal{G}$ is compatible with $\mathcal{P}(\mathcal{E}_i^1)$, $P_{m+1} \xrightarrow{\alpha}$ in $(\{[\mathcal{E}_i^1]\} \mid [\mathcal{G}])\backslash N_{\mathcal{P}(\mathcal{E}_i^1)}$. Therefore $P_{m+1} \xrightarrow{\alpha}\!\downarrow [\mathcal{A}]$. □

Lemma 3.6.3. *If* $\mathcal{G}$ *is deadlock-free and compatible with* $\mathcal{A} - \{\mathcal{G}\}$ *against* $N_{\mathcal{A}}$, $\mathcal{A}$ *is deadlock-free.*

Proof. The result is obvious by the definitions. □

Proof of Theorem 3.5.4 Similar to the proof of Theorem 3.5.3, we let $\{[\mathcal{E}_i^1]\}$ refering to $[\mathcal{E}_i^1] \mid \cdots \mid [\mathcal{E}_i^{k_i}]$ for each $i \in [1,m]$, and $[\mathcal{A}] = (\{[\mathcal{E}_1^1]\} \mid \cdots \mid \{[\mathcal{E}_m^1]\} \mid [\mathcal{G}])\backslash N_{\mathcal{A}}$. We first show a lemma, which is dual to Lemma 3.6.1:

Lemma 3.6.4. *If $\mathcal{G}$ is compatible with $\{\mathcal{E}_i^1, \ldots, \mathcal{E}_i^{k_i}\}$ for each $i \in [1,m]$, and $(P_i \mid P_{m+1})\backslash N_{\mathcal{P}(\mathcal{E}_i^1)}$ is reachable from $(\{[\mathcal{E}_i^1]\} \mid [\mathcal{E}])\backslash N_{\mathcal{P}(\mathcal{E}_i^1)}$, then there are R, R' such that $(R \mid P_i \mid R' \mid P_{m+1})\backslash N_{\mathcal{P}(\mathcal{E}_i^1)}$ is reachable from $[\mathcal{A}]$.*

Proof of of Theorem 3.5.4. Suppose $\mathcal{G}$ is compatible with $\{\mathcal{E}_i^1, \ldots, \mathcal{E}_i^{k_i}\}$ and $\mathcal{E}_i^1$, $\ldots, \mathcal{E}_i^{k_i}, \mathcal{G}$ are mutually non-starving for each $i \in [1,m]$. Suppose $P_i = P_i^1 \mid \ldots \mid P_i^{k_i}$ for each $i \in [1,m]$ and $P = (P_1 \mid \ldots \mid P_{m+1})\backslash N_{\mathcal{A}} \in Proc([\mathcal{A}])$ and $P_i^j \xrightarrow{}$ where $j \in [1,k_i]$. Then, by Lemma 3.6.1, $(P_i \mid P_{m+1})\backslash N_{\mathcal{P}(\mathcal{E}_i^1)} \in Proc((\{[\mathcal{E}_i^1]\} \mid [\mathcal{G}])\backslash N_{\mathcal{P}(\mathcal{E}_i^1)})$. Thus, there are $Q_i = Q_i^1 \mid \ldots \mid Q_i^{j-1} \mid P_i^j \mid Q_i^{j+1} \mid \ldots Q_i^{k_i}$ and Q_{m+1} such that $(P_i \mid P_{m+1})\backslash N_{\mathcal{P}(\mathcal{E}_i^1)} \Longrightarrow (Q_i \mid Q_{m+1})\backslash N_{\mathcal{P}(\mathcal{E}_i^1)}$ and $P_i^j \xrightarrow{}\downarrow$ $(Q_i \mid Q_{m+1})\backslash N_{\mathcal{P}(\mathcal{E}_i^1)}$. By Lemma 3.6.4, there are R, R' such that $(R \mid Q_i \mid R' \mid Q_{m+1})\backslash N_{\mathcal{A}} \in Proc([\mathcal{A}])$. Thus, $P_i^j \xrightarrow{}\downarrow (R \mid Q_i \mid R' \mid Q_{m+1})\backslash N_{\mathcal{A}}$. □

Proof of Theorem 3.5.5 Clauses (1) and (2) in Theorem 3.5.5 immediately follow from Theorem 3.5.7 (see below).

Proof of Theorem 3.5.7 We decompose the proof of Theorem 3.5.7 into the following three lemmas.

Lemma 3.6.5. *Provided $\mathfrak{E}$ is disjointed, $P = (P_1 \mid P_2)\backslash N_{\mathcal{E}_1^{\mathfrak{E}}}$ is reachable from $R = ([\mathcal{E}_1^{\mathfrak{E}}] \mid [\mathcal{G}])\backslash N_{\mathcal{E}_1^{\mathfrak{E}}}$ if and only if $P' = (P_1\sigma \mid P_2\sigma)\backslash N_{\mathcal{E}_2^{\mathfrak{E}}}$ is reachable from $R' = ([\mathcal{E}_2^{\mathfrak{E}}] \mid [\mathcal{G}])\backslash N_{\mathcal{E}_2^{\mathfrak{E}}}$, where $\sigma = \{p, q/q, p\}$ and p, q are the names of $\mathcal{E}_1, \mathcal{E}_2$, respectively.*

Proof. Note that by Property 3.4.1, we have that $P' = P\sigma$ and $R' = R\sigma$, and that $\alpha \in N_{\mathcal{E}_1^{\mathfrak{E}}}$ iff $\alpha\sigma \in N_{\mathcal{E}_2^{\mathfrak{E}}}$. We firstly consider the direction from left to right. The proof is by induction on the number of transitions from R to P. Suppose $Q = (Q_1 \mid Q_2)\backslash N_{\mathcal{E}_1^{\mathfrak{E}}}$ is reachable from R and $Q \xrightarrow{\alpha} P$. By induction hypothesis $Q' = Q\sigma = (Q_1\sigma \mid Q_2\sigma)\backslash N_{\mathcal{E}_2^{\mathfrak{E}}}$ is reachable from R'. Similar to the proof of Lemma 3.6.1, 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 $\alpha \notin N_{\mathcal{E}_1^{\mathfrak{E}}}$, then $Q\sigma \xrightarrow{\alpha\sigma} P\sigma$ is derived from

$Q_2\sigma \xrightarrow{\alpha\sigma} P_2\sigma$ for $\alpha\sigma \notin N_{\mathcal{E}_2^{\mathfrak{E}}}$. (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 3.6.6. *Provided $\mathfrak{E}$ is disjointed, $\mathcal{G}$ is compatible with $\{\mathcal{E}_1^{\mathfrak{E}}\}$ against $N_{\mathcal{E}_1^{\mathfrak{E}}}$ if and only if $\mathcal{G}$ is compatible with $\{\mathcal{E}_2^{\mathfrak{E}}\}$ against $N_{\mathcal{E}_2^{\mathfrak{E}}}$.*

Proof. Lemma 3.6.6 is based on Lemma 3.6.5 in the same vein that Lemma 3.6.2 is based on Lemma 3.6.1. □

Lemma 3.6.7. *Provided $\mathfrak{E}$ is disjointed, $\mathcal{E}_1^{\mathfrak{E}}$ and $\mathcal{G}$ are mutually non-starving against $N_{\mathcal{E}_1^{\mathfrak{E}}}$ if and only if $\mathcal{E}_2^{\mathfrak{E}}$ and $\mathcal{G}$ are mutually non-starving against $N_{\mathcal{E}_2^{\mathfrak{E}}}$.*

Proof. Let $\sigma = \{p, q/q, p\}$ where p, q are the names of $\mathcal{E}_1, \mathcal{E}_2$, respectively. We show that if $P = (P_1 \mid P_2)\backslash N_{\mathcal{E}_1^{\mathfrak{E}}}$ is reachable from $([\mathcal{E}_1]^{\mathfrak{E}} \mid [\mathcal{G}])\backslash N_{\mathcal{E}_1^{\mathfrak{E}}}$ and $P_i \xrightarrow{\alpha}$ in P $(i = 1, 2)$, then $P\sigma = (P_1\sigma \mid P_2\sigma)\backslash N_{\mathcal{E}_2^{\mathfrak{E}}}$ is reachable from $([\mathcal{E}_2]^{\mathfrak{E}} \mid [\mathcal{G}])\backslash N_{\mathcal{E}_2^{\mathfrak{E}}}$ and $P_i\sigma \xrightarrow{\alpha\sigma}$ in $P\sigma$. The proof is similar to the proof of Lemma 3.6.5. □

Proof of Theorem 3.5.8 The proof of Theorem 3.5.8 is decomposed into three lemmas below whose proofs are similar to those of previous lemmas.

Lemma 3.6.8. *Provided $\mathfrak{E}$ is excluded, $P = (P_1 \mid P_2)\backslash N_{\mathcal{E}_1^{\mathfrak{E}}}$ is reachable from $R = ([\mathcal{E}_1^{\mathfrak{E}}] \mid [\mathcal{G}])\backslash N_{\mathcal{E}_1^{\mathfrak{E}}}$ if and only if $P' = (P_1\sigma \mid P_2\sigma)\backslash N_{\mathcal{E}_2^{\mathfrak{E}}}$ is reachable from $R' = ([\mathcal{E}_2^{\mathfrak{E}}] \mid [\mathcal{G}])\backslash N_{\mathcal{E}_2^{\mathfrak{E}}}$, where $\sigma = \{p, q/q, p\}$ and p, q are the names of $\mathcal{E}_1, \mathcal{E}_2$, respectively.*

Lemma 3.6.9. *Provided $\mathfrak{E}$ is excluded, if $\mathcal{G}$ is compatible with $\{\mathcal{E}_i^{\mathfrak{E}}\}$ against $N_{\mathcal{E}_i^{\mathfrak{E}}}$ for some $i \in [1, m]$, then $\mathcal{G}$ is compatible with $\{\mathcal{E}_1^{\mathfrak{E}}, \ldots, \mathcal{E}_m^{\mathfrak{E}}\}$ against $\bigcup_{i=1}^m N_{\mathcal{E}_i^{\mathfrak{E}}}$.*

Lemma 3.6.10. *Provided $\mathfrak{E}$ is excluded, if $\mathcal{E}_i^{\mathfrak{E}}$ and $\mathcal{G}$ are mutually non-starving against $N_{\mathcal{E}_i^{\mathfrak{E}}}$ for some $i \in [1, m]$, then $\mathcal{E}_1^{\mathfrak{E}}, \ldots, \mathcal{E}_m^{\mathfrak{E}}$ and $\mathcal{G}$ are mutually non-starving against $\bigcup_{i=1}^m N_{\mathcal{E}_i^{\mathfrak{E}}}$.*

3.7 Related Work and Discussions

We conclude this chapter with discussions on the novelty of our approach, the related work and the limitations that are addressed in the subsequent parts of this thesis.

ACDL is novel in its way to describes connector-based architectures. 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.

Another innovation is the analytic techniques of temporal properties of an architecture, which are developed based on ACDL. By employing π-calculus 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 has been carried out in the previous literature, such as [Allen and Garlan, 1997] and [Bernardo et al., 2002]. But the main novelty of our method is that, first, more general than the acyclic/cyclic sharp division in [Bernardo et al., 2002], it uses a partition on the whole set of components in architecture to achieve the finest-grain of the compositional analysis, and second, 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 [Inverardi et al., 2000] and [Aldini and Bernardo, 2005] did indicate that their methods apply to the analysis of liveness properties, ours, which seriously deals with interaction-liveness, is still enlightening.

The architectural topology that we consider, i.e. centralized-mode architectural connection, is close to the coordinator-based architecture style investigated in [Tivoli and Inverardi, 2008], 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 [Magee et al., 1995], 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 [Oquendo, 2004] is a powerful formal specification language based on the high-order typed π-calculus, and is equipped with the analysis language π-AAL 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 in this chapter.

In spite of its novelty and its associated useful verification techniques, ACDL needs further development to enhance its practical utility. One obvious problem is its lack of tool support. There are two ways to equip ACDL with supporting tools. One way is to directly develop a toolkit for ACDL. The other way is to couple ACDL with a mature process-algebraic ADL such as Wright and PADL, by building a translator from ACDL syntax to the target language. For example, an ACDL description of the connector in Figure 3.1b can be transformed into a Wright description (with more roles).

On the theoretic aspect, the work in this chapter has the following limitations. Firstly, ACDL assumes the connector-based architecture. Secondly, ACDL does not handle the *dynamic* architecture, as the configuration of a system in the ACDL description is fixed. Thirdly, the translation of ACDL into $\mathcal{PA}_1$ implies

some behavioural restrictions of components. For example, the control action $target\langle C\rangle$ of the client component type description in Figure 3.6 implies that all clients *must* attempt to call all servers, as C ranges over the names of all servers. Fourthly, the instantiation of components with their types is purely parameterisation. Fifthly, a part of the analytic techniques is based on an overly restricted condition 'excludedness' (Page 27). Finally, just like other process-algebraic ADLs, ACDL, in spite of its suitability for formal reasoning, is too abstract to express some high-level algorithmic and data structures in the coarse-grained system specifications in a convenient way. The second to fifth limitations are overcome by the approach in Chapter 4. The first limitation motivates the work in Chapter 5. The last one is partially addressed in Chapter 6.

Chapter 4

Connector-based Architecture: Dynamism

4.1 Introduction

Dynamism is an essential requirement for many software systems. Dynamism refers to the runtime re-configuration of components in the architecture. This chapter studies the modelling and analysis of the dynamic connector-based architecture. This chapter presents a process-algebraic semantic approach to formalise the dynamic connector-based architecture and develops various analytic techniques to facilitate the checking of some system properties of interest, regardless of the uncertainty of the system configuration. The work in the chapter overcome the limitations of that in the last chapter in various aspects (see 3.7). Moreover, besides deadlock-freedom and non-starvation, the system properties of conservation and completeness are also dealt with, the former of which says that the behaviour of the whole system can be reduced to that of the connector, whilst the latter of which states that the connector does not overly restrict the behaviour of the whole system. The most important technical results in this chapter can be summarised as follows: in spite of of semantics variances between component types and instances, the verification state space of the four properties in a dynamic connector-based system can be reduced from the entire universe of possible system configurations to a *fixed* configuration.

The background of architectural description is provided in Section 4.1.1. Problems that motivate the formalism and formal analysis are explained in Section 4.1.2. A version of PA underlaying the semantic model is defined in Section 4.2. A running example of this chapter is provided in Section 4.3. Section 4.4 presents the semantic model together with formulations of the studied behavioural properties. Section 4.5 is dedicated to the formal analysis. For readability, proofs details of the theorems in Section 4.5 are given in Section 4.6. Section 4.7 concludes this chapter.

4.1.1 Background

Although PA has been used as formal notations for architectural modelling and analysis, independent of the high-level syntax of a specification language, the process-algebraic semantic model in this chapter is based on the conceptual framework of software architecture shared by the majority of ADLs. In the last chapter (Chapter 3), the ADL Wright is used as an example to illustrate the ADL description of software architecture. Here, two basic features of ADLs are summarised:

- (Most) ADLs emphasise the distinction of components, which are the computation loci, and connectors, which facilitate the interactions of components.
- ADLs separate the descriptions of software architectures into the levels of types and instances. For example, abstract components or component types are provided in the type level, and component instantiations in terms of parameterisation and type conformance are specified in the instance level.

Dynamism of a system refers to the runtime re-configuration of its components. In other words, a system is dynamic if the communication topology between components can change in the runtime. Many ADLs (e.g. Darwin [Magee et al., 1995] and C2 [Medvidovic et al., 1999]) support the component-aspect dynamism and impose no number restriction on instantiations of components.

4.1.2 Problem: Uncertainty of Component Numbers

For connector-based dynamic systems, because all (direct) communications are between each component and the connector in the system, re-configuration is equivalent to the insertion and removal of components with respect to the connector. However, even in the reduced form of dynamic software architecture, the uncertainty of the runtime system configuration leads to a difficulty in the verification of behavioural properties: the verification state space equals to the universe of all possible configurations that may happen in the runtime.

Fortunately, in the conceptual framework of (some) ADLs, dynamism is governed and restricted by the type-level fragment of architectural description. For example, in Wright (and similarly in ACDL), the behaviours of component types are first described before specifying the component instantiation. The last chapter investigates the applicability to reduce the verification state space by using the type-level information of architectural description.

ACDL and the associated analytic techniques are able to address at least some part of this verification difficulty. However, they have several shortcomings: firstly, ACDL is not designed to describe dynamic systems, as the component instantiation is fixed at the compile time; secondly, the syntax of ACDL and its semantic translation into $\mathcal{PA}_1$ impose some (sometimes unnecessary) restrictions on the behaviours and, hence, its formal analysis only applies to a limited group of systems. Finally, the relation of the type-instance component conformance is just parameterisation in ACDL, but some semantic variance is often allowed between a component type and its instances, such as the 'type conformance' in C2.

This chapter presents a semantic approach to modelling dynamic connector-based systems, together with a set of formal analytic techniques for the verification of some basic properties of interest, e.g. deadlock-freedom, non-starvation, conservation, and completeness, where conservation says that behaviours of the system can be reduced to those of connectors, whilst completeness states that connectors do not overly restrict behaviours of the system.

process	$P, Q ::=$	$P \times Q$	times
	$\mid$	$P \parallel Q$	parallel
	$\mid$	$\mathbf{0}$	terminal
	$\mid$	X	variable
	$M, M' ::=$	$M + M'$	sum
	$\mid$	$\lambda?.P$	receiving
	$\mid$	$\lambda!.P$	sending
action	$\alpha, \beta, \gamma ::=$	$\lambda?$	input
	$\mid$	$\lambda!$	output
	$\mid$	τ	internal
message	$\lambda ::=$	$\langle a_1, \ldots, a_k \rangle$	

Figure 4.1: Syntax of $\mathcal{PA}_2$

4.2 Process Algebra $\mathcal{PA}_2$

The syntax of $\mathcal{PA}_2$, the PA tailored as the formal basis of the semantic model, is given in Figure 4.1. Like $\mathcal{PA}_1$, the basic sets for $\mathcal{PA}_2$ are a set of names (a, b, c, x, y, z) and a set of process variables (X). But $\mathcal{PA}_2$ has several features that are different from $\mathcal{PA}_1$. The first different feature is non-'value-passing' behaviours in $\mathcal{PA}_2$ in the sense that behaviours induced by $a?(x)$ in $\mathcal{PA}_1$. Also, the parallel composition $\mid$ in $\mathcal{PA}_1$ is separated as the production $\times$ and the synchronisation composition $\parallel$ in $\mathcal{PA}_2$. Another feature of $\mathcal{PA}_2$ is that channels are sequences of names and coined as messages. Lastly, the hidden operation $\backslash$ in $\mathcal{PA}_1$ is not needed.

The more specific explanations on the syntax of $\mathcal{PA}_2$ follow. $\lambda?, \lambda!, \tau$ denote input, output, and internal actions, respectively. $P + Q$ is the sum of P and Q, $P \times Q$ their interleaving composition, and $P \parallel Q$ their synchronized composition. Let Act denote the set of actions, $Proc$ denote the set of processes, $Proc'$ the set of processes constructed *without* $\parallel$ (i.e. '$\parallel$'-free processes), and $Proc''$ the set of processes constructed *without* $\times$ or $\parallel$ (i.e. '$\parallel$'- and '$\times$'-free processes) in $\mathcal{PA}_2$. For each process variable X we define a *recursive equation* $X \stackrel{\text{def}}{=} M$ where $M \in Proc''$. When writing $P = P_1 \dagger P_2 \dagger \ldots \dagger P_n$ where $\dagger \in \{+, \parallel\}$, we assume

the law of association to the left for P.

Like in the last chapter, we use N to denote a set of names, and $\sharp\alpha$ is the dual action of α, i.e., if α is $\lambda!$ (resp. $\lambda?$) then let $\sharp\alpha$ be $\lambda?$ (resp. $\lambda!$). $P\{c/a\}$ is the process obtained from P by substitution of c for a. We also define *abstract* processes: $P(a : N)$ is an abstract process that intends to behave as $P\{c/a\}$ for any $c \in N$. If $P(a : N)$ is clear in the context, we use $P\langle c\rangle$ to denote $P\{c/a\}$. Suppose $a_i \neq a_j$ for each $1 \leq i \neq j \leq n$. $P\{c_1/a_1, \ldots c_n/a_n\}$ denotes the simultaneous substitution of c_1 to c_n for a_1 to a_n in P. $P(a_1 : A_1, \ldots a_n : A_n)$ denotes $P(a_1 : A_1) \ldots (a_n : A_n)$ (an n-time abstract process) which is intended to behave as $P\{c_1/a_1, \ldots c_n/a_n\}$ where $c_i \in A_i$ for each $1 \leq i \leq n$.

The operational semantics of PA are rules in Figure 4.2, plus the symmetric forms of the rules [SUM-2], [TIMES-2], and [COM-2]. Let $\widetilde{\alpha} = \alpha_1, \ldots, \alpha_n$ $(n \geq 0)$; $\sharp\widetilde{\alpha}$ is $\sharp\alpha_1, \ldots, \sharp\alpha_n$ and $\xrightarrow{\tilde{\alpha}}$ is the *concatenation* of the labelled transitions $\xrightarrow{\alpha_1}$ $, \ldots, \xrightarrow{\alpha_n}$. $P \xrightarrow{\alpha}$ means $P \xrightarrow{\alpha} Q$ for some Q, and $P \xrightarrow{\tilde{\alpha}}$ means $P \xrightarrow{\tilde{\alpha}} Q$ for some Q. $P \longrightarrow$ if $P \xrightarrow{\alpha}$ for some α. If $P \xrightarrow{\tilde{\alpha}}$, we say $\tilde{\alpha}$ is a (finite) *trace* of P. Just as in the last chapter, for the convenience of discussions, we let

$$Proc(P) = \{Q \mid P \Longrightarrow Q\}$$
$$Act(P) = \{\alpha \mid Q \xrightarrow{\alpha}, Q \in Proc(P)\}$$

The following lemmas are important to the techniques in this chapter.

Lemma 4.2.1. *For each $P \in Proc$, $Proc(P)$ and $Act(P)$ are finite.*[1]

In words, Lemma 4.2.1 states that the state space and the action space of any process in $\mathcal{PA}_2$ are finite.

Lemma 4.2.2. *If $P \in Proc'$ then $\tau \notin Act(P)$.*

Lemma 4.2.2 says that if P is '$\|$'-free then it is 'τ'-free. Analysis of behaviours of 'τ'-free processes is technically simpler than that of general processes.

P and Q are (strongly) *bisimilar*, denoted by $P \simeq Q$, if there is a relation $R \subseteq Proc \times Proc$ such that $\langle P, Q\rangle \in R$ and

[1] In standard CCS, the set of reachable processes of a process could be infinite. But for our process calculus, M in the recursive equation $X \stackrel{\text{def}}{=} M$ is restricted to be '$\|$'- and '$\times$'-free, and therefore, $Proc(P)$ is finite.

$$\alpha.P \xrightarrow{\alpha} P \qquad \text{[ACT-2]}$$

$$\frac{P \xrightarrow{\alpha} P' \quad X \stackrel{\text{def}}{=} P}{X \xrightarrow{\alpha} P'} \qquad \text{[VAR-2]}$$

$$\frac{P \xrightarrow{\alpha} P'}{P + Q \xrightarrow{\alpha} P'} \qquad \text{[SUM-2]}$$

$$\frac{P \xrightarrow{\alpha} P'}{P \times Q \xrightarrow{\alpha} P' \times Q} \qquad \text{[TIMES-2]}$$

$$\frac{P \xrightarrow{\lambda!} P' \quad Q \xrightarrow{\lambda?} Q'}{P \parallel Q \xrightarrow{\tau} P' \parallel Q'} \qquad \text{[COM-2]}$$

Figure 4.2: Operational semantics of $\mathcal{PA}_2$

- if $\langle P', Q' \rangle \in R$ and $P' \xrightarrow{\alpha} P''$, then there is Q'' such that $Q' \xrightarrow{\alpha} Q''$ and $\langle P'', Q'' \rangle \in R$
- if $\langle P', Q' \rangle \in R$ and $Q' \xrightarrow{\alpha} Q''$, then there is P'' such that $P' \xrightarrow{\alpha} P''$ and $\langle P'', Q'' \rangle \in R$

P is *deterministic* if it enjoys the following property: if $P \xrightarrow{\widetilde{\alpha}} Q$ and $P \xrightarrow{\widetilde{\alpha}} Q'$ for any $\widetilde{\alpha}$, then $Q = Q'$.

Lemma 4.2.3. *For each process P, there is a deterministic process Q such that $P \xrightarrow{\widetilde{\alpha}}$ if and only if $Q \xrightarrow{\widetilde{\alpha}}$ for any $\widetilde{\alpha}$.*

Intuitively, Lemma 4.2.3 says that trace equivalence relation does not discriminate between determinism and non-determinism of processes.

4.3 Running Example

The system that we use as the motivating and running example is a client-server system. The most important feature of this system is that all clients and servers are linked by a middleware that is responsible for transferring messages between them. An essential requirement for this system is (component-aspect) dynamism: clients or servers can be added to or removed from the system. A typical scenario is depicted in Figure 4.3 to illustrate the functionality of the system. Since

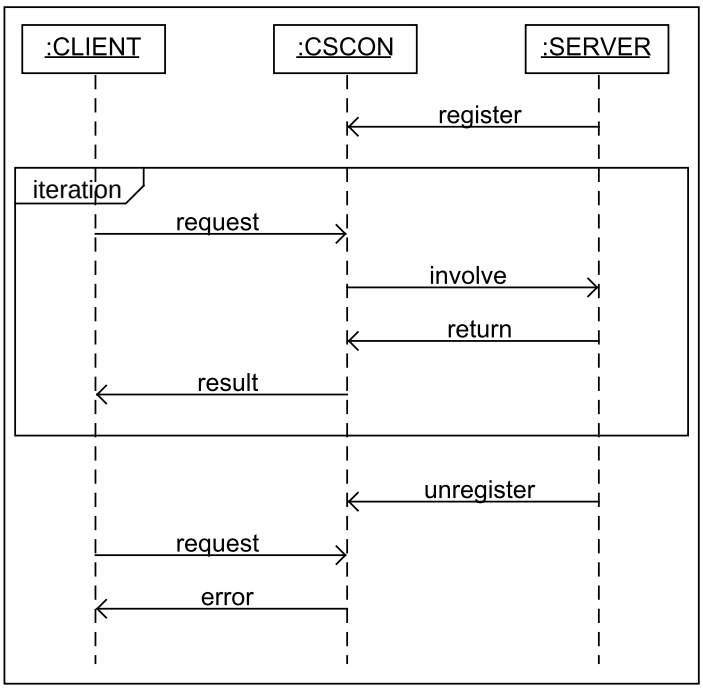


Figure 4.3: A scenario of the client-server system

behavioural modeling and analysis are the central topic of this paper, we focus on behavioural descriptions of the system. Following ADLs, the system's architecture is described in the type and instance levels, in the style of update rules of control-state *Abstract State Machines* (ASMs) [Börger and Stärk, 2003]. Figure 4.4 specifies the architecture type $CSTYPE_d$ of the system, which consists of a client type *CLIENT*, a server type *SERVER*, and a connector *CSCON*.

We first walk through the specification of the client type *CLIENT*. *CLIENT* has two typed parameters, specified by the expression '$c : clt, s : sev$', where *clt* and *sev* are the name spaces of all clients and servers, respectively. *CLIENT* has three control states, which are values of the variable *state*. 1 is the initial state and 0 is the terminal one. In state 1, *CLIENT* sends message $\langle request, c, s\rangle$ (the symbol '!' indicates that the message is to be sent) and then arrives at state 2. In state 2, *CLIENT* receives message $\langle result, c, s\rangle$ or $\langle error, c, s\rangle$ (the symbol '?' indicates that the message is to be received), and arrives at the terminal state.

Several places in the specification of the server type *SERVER* and the con-

Component type $CLIENT(c : clt, s : sev)$:

if $state = 1$ **then** $\langle request, c, s\rangle!$ **and** $state := 2$

if $state = 2$ **then**

if true then $\langle result, c, s\rangle?$ **and** $state := 0$

if true then $\langle error, c, s\rangle?$ **and** $state := 0$

Component type $SERVER(s : sev)$:

if $state = 1$ **then** $\langle register, s\rangle!$ **and** $state := 2$

if $state = 2$ **then**

if true then $\langle involve, x : clt\rangle?$ **and** enqueue(x, Que)

if empty(Que) = 'n' **then let** $y =$ head(Que)

and $\langle return, y\rangle!$ **and** dequeue(Que)

if empty(Que) = 'y' **then** $\langle unregister, s\rangle!$ **and** $state := 0$

Connector $CSCON$:

if $state_1 = 1$ **then** $\langle request, x : clt, y : sev\rangle?$ **and** $state_1 := 2$

if $state_1 = 2$ **then**

if $y \in RegSev$ **then** $\langle involve, x, y\rangle!$ **and** $state_1 := 1$

else $\langle error, x, y\rangle!$ **and** $state_1 := 1$

if $state_2 = 1$ **then** $\langle return, z : clt, w : sev\rangle?$ **and** $state_2 := 2$

if $state_2 = 2$ **then** $\langle result, z, w\rangle!$ **and** $state_2 := 1$

if $state_3 = 1$ **then**

if true then $\langle register, v : sev\rangle?$ **and** $RegSev := RegSev \cup \{v\}$

if true then $\langle unregister, u : sev\rangle?$ **and** $RegSev := RegSev \backslash \{u\}$

Figure 4.4: Architecture type $CSTYPE_d$ of the client-server system

nector *CSCON* are worthy of notice. In state 2, *SERVER* has three branches to proceed that are guarded by conditions. In the first branch, the condition is an unconditional truth, so it can perform $\langle involve, x : clt\rangle?$ and enqueue(x, Que) unconditionally. The input event $\langle involve, x : clt\rangle?$ is slightly different from the aforementioned input event of *CLIENT*, because it has a variable x ranging over clt (the name space of clients). If this event happens, *SERVER* receives a message $\langle involve, a\rangle$ such that $a \in clt$ and assigns a to x. enqueue(x, Que) is an internal action, meaning adding the value of x into the queue whose current value is Que. In the second branch, the action $\langle return, y\rangle!$ means sending a message $\langle return, c\rangle$ such that c is the value of y. *CSCON* has three controlled variables. Hence, it can be considered as the interleaving product of three sub-ASMs. Also, events of *CLIENT* and *SERVER* are dual to those of *CSCON*.

A difficulty inherent in the dynamism of the system is that, even if the architecture type $CSTYPE_d$ is provided, the actual configuration of the system is flexible. The configuration could consist of any clients and servers whose names are within clt and sev, respectively. More trick are the semantic variances between components and component types. For example, Figure 4.5 documents two clients and a server. In spite of conformance, we see a degree of semantic difference between $Client_1$, $Client_2$ and *CLIENT*, and between *Server* and *SERVER*. More specifically, *CLIENT* describes *one* communication session as state 0 is the terminal state; $Client_1$ designates a server named s_1 and iterates the session infinitely; $Client_2$ firstly requests a designated server named s_1, and then requests a random server if no server named s_1 is available. The semantics of *Server* is different from that of *SERVER* in two places: First, after performing the *unregister* event, *SERVER* terminates but *Server* has the *register* event in future; secondly, *Server* monitors the value of *update* and decides whether to *unregister* or *register*.

Therefore, the verification of, say, the following properties requires exhausting all possible configurations:

- whether the system is deadlock-free;
- whether each component, if not terminated, will not be deprived of the right to interact with the connector;

Component instance $Client_1 : CLIENT$:

if $state = 1$ **then** $\langle request, c_1, s_1 \rangle!$ **and** $state := 2$

if $state = 2$ **then**

if true then $\langle result, c_1, s_1 \rangle?$ **and** $state := 1$

if true then $\langle error, c_1, s_1 \rangle?$ **and** $state := 1$

Component instance $Client_2 : CLIENT$:

if $state = 1$ **then** $\langle request, c_2, s_1 \rangle!$ **and** $state := 2$

if $state = 2$ **then**

if true then $\langle result, c_2, s_1 \rangle?$ **and** $state := 1$

if true then $\langle error, c_2, s_1 \rangle?$ **and** $state := 3$

if $state = 3$ **then choose** $any \in sev$

and $\langle request, c_2, any \rangle!$ **and** $state := 4$

if $state = 4$ **then**

if true then $\langle result, clt_2, any \rangle?$ **and** $state := 1$

if true then $\langle error, clt_2, any \rangle?$ **and** $state := 3$

Component instance $Server : SERVER$:

if $state = 1$ **then** $\langle register, s_1 \rangle!$ **and** $state := 2$

if $state = 2$ **then**

if empty(Que) = 'y' **and** $upgrade$ = 'ready'
then $\langle unregister, s_1 \rangle!$ **and** $state := 3$

if true then $\langle involve, x : clt \rangle?$ **and** enqueue(x, Que)

if empty(Que) = 'n' **then let** y = head(Que)
and $\langle return, y \rangle!$ **and** dequeue(Que)

if $state = 3$ **then**

if $upgrade$ = 'done' **then** $\langle register, s \rangle!$ **and** $state := 2$

Figure 4.5: Component examples of the client-server system

- whether *CSCON* restricts the system's behaviours;
- whether all behaviours of each component is realizable, given a suitable configuration.

It should be noted that the semantic variances between client and servers and their respective types discussed above further complicates the matter, making it more than a number-sensitivity problem of instance copies. Our semantic model and formal techniques aim to address this problem. The component conformance relation formulated in our model handles those semantic variances. Moreover, our analytic techniques demonstrate that the verification of properties, such as the above four, actually depends on a specific configuration of the system that can be deduced from descriptions in the architecture type $CSTYPE_d$.

Translation of ASM to PA High-level specification languages such as ASMs have the advantage of capturing the algorithmic designs and data structures in a convenient way, which by contrast is the shortcoming of light-weight formalisms such as process algebra. We supplement some information about our selected fragment of ASMs and illustrate a way to translate ASM update-rules into recursive equations in process algebra. For a comprehensive introduction to ASMs, we refer to [Börger and Stärk, 2003].

The presentation of ASM update rules for *CSsystem* implicitly presupposes an ASM signature, i.e. a collection of symbols, and an initial interpretation, such as one in the first-order logic. For example, in the specification of *Server* in Figure 4.5, the symbols *state*, *Que*, x, and y are variables whose values, by the initial interpretation, should be 0, empty sequence, void, and void, respectively. The symbols *register*, *unregister*, *involve*, and *return* are constants. enqueque and dequeue are 2- and 1-ary static functions. The variable *update* is a monitored variable whose value is not controlled by *Server*. In interpreting the symbols, to distinguish symbols with the same name but in different architectural elements, we prefix them with the names of architectural elements. Hence, we prefix '*state*' with '*Server*'.

In executing an update rule, values of some variables are changed, resulting in an update of the interpretation. For *CSsystem*, the updates are always triggered

by events of sending or receiving messages. As one may expect, events in the ASM update rules are interpreted as input or output actions in process algebra. The interpretation in each step of the update corresponds to a process identifier. In this way, the update rules define a set of recursive equations in process algebra. For example, the following are two possible recursive equations for *Server*: (1) if $Que = \epsilon$ and $update =$ 'ready', then

$$X_{[2,\epsilon]} \stackrel{\text{def}}{=} \sum_{a \in clt} \langle involve, a \rangle ?.\ X_{[3,a]} + \langle unregister, s \rangle !.\ X_{[3,\epsilon]}$$

(2) if $Que \neq \epsilon$, then

$$X_{[2,Que]} \stackrel{\text{def}}{=} \sum_{a \in clt} \langle involve, a \rangle ?.\ X_{[2,Que_a]} + \langle return, c \rangle !.\ X_{[2,Que']}$$

where $Que_a = \text{enqueue}(a, Que)$ such that $a \in clt$, $c = \text{head}(Que)$, and $Que' = \text{dequeue}(Que)$.

4.4 Architectural Modelling and Properties

In this section, we model the connector-based architecture and define its properties of interest in $\mathcal{PA}_2$. The main architecture-related concepts, including component types, architecture types, the component conformance relation, and component configurations are formulated in $\mathcal{PA}_2$. Several properties of the architecture is defined, including deadlock-freedom, non-starvation, conservation, and completeness. The property of dynamism is not defined directly, but captured by the uncertainty of the component configuration of the modelled architecture. The running example is used to illustrate those concepts and properties.

4.4.1 The Semantic Model

Firstly, we model components and connectors as processes with syntactic restrictions.

Definition 4.4.1 (Component, connector)**.** *Components $\mathcal{E}\langle a \rangle$, where a is the*

name of $\mathcal{E}$, are processes in $Proc''$. Connectors $\mathcal{G}$ are processes in $Proc'$.

Components and connectors are '$\|$'-free processes (i.e. processes in $Proc'$), because parallel compositions are only for inter-components compositions. For technical reasons (see Sect. 4.5.2), we further assume that components are '$\times$'-free (hence, components are processes in $Proc''$).[1] By Lemma 4.2.2, components and connectors in our semantic model do not perform internal actions. Hence, we do not need to absorb the internal transitions when comparing two traces.

Definition 4.4.2 (Component type)**.** *A component type is an abstract process*

$$\mathfrak{E} = X(a_0 : N_0, \ldots, a_m : N_m)$$

where (1) $X\langle a_1, \ldots, a_m\rangle \in Proc''$, (2) m is the number of formal parameters of $\mathfrak{E}$, and (3) a_1 is to be replaced by a component name, with N_1 being the name space of instantiating components of $\mathfrak{E}$. Usually, we denote a_0 and N_0 as $a_{\mathfrak{E}}$ and $N_{\mathfrak{E}}$, respectively.

Component types, together with a connector, define an architecture type. As indicated before, the semantic model is peculiar to the dynamic connector-based architecture types.

Definition 4.4.3 (Architecture type)**.** *A (dynamic, connector-based) architecture type is a collection of component types and a connector:*

$$\mathfrak{A} = \langle \mathfrak{E}_1^{\mathfrak{A}}, \ldots, \mathfrak{E}_n^{\mathfrak{A}}, \mathcal{G}^{\mathfrak{A}}\rangle$$

where $\mathcal{G}$ denotes the connector, and each $\mathfrak{E}_i$ is a component type.

We define a kind of components associated with component types, which are called canonical components.

Definition 4.4.4 (Canonical component)**.** *Let $\mathfrak{E}$ be defined as in Definition 4.4.2. If $a \in N_{\mathfrak{E}}$, we call*

$$\mathcal{E}_c^{\mathfrak{E}}\langle a\rangle = \sum_{a_1 \in A_1, \ldots a_m \in A_m} X\langle a, a_1, \ldots a_m\rangle$$

[1] In practice, the restrictions amount to the prohibition to use the operator of interleaving composition to formalise component behaviours.

a canonical component of $\mathfrak{E}$.

Canonical components are enormously important to the semantic model and analytic techniques developed later. The first utility of canonical components is to formulate the conformance relation between components and component types.

Definition 4.4.5 (Component conformance)**.** $\mathcal{E}\langle a\rangle$ *conforms to* $\mathfrak{E}$ *(or, equivalently speaking,* $\mathcal{E}\langle a\rangle$ *is an instance of* $\mathfrak{E}$*), denoted* $\mathfrak{E} \vdash \mathcal{E}\langle a\rangle$*, if (1)* $a \in N_{\mathfrak{E}}$*, and (2) there is a relation* $\mathcal{R} \subseteq Proc \times Proc$ *such that* $\langle \mathcal{E}_c^{\mathfrak{E}}\langle a\rangle, \mathcal{E}\langle a\rangle\rangle \in \mathcal{R}$ *and for each* $\langle P_1, P_2\rangle \in \mathcal{R}$*:*

- *if* $P_1 = \mathbf{0}$ *then* $\langle \mathcal{E}_c^{\mathfrak{E}}\langle a\rangle, P_2\rangle \in R$ *or* $P_2 = \mathbf{0}$*;*
- *if* $P_1 \xrightarrow{\alpha} P_1'$ *then* $P_2 \xrightarrow{\alpha} P_2'$ *and* $\langle P_1', P_2'\rangle \in R$ *for some* P_2'*;*
- *if* $P_2 \xrightarrow{\alpha} P_2'$ *then* $P_1 \xrightarrow{\alpha} P_1'$ *and* $\langle P_1', P_2'\rangle \in R$ *for some* P_1'*.*

If $\mathfrak{E} \vdash \mathcal{E}\langle a\rangle$, sometimes we write $\mathcal{E}^{\mathfrak{E}}$ to indicate the type of $\mathcal{E}$. Clearly, $\mathfrak{E} \vdash \mathcal{E}_c^{\mathfrak{E}}\langle a\rangle$. Intuitively, this means that a canonical component of a component type is an instance of that component type indeed.

While component types define the common patterns or constraints of the instantiating components, the dynamism of architecture types refers to the uncertainty of actual component configurations, which are defined as the interleaving compositions of components.

Definition 4.4.6 (Component configuration)**.** *A component configuration of* $\mathfrak{A}$ *is a process of the form*

$$\mathcal{F}^{\mathfrak{A}} = \mathcal{E}_1\langle a_1\rangle \times \ldots \times \mathcal{E}_n\langle a_n\rangle$$

such that, for each $1 \leq i \neq j \leq n$*,* $a_i \neq a_j$ *and* $\mathfrak{E}_i^{\mathfrak{A}} \vdash \mathcal{E}_i\langle a\rangle$ *for some* $\mathcal{E}_i^{\mathfrak{A}}$*.*

We stress the fact that the order of components in a component configuration (i.e. the order of processes in the interleaving composition) is of no importance (with respect to the techniques presented later in this chapter).

An architecture instance is the synchronised composition of a component configuration and a connector.

Definition 4.4.7 (Architecture instance)**.** *An architecture instance of* $\mathfrak{A}$ *is a process of of the form* $\mathcal{A}^{\mathfrak{A}} = \mathcal{F}^{\mathfrak{A}} \parallel \mathcal{G}^{\mathfrak{A}}$.

Throughout this chapter, if the underlying architecture type $\mathfrak{A}$ is clear in the context, we write $\mathcal{G}$, $\mathfrak{E}$, and $\mathcal{F}$ instead of $\mathcal{G}^{\mathfrak{A}}$, $\mathfrak{E}^{\mathfrak{A}}$, and $\mathcal{F}^{\mathfrak{A}}$, respectively.

4.4.2 Architectural Properties

Based on the formal modelling of the architecture, we are able to formulate the aforementioned architectural properties. The first one is deadlock-freedom.

Definition 4.4.8 (Deadlock-freedom)**.** $\mathcal{A} = \mathcal{F} \parallel \mathcal{G}$ *is deadlock-free, if the following proposition holds: if* $\mathcal{A} \Longrightarrow P \parallel Q$ *and* $P \longrightarrow$*, then* $P \parallel Q \longrightarrow$. $\mathfrak{A}$ *is deadlock-free if each instance* $\mathcal{A}$ *of* $\mathfrak{A}$ *is deadlock-free.*

If some of the components in a deadlock-free architecture instance can still proceed, then the architecture instance can proceed. The deadlock-freedom of an architecture type is defined in terms of the deadlock-freedom of all of its architecture instances. Hence, the verification of deadlock-freedom of an architecture depends on all of its architecture instances.

It should be noted that deadlock-freedom formulated in Definition 4.4.8 is different the one formulated in Definition 3.5.1 on Page 24. Informally, Definition 4.4.8 is weaker than Definition 3.5.1, because the condition 'some of the components in a deadlock-free architecture instance can still proceed' is not needed in the latter. Definition 4.4.8 emphasises the fact that components rather than connectors are the computational loci of a complex system, but both definitions are good candidates for capturing the intuitive notion of deadlock-freedom in architectural analysis.

A stronger property than deadlock-freedom is non-starvation, which says that, at each point during the runtime, non-terminated components can interact with the connector in future.

Definition 4.4.9 (Non-starvation)**.** $\mathcal{A} = (\mathcal{F} \times \mathcal{E}\langle a \rangle) \parallel \mathcal{G}$ *is* non-starving,[1] *if the following proposition holds: if* $\mathcal{A} \Longrightarrow (P \times Q) \parallel R$ *and* $Q \longrightarrow$*, then there are* P'

[1] Here and in the sequel, by writing $\mathcal{A}(\mathcal{F} \times \mathcal{E}\langle a \rangle) \parallel \mathcal{G}$, it is assumed that $\mathcal{F} \times \mathcal{E}\langle a \rangle$ is a configuration (see Definition 4.4.6).

and R' such that $P \parallel R \Longrightarrow P' \parallel R'$ and $Q \parallel R' \longrightarrow$. $\mathfrak{A}$ is non-starving if each instance $\mathcal{A}$ of $\mathfrak{A}$ is non-starving.

Lemma 4.4.10. *If an architecture instance is non-starving then it is deadlock-free.*

Definition 4.4.11 (Conservation)**.** *$\mathfrak{A}$ is conservative, if, for each $\widetilde{\alpha}$ such that $\mathcal{G} \xrightarrow{\widetilde{\alpha}}$, there is a configuration $\mathcal{F}$ such that $\mathcal{F} \xrightarrow{\sharp\widetilde{\alpha}}$.*

Conservation is a property peculiar to connector-based architecture types. An architecture type is conservative if each trace of the connector corresponds to a trace of some component configuration. In other words, the behaviours of all architecture instances are refined by the connectors. The practical significance of conservation is that, if we want to know whether the system would behave abnormally, it is sufficient to check the connector only.

Definition 4.4.12 (Completeness)**.** *$\mathfrak{A}$ is complete, if, for each component $\mathcal{E}\langle a\rangle$ of $\mathfrak{A}$ and $P \in Proc(\mathcal{E}\langle a\rangle)$, there is $\mathcal{A} = (\mathcal{F} \times \mathcal{E}\langle a\rangle) \parallel \mathcal{G}$ such that $\mathcal{A} \Longrightarrow (Q \times P) \parallel R$ for some Q, R.*

The following lemma gives an alternative characterization of completeness.

Lemma 4.4.13. *$\mathfrak{A}$ is complete if and only if the following proposition holds: for each component $\mathcal{E}\langle a\rangle$ of $\mathfrak{A}$ and $P \in Proc(\mathcal{E}\langle a\rangle)$, if $P \xrightarrow{\alpha}$, then there is $\mathcal{A} = (\mathcal{F} \times \mathcal{E}\langle a\rangle) \parallel \mathcal{G}$ such that $\mathcal{A} \Longrightarrow (Q \times P) \parallel R$ and $R \xrightarrow{\sharp\alpha}$ for some Q, R.*

If an architecture type is complete, then any behaviour prescribed by the component specification is realisable by the connector, given a suitable component configuration. If a complete architecture type is connector-based, then the connector does not exclude the behaviours of components. The practical significance of completeness is that, if we want to know whether the system realises some expected behaviour, it is sufficient to check the connector only. In this sense, completeness is dual to conservation.

We define a useful group of architecture types, in which all canonical components are deterministic.

Definition 4.4.14. *An architecture type $\mathfrak{A}$ is* deterministic *if its connector and all of its canonical components are deterministic.*

4.4.3 Running Example Revisited

We apply the formulations of architectural concepts and properties to our running example. An architecture instance of $CStype = \langle CLIENT, SERVER, CSCON \rangle$ has the semantics

$$(Client_1\langle c_1 \rangle \times Client_2\langle c_2 \rangle \times Server\langle c \rangle) \parallel CSCON$$

$CLIENT(c : clt, s : sev)$ and $SERVER(s : sev)$ are the client and server component types, respectively. Recall that clt and sev are the name spaces of clients and servers. We have that:

$$CLIENT(c : clt, s : sev) \vdash Client_1\langle c_1 : clt \rangle$$

$$CLIENT(c : clt, s : sev) \vdash Client_2\langle c_2 : clt \rangle$$

$$SERVER(s : sev) \vdash Server\langle s_1 : sev \rangle$$

Therefore, the definition of conformance relation handles those variances between $Client_1\langle c_1 : clt \rangle$, $Client_2\langle c_2 : clt \rangle$ and $CLIENT(c : clt, s : sev)$ and between $Server\langle s_1 : sev \rangle$ and $SERVER(s : sev)$ explained in Section 4.3.

Also, since $CLIENT(c : clt, s : sev)$ and $SERVER(s : sev)$ are deterministic, by definition, $CSTYPE_d$ is deterministic. Moreover, $CSTYPE_d$ enjoys the properties of deadlock-freedom, non-starvation, and completeness. However, it is not conservative. For example,

$$\langle request, c_1, s_1 \rangle ?, \langle request, c_1, s_1 \rangle ?, \ldots$$

is a trace of $CSCON$, but

$$\langle request, c_1, s_1 \rangle !, \langle request, c_1, s_1 \rangle !, \ldots$$

is not a trace of $CLIENT(c : clt, s : sev)$.

4.5 Formal Analysis

4.5.1 Conformance

The conformance relation can be checked by some algorithm.

Theorem 4.5.1. *There exists an algorithm for deciding whether* $\mathfrak{E} \vdash \mathcal{E}\langle a\rangle$ *for any* $\mathfrak{E}$ *and* $\mathcal{E}\langle a\rangle$.

An algorithm similar to the bisimulation checking suffices to decide this problem. However, a complete description of such an algorithm is based on some tedious auxiliary definitions. For presentation purposes, here we only illustrate the main steps of such an algorithm. To check whether $\mathfrak{E} \vdash \mathcal{E}\langle a\rangle$, first, we declare a dynamic set $\mathcal{R} \subseteq Proc(\mathcal{E}_c^{\mathfrak{E}}\langle a\rangle) \times Proc(\mathcal{E}\langle a\rangle)$ and let $\mathcal{R} = \{\langle \mathcal{E}_c^{\mathfrak{E}}\langle a\rangle, \mathcal{E}\langle a\rangle\rangle\}$. Then, we use a depth-first search to visit a subset of $Proc(\mathcal{E}_c^{\mathfrak{E}}\langle a\rangle) \times Proc(\mathcal{E}\langle a\rangle)$ and add elements into $\mathcal{R}$ in a manner such that for each $\langle P_1, P_2\rangle \in \mathcal{R}$ satisfying the three conditions in Definition 4.4.5, namely:

- if $P_1 = \mathbf{0}$ then $\langle \mathcal{E}_c^{\mathfrak{E}}\langle a\rangle, P_2\rangle \in \mathcal{R}$ or $P_2 = \mathbf{0}$;
- if $P_1 \xrightarrow{\alpha} P_1'$ then $P_2 \xrightarrow{\alpha} P_2'$ and $\langle P_1', P_2'\rangle \in R$ for some P_2';
- if $P_2 \xrightarrow{\alpha} P_2'$ then $P_1 \xrightarrow{\alpha} P_1'$ and $\langle P_1', P_2'\rangle \in R$ for some P_1'.

If such $\mathcal{R}$ exists, then $\mathfrak{E} \vdash \mathcal{E}\langle a\rangle$.

The following theorem presents some properties of the conformance relation.

Theorem 4.5.2. *1.* $\mathfrak{E} \vdash \mathcal{E}_c^{\mathfrak{E}}\langle a\rangle$ *for each* $a \in N_{\mathfrak{E}}$;

2. if $\mathfrak{E} \vdash \mathcal{E}\langle a\rangle$ *and* $\mathfrak{E} \vdash \mathcal{E}'\langle a\rangle$ *then* $\mathfrak{E} \vdash \mathcal{E}\langle a\rangle + \mathcal{E}'\langle a\rangle$;

3. if $\mathfrak{E} \vdash \mathcal{E}\langle a\rangle$ *then* $\mathfrak{E} \vdash \mathcal{E}\langle a\rangle^*$ *(where* $\mathcal{E}\langle a\rangle \in Proc''$ *and* $\mathcal{E}\langle a\rangle^*$ *is the iteration of* $\mathcal{E}\langle a\rangle$ *and formally defined in the next subsection);*

4. if $\mathfrak{E} \vdash \mathcal{E}\langle a\rangle$ *and* $\mathcal{E}\langle a\rangle \backsimeq \mathcal{E}'\langle a\rangle$ *then* $\mathfrak{E} \vdash \mathcal{E}'\langle a\rangle$.

Algorithm 1: Deadlock-freedom checking for $\mathfrak{A}$

Data: $\mathcal{A}_c^*$
Output: 'yes' or 'no'

let $bool = 1$
foreach $(P_1 \times \ldots \times P_k) \parallel Q \in Proc(\mathcal{A}_c^*)$ **do**
  if $P_1 \times \ldots \times P_k = \mathcal{F}_c^*$ **then**
    if $X_{P_{c,i}} \parallel Q \not\longrightarrow, \exists 1 \leq i \leq k$ **then**
      $bool := 0$
      break
  else
    let $P_i' = P_i\{\mathbf{0}/P_{c,i}\}, \forall\, 1 \leq i \leq k$
    if $(P_1' \times \ldots \times P_k') \parallel Q' \not\longrightarrow$ **then**
      $bool := 0$
      break
if $bool = 1$ **then return** 'yes'
else return 'no'

4.5.2 Deadlock-freedom and Non-starvation

To check deadlock-freedom and non-starvation, we construct an architecture instance that can 'mimic' (in some sense) behaviours of all architecture instances. By so doing, we reduce the searching space from the entire universe of architecture instances into a single architecture instance. These checking techniques apply to deterministic and non-deterministic architecture types.

The specific architecture instance is constructed by the following procedure. For any given $P \in Proc''$, we choose a new process identifier X_P. The *iteration* of P, denoted by P^*, is obtained from P by substituting X_P with $\mathbf{0}$ in P, and the recursive equation for X_P is $X_P \stackrel{\text{def}}{=} P^*$. By Theorem 4.5.2, the iteration of $\mathcal{E}_c^{\mathfrak{E}}\langle a\rangle$, i.e. $\mathcal{E}_c^{\mathfrak{E}}\langle a\rangle^*$, is a component.

An important property of canonical components is captured by the following lemma.

Lemma 4.5.3. *(1) If* $\mathcal{E}^{\mathfrak{E}}\langle a\rangle \xrightarrow{\widetilde{\alpha}}$, *then* $\mathcal{E}_c^{\mathfrak{E}}\langle a\rangle^* \xrightarrow{\widetilde{\alpha}}$. *(2) If* $\mathcal{E}_c^{\mathfrak{E}}\langle a\rangle^* \xrightarrow{\widetilde{\alpha}}$, *then there is* $\mathcal{E}^{\mathfrak{E}}\langle a\rangle$ *such that* $\mathcal{E}^{\mathfrak{E}}\langle a\rangle \xrightarrow{\widetilde{\alpha}}$.

Lemma 4.5.3 demonstrates that any trace of a component is a trace of the

Algorithm 2: Non-starvation checking for $\mathfrak{A}$

Data: $\mathcal{A}_c^*$
Output: 'yes' or 'no'

let $bool = 1$
foreach $(P_1 \times \ldots \times P_k) \parallel Q \in Proc(\mathcal{A}_c^*)$ **do**
  let $P_i' = P_i\{\mathbf{0}/X_{P_{c,i}}\}$, $\forall\, 1 \leq i \leq k$
  foreach $1 \leq i \leq k$ **do**
    let $R_i = P_1' \times \ldots P_{i-1}' \times P_{i+1}' \times \ldots P_k'$
    if *there are* R_i' *and* Q' *such that*
      1. $R_i \parallel Q \Longrightarrow R_i' \parallel Q'$
      2. $P_i \parallel Q' \longrightarrow$
    then
      skip
    else
      $bool := 0$
      break
if $bool = 1$ **then return** 'yes'
else return 'no'

iteration of the canonical component of the same type, and that any trace of the iteration of a canonical component is a trace of some component of the same type. The analytic techniques developed later make use of this lemma.

From now on, we use $P_c^{\mathfrak{A}}$ to denote a canonical component (of some component type) in $\mathfrak{A}$ and assume that $P_{c,1}^{\mathfrak{A}}, \ldots, P_{c,k}^{\mathfrak{A}}$ enumerate all canonical components of $\mathfrak{A}$. Recall that components belong to $Proc''$, which is one of the assumptions in our semantic model (see Section 4.4.1). Let

$$\mathcal{F}_c^{\mathfrak{A}} = P_{c,1}^{\mathfrak{A}} \times \ldots \times P_{c,k}^{\mathfrak{A}} \qquad \mathcal{A}_c^{\mathfrak{A}} = \mathcal{F}_c^{\mathfrak{A}} \parallel \mathcal{G}^{\mathfrak{A}}$$
$$\mathcal{F}_c^{\mathfrak{A}*} = X_{P_{c,1}^{\mathfrak{A}}} \times \ldots \times X_{P_{c,k}^{\mathfrak{A}}} \qquad \mathcal{A}_c^{\mathfrak{A}*} = \mathcal{F}_c^{\mathfrak{A}*} \parallel \mathcal{G}^{\mathfrak{A}}$$

For readability, we just write $P_{c,i}$, $\mathcal{F}_c$, $\mathcal{F}_c^*$, and $\mathcal{A}_c^*$, assuming a single underlying architecture type $\mathfrak{A}$.

Since for each $X_{P_{c,i}}$ there exists $\mathfrak{E}^{\mathfrak{A}}$ such that $\mathfrak{E}^{\mathfrak{A}} \vdash X_{P_{c,i}}$ (see Theorem 4.5.2), $\mathcal{A}_c^*$ is an architecture instance of $\mathfrak{A}$. Clearly, if $\mathfrak{A}$ is deadlock-free (resp. non-starving) then $\mathcal{A}_c^*$ is deadlock-free (resp. non-starving). The opposite direction

of the above proposition is, unfortunately, wrong. Hence, we reduce deadlock-freedom and non-starvation of $\mathfrak{A}$ into two properties that are slightly stronger than the deadlock-freedom and non-starvation of $\mathcal{A}_c^*$. The main steps of algorithms for checking two properties are presented in Algorithms 1 and 2, both of which are based on depth-first searches. The following theorems confirm the correctness of the two algorithms.

Theorem 4.5.4. *$\mathfrak{A}$ is deadlock-free if and only if Algorithm 1 returns 'yes'.*

Theorem 4.5.5. *$\mathfrak{A}$ is non-starving if and only if Algorithm 2 returns 'yes'.*

An important feature of both algorithms is that they search processes in $Proc(\mathcal{A}_c^*)$ but verify properties of $\mathfrak{A}$. The state space of $\mathcal{A}_c^*$ is strictly smaller than that of $\mathfrak{A}$ except for a few extreme cases.

4.5.3 Conservation and Completeness (1)

Before we present the techniques to handle conservation of architecture types, we put forward an auxiliary definition. Let $P \ll Q$ if for any $\tilde{\alpha}$ if $P \xrightarrow{\tilde{\alpha}}$ implies $Q \xrightarrow{\sharp\tilde{\alpha}}$. Intuitively, $\ll$ formulates a 'dual' version of trace inclusion between two processes. Because $Proc(P)$ is finite for any P (see Lemma 4.2.1), we have the following lemma.

Theorem 4.5.6. *There is an algorithm to decide whether $P \ll Q$ for each P, Q.*

Theorem 4.5.7. *$\mathfrak{A}$ is conservative if and only if $\mathcal{G} \ll \mathcal{A}_c^*$.*

Now we deal with completeness. First, we augment the syntax of $\mathcal{PA}_2$ with a new connective $\lfloor$ and its relevant semantic rule is as follows:

$$\frac{P_1 \parallel P_2 \Longrightarrow R_1 \parallel R_2 \quad R_1 \xrightarrow{\alpha} R_1' \quad R_1' \parallel R_2 \Longrightarrow Q_1 \parallel Q_2}{P_1 \lfloor P_2 \xrightarrow{\alpha} Q_1 \lfloor Q_2}$$

In the new PA notation, we still have Lemma 4.5.6. We use $\mathcal{F}_c^*[-i]$ to denote the canonical configuration of $\mathfrak{A}$ that 'misses' $P_{c,i}$, i.e.

$$\mathcal{F}_c^*[-i] = P_{c,1} \times \ldots P_{c,i-1} \times P_{c,i+1} \times \ldots P_{c,k} \ .$$

Theorem 4.5.8. $\mathfrak{A}$ *is complete if and only if* $P_{c,i} \ll \mathcal{G}\lfloor\mathcal{F}_c^*[-i]$ *for all* $1 \leq i \leq k$.

A point worthy of notice is that Theorem 4.5.8 implies a compositional method (algorithm) to check the completeness of $\mathfrak{A}$, in the sense that the method checks canonical components one by one. Hence, such a method can be decomposed into a sequence of sub-algorithms, each of which checks some $P_{c,i}$ against $\mathcal{A}_c^*$. In the next subsection, we further develop these compositional methods.

4.5.4 Conservation and Completeness (2)

Although algorithms for checking the conservation and completeness of deterministic architecture types in general have been presented, we further develop verification techniques for these two properties. Techniques in the subsections are based on the deterministic assumption of architecture types.

We have explained the compositional nature of Theorem 4.5.8: it can be decomposed into sub-algorithms checking each canonical component $P_{c,i}$ against $\mathcal{A}_c^*$. For a connector-based architectural instance in which components can only communicate with the central connectors, one would expect to check $P_{c,i}$ against $\mathcal{G}$ only. The advantage of such connector-based compositional checking is the facilitation of identifying the problematic parts of a system [Su et al., 2010]. For example, if $P_{c,i}$ and $\mathcal{A}_c$ are the input of this algorithm and a negative answer is returned, then the problem could be that $P_{c,i}$ has unintended behaviours or that $\mathcal{G}$ fails to realise some behaviour of $P_{c,i}$.

The connector-based compositional checking needs to further assume the following two properties:

- $Act(P_{c,i}) \cap Act(P_{c,j}) = \varnothing$ if $i \neq j$
- $\bigcup_{i=1}^{k} Act(P_{c,i}) = \{\sharp\alpha \mid \alpha \in Act(\mathcal{G})\}$

The first property says that actions are unique to canonical components. The second says that actions of components are dual to actions of the connector, and vice versa. The assumption of these two properties should not be an excessive restriction and their verification is usually in accordance with the structural aspect of architectural descriptions. As an example, $CSTYPE_d$ in the running example satisfies both properties.

Let $\widetilde{\alpha}|A$ be the projection of $\widetilde{\alpha}$ on $A \subseteq Act$. $P \xrightarrow{\widetilde{\gamma}}_A Q$ if and only if there is $\widetilde{\alpha}$ such that $P \xrightarrow{\widetilde{\alpha}} Q$ and $\widetilde{\gamma} = \widetilde{\alpha}|A$. Note that $\longrightarrow$ is just $\longrightarrow_{Act}$.

Definition 4.5.9. *(1)* $\mathcal{G} \preccurlyeq P_c$ *if* $\mathcal{G} \xrightarrow{\widetilde{\alpha}}_{Act(P_c)}$ *implies* $P_c^* \xrightarrow{\sharp\widetilde{\alpha}}$*; (2)* $\mathcal{G} \succcurlyeq P_c$ *if* $P_c^* \xrightarrow{\widetilde{\alpha}}$ *implies* $\mathcal{G} \xrightarrow{\sharp\widetilde{\alpha}}_{Act(P_c)}$.

The following theorem links $\mathcal{G} \preccurlyeq P_c$ and the conservative property of architecture types.

Theorem 4.5.10. $\mathfrak{A}$ *is conservative if and only if* $\mathcal{G} \preccurlyeq P_c$ *for each* P_c *(in* $\mathfrak{A}$*.)*

Besides facilitating the identification of potential design flaws in the system, the connector-based compositional checking also simplifies the verification, because it does not generate the whole set of processes in $Proc(\mathcal{A}_c^*)$.

For the property of completeness, a similar composition checking is, however, possible only based on the conservation of $\mathfrak{A}$.

Theorem 4.5.11. *Suppose* $\mathfrak{A}$ *is conservative.* $\mathfrak{A}$ *is complete if and only if* $\mathcal{G} \succcurlyeq P_c$ *for* P_c *(in* $\mathfrak{A}$*).*

Compared with Theorem 4.5.8, Theorem 4.5.11 presents a finer-grained compositional checking techniques for completeness, which is between the connector and a canonical component only. Hence, it is more helpful for identifying the design flaws. Also, similar to the case of conversation checking, it simplifies the verification by exempting the generation of all processes in $Proc(\mathcal{A}_c^*)$.

4.6 Proof Details

In this section, the proofs of the tricky lemmas and theorems are given.

Proof of Lemma 4.2.1. The finiteness of action labels are clear, thus we only show the finiteness of process states.

For each $M \in Proc''$ (a process without $\|$ or $\times$) and let $sub(M)$ be the set of *sub-processes* (the definition of which is standard) of M. We use M_I to denote the process such that $X \stackrel{\text{def}}{=} M_I$. Let D be the (finite) set of process identifiers and

$$S = \bigcup_{X \in D} sub(M_x) \cup D \subseteq Proc''$$

S is obviously finite. Further, it can be shown that $Proc(X) \subseteq S$ for each $X \in D$: First, $X \in S$. Secondly, suppose $Q \in Proc(X)$ and $Q \xrightarrow{\alpha} Q'$. By induction hypothesis, $Q \in S$. Because $Q \in Proc''$, by the syntax of Q, Q is a process identifier or a process of the form $\alpha_1.Q_1 + \ldots + \alpha_n.Q_n$. In either case, we have that $Q' \in S$.

We now show $Proc(P)$ is finite, for any $P \in Proc$. First, $Proc(\mathbf{0}) = \varnothing$, and $Proc(X) \subseteq S$ as shown above. Also, by induction hypothesis, we can show the sets below are finite:

- $Proc(P_1 + P_2) = Proc(P_1) \cup Proc(P_2) \cup \{P_1 + P_2\}$
- $Proc(P_1 \times P_2) \subseteq \{P_1' \times P_2' \mid P_1' \in Proc(P_1), P_2' \in Proc(P_2)\}$
- $Proc(P_1 \parallel P_2) \subseteq \{P_1' \parallel P_2' \mid P_1' \in Proc(P_1), P_2' \in Proc(P_2)\}$

This completes the proof. □

Proof of Lemma 4.2.2. The internal action τ will only generated by Rule [COM-2], however, this rule is not applicable in any $P \in Proc''$ as P is $\parallel$-free. □

Proof of Theorem 4.5.1. A proof of this theorem is sketched immediate after the presentation of the theorem. The component conformance problem can be solved by a variant of bisimulation checking algorithm. □

Proof of Lemma 4.4.13. We call the following proposition of $\mathfrak{A}$ the *alt-completeness*: for each component $\mathcal{E}\langle a\rangle$ of $\mathfrak{A}$ and $P \in Proc(\mathcal{E}\langle a\rangle)$, if $P \xrightarrow{\alpha}$, then there is $\mathcal{A} = (\mathcal{F} \times \mathcal{E}\langle a\rangle) \parallel \mathcal{G}$ such that $\mathcal{A} \Longrightarrow (Q \times P) \parallel R$ and $R \xrightarrow{\sharp\alpha}$ for some Q, R. (1) We first show that the alter-completeness of $\mathfrak{A}$ entails the completeness of $\mathfrak{A}$. Let $P \in Proc(\mathcal{E}\langle a\rangle)$, i.e. $\mathcal{E}\langle a\rangle \Longrightarrow P$. If $P = \mathcal{E}\langle a\rangle$ then we are done. Otherwise, let $\mathcal{E}\langle a\rangle \Longrightarrow P' \xrightarrow{\alpha} P$. By alt-completeness, there is $\mathcal{F}$ such that $(\mathcal{F} \times \mathcal{E}\langle a\rangle) \parallel \mathcal{G} \Longrightarrow (Q \times P') \parallel R$ for some Q, R such that $R \xrightarrow{\sharp\alpha} R'$ for some R'. Thus, $(\mathcal{F} \times \mathcal{E}\langle a\rangle) \parallel \mathcal{G} \Longrightarrow (Q \times P) \parallel R'$.

(2) Suppose $\mathfrak{A}$ is complete, and let $P \in Proc(\mathcal{E}\langle a\rangle)$ for some $\mathcal{E}\langle a\rangle$ and $P \xrightarrow{\alpha} P'$. We construct a new process P^{new} in the following way: let $P^{\text{new}} = \mathcal{E}\langle a\rangle\{P^{\text{new}\prime}/P\}$ where $P^{\text{new}\prime} = \alpha.X^{\text{new}} + P$ and $X^{\text{new}} \stackrel{\text{def}}{=} P'$. We can show that $P^{\text{new}} \simeq P$ and thus P^{new} is component of $\mathfrak{A}$. By completeness of $\mathfrak{A}$,

$(\mathcal{F} \times P^{\text{new}}) \parallel \mathcal{G} \Longrightarrow (Q \times X^{\text{new}}) \parallel R$. Thus, $(\mathcal{F} \times P^{\text{new}}) \parallel \mathcal{G} \Longrightarrow (Q' \times P^{\text{new}\prime}) \parallel R' \xrightarrow{\tau} (Q \times X^{\text{new}}) \parallel R$ and $R' \xrightarrow{\sharp\alpha} R$ for some Q', R'. Thus, $(\mathcal{F} \times \mathcal{E}) \parallel R' \Longrightarrow (Q' \times P') \parallel R' \xrightarrow{\tau} (Q \times P) \parallel R$ and $R' \xrightarrow{\sharp\alpha} R$. This completes the proof. □

The proof of Theorem 4.5.1 is sketched in the text. The proofs of Theorem 4.5.2 and Lemma 4.5.3 immediately follow the definitions of the component type $\mathfrak{E}\langle a \rangle$ (Definition 4.4.2), the component conformance relation $\vdash$ (Definition 4.4.5), and the iteration of processes P^* (Page 54).

Theorem 4.5.4 can be rephrased as follows:

Theorem 4.6.1. *$\mathfrak{A}$ is deadlock-free if and only if the following two conditions hold for $\mathcal{A}_c^*$ (where $\mathcal{A}_c$ is the canonical architecture instance for $\mathfrak{A}$): for each $(P_1 \times \ldots \times P_k) \parallel Q \in Proc(\mathcal{A}_c^*)$,*

1. *if $(P_1 \times \ldots \times P_k) = \mathcal{F}_c^*$, then $I_{P_{c,i}} \parallel C' \longrightarrow$ for each $1 \leq i \leq k$,*
2. *otherwise, $(P_1' \times \ldots \times P_k') \parallel Q \longrightarrow$ where $P_i' = P_i\{\mathbf{0}/X_{P_{c,i}}\}$ for each $1 \leq i \leq k$.*

Proof. (1) Suppose $\mathfrak{A}$ is deadlock-free. Let $(P_1 \times \ldots \times P_k) \parallel Q \in Proc(\mathcal{A}_c^*)$. (1a) If $P_1 \times \ldots \times P_k = \mathcal{F}_c^*$, there are $\mathcal{F}, R$ such that

$$R = \underbrace{\mathbf{0} \times \ldots \mathbf{0}}_{i-1} \times I_{P_{c,i}} \times \underbrace{\mathbf{0} \times \ldots \mathbf{0}}_{k-i} \ ,$$

and $R \parallel Q \in Proc(\mathcal{F} \parallel \mathcal{G})$. Hence, $X_{P_{c,i}} \parallel C' \longrightarrow$. (1b) If $P_1 \times \ldots \times P_k \neq \mathcal{F}_c^*$, there are $\mathcal{F}', R'$ such that

$$R' = P_i\{\mathbf{0}/X_{P_{c,i}}\} \times \ldots \times P_i\{\mathbf{0}/I_{P_{c,i}}\} \ ,$$

and $R' \parallel Q \in Proc(\mathcal{F}' \parallel \mathcal{G})$. Hence, $R' \parallel Q \longrightarrow$.

(2) Suppose $\mathfrak{A}$ is *not* deadlock-free, i.e. there is $\mathcal{F}$ such that $R \parallel Q \in Proc(\mathcal{F} \parallel \mathcal{G})$, $R \longrightarrow$, and $R \parallel Q \not\longrightarrow$ for some R. We show that at least one of the two conditions above is violated. Let $\mathcal{F} \xrightarrow{\tilde{\alpha}} R$ and $\mathcal{G} \xrightarrow{\sharp\tilde{\alpha}} Q$. We claim that there is $R' = P_1 \times \ldots \times P_k$ such that $F_c^* \xrightarrow{\tilde{\alpha}} R'$, and for each β:

- if $R \xrightarrow{\beta}$ then $R' \xrightarrow{\beta}$, and

- if $P'_1 \times \ldots \times P'_k \xrightarrow{\beta}$ where $P'_i = P_i\{\mathbf{0}/X_{P_{c,i}}\}$ then $R \xrightarrow{\beta}$.

This claim can be shown by induction on $\tilde{\alpha}$. By this claim, if $R' = \mathcal{F}_c^*$ the first condition is violated; otherwise, the second condition fails. □

Theorem 4.5.5 can be rephrased as follows:

Theorem 4.6.2. *$\mathfrak{A}$ is non-starving if and only if the following condition holds for $\mathcal{A}_c^*$ (where $\mathcal{A}_c$ is the canonical architecture instance for $\mathfrak{A}$): let $(P_1 \times \ldots \times P_k) \parallel Q \in Proc(\mathcal{A}_c^*)$ and*

$$R_i = P_1\{\mathbf{0}/X_{P_{c,1}}\} \times \ldots P_{i-1}\{\mathbf{0}/X_{P_{c,i-1}}\} \times P_{i+1}\{\mathbf{0}/X_{P_{c,i+1}}\} \times \ldots P_k\{\mathbf{0}/X_{P_{c,k}}\} \ ,$$

where $1 \leq i \leq k$; there are R'_i, Q' such that $R_i \parallel Q \Longrightarrow R'_i \parallel Q'$ and $P_i \parallel Q' \longrightarrow$.

Proof. (1) Suppose $\mathfrak{A}$ is non-starving. There is configurtation $\mathcal{F} \times \mathcal{E}\langle a \rangle$ (of $\mathfrak{A}$) and $(R_i \times P_i) \parallel Q \in Proc((\mathcal{F} \times \mathcal{E}\langle a \rangle) \parallel \mathcal{G})$. Thus, there are R'_i, Q' such that $R_i \parallel Q \Longrightarrow R'_i \parallel Q'$ and $P_i \parallel Q' \longrightarrow$.

(2) Suppose $\mathfrak{A}$ is *not* non-starving, i.e. there is $\mathcal{A} = (\mathcal{F} \times \mathcal{E}\langle a \rangle) \parallel \mathcal{G}$ such that $(R \times P) \parallel Q \in Proc(\mathcal{A})$, $P \longrightarrow$, and there are no R', Q' such that $R \parallel Q \Longrightarrow R' \parallel Q'$ and $P \parallel Q' \longrightarrow$. Let $\mathcal{F} \times \mathcal{E}\langle a \rangle \xrightarrow{\tilde{\alpha}} R \times P$ and $\mathcal{G} \xrightarrow{\sharp\tilde{\alpha}} Q$. Then, we claim that there are $P_1, \ldots, P_k$ such that

1. $\mathcal{F}_c^* = P_{c,1} \times \ldots \times P_{c,k} \xrightarrow{\tilde{\alpha}} P_1 \times \ldots \times P_k$, and
2. there is i $(1 \leq i \leq k)$ such that
   (a) if $P'_1 \times \ldots P'_{i-1} \times P'_{i+1} \times \ldots P'_k \xrightarrow{\tilde{\beta}}$ where $P'_j = P_j\{\mathbf{0}/X_{P_{c,l_j}}\}$ for each $1 \leq j \leq i-1$ and $i+1 \leq j \leq k$ then $R \xrightarrow{\tilde{\beta}}$,
   (b) $P \xrightarrow{\gamma}$ if and only if $P_i \xrightarrow{\gamma}$.

This claim can be shown by induction on $\tilde{\alpha}$. It follows that there are no R', Q' such that $(P'_1 \times \ldots P'_{i-1} \times P'_{i+1} \times \ldots P'_k) \parallel Q \Longrightarrow R' \parallel Q'$ and $P_i \parallel Q' \longrightarrow$. □

Proof of Theorem 4.5.6. Let $\Phi \subseteq Proc(P) \times 2^{Proc(Q)}$ satisfies the following condition: (1) $\langle P, \{Q\} \rangle \in \Phi$, and (2) if $\langle R, S \rangle \in \Phi$, then

- there is $\tilde{\alpha}$ such that $P \xrightarrow{\tilde{\alpha}} R$ and $S = \{Q' \mid Q \xrightarrow{\sharp\tilde{\alpha}} Q'\}$, and

- if $R \xrightarrow{\beta} R'$ and then there is S' such that $\langle R', S' \rangle \in \Phi$ and $S' = \{Q'' \mid Q' \xrightarrow{\sharp\beta} Q'', Q' \in S\}$.

Because $Proc(P) \times 2^{Proc(Q)}$ is a finite set (by Lemma 4.2.1), thus Φ is decidable. Then, the following checking is performed on elements of Φ: if there are γ and $\langle R, S \rangle \in \Phi$ such that $P \xrightarrow{\gamma}$ but $\{Q'' \mid Q' \xrightarrow{\sharp\gamma} Q'', Q' \in S\} = \varnothing$, then $P \ll Q$ is false; otherwise, $P \ll Q$ is true. □

Proof of Theorem 4.5.7. (1) Suppose $\mathfrak{A}$ is conservative. Let $\mathcal{G} \xrightarrow{\tilde{\alpha}}$. Then, there is a configuration $\mathcal{F}$ (of $\mathfrak{A}$) such that $\mathcal{F} \xrightarrow{\sharp\tilde{\alpha}}$. Thus, $\mathcal{F}_c^* \xrightarrow{\sharp\tilde{\alpha}}$. (2) The other direction is immediate as $\mathcal{F}_c^*$ is a configuration of $\mathfrak{A}$. □

Proof of Theorem 4.5.8. (1) Suppose $\mathfrak{A}$ is complete and $P_{c,i} \xrightarrow{\tilde{\alpha}}$ for some $i, \tilde{\alpha}$. Let $\mathcal{E}\langle a \rangle$ be a component (in $\mathfrak{A}$) such that

- $\mathcal{E}\langle a \rangle \xrightarrow{\tilde{\alpha}} P$ (see Lemma 4.5.3) and
- if $\mathcal{E}\langle a \rangle \xrightarrow{\tilde{\alpha}} P'$ then $P' = P$.

There is a configuration $\mathcal{F}$ such that $(\mathcal{F} \times \mathcal{E}\langle a \rangle) \parallel \mathcal{G} \Longrightarrow (R \times P) \parallel Q$. In other words, $\mathcal{E}\langle a \rangle \parallel (\mathcal{G} \lfloor \mathcal{F}) \Longrightarrow P \parallel (Q \lfloor R)$. Thus, $\mathcal{G} \lfloor \mathcal{F} \xrightarrow{\sharp\tilde{\alpha}} Q \lfloor R$. Thus, $\mathcal{G} \lfloor \mathcal{F}_c^* \xrightarrow{\sharp\tilde{\alpha}}$. Hence, $P_{c,i} \ll \mathcal{G} \lfloor \mathcal{F}_c^*$.

(2) Suppose $P \ll C \lfloor F$ and $P' \in Proc(P)$ where P is a component of $\mathfrak{A}$. Let $P \xrightarrow{\tilde{\alpha}} P'$. There is $P_{c,i}$ such that $P_{c,i} \xrightarrow{\tilde{\alpha}}$. Thus, $C \lfloor F_c^*[-i] \xrightarrow{\sharp\tilde{\alpha}} C' \lfloor F'$ for some C', F'. Thus, $(F_c^*[-i] \times P) \parallel C \Longrightarrow (F' \times P') \parallel C'$. Hence, the completeness of $\mathfrak{A}$ follows. □

Proof of Theorem 4.5.10. (1) Suppose $\mathfrak{A}$ is conservative and $\mathcal{G} \xrightarrow{\tilde{\alpha}}_{Act(P_{c,i})}$ for some $P_{c,i}$. Then, let $\mathcal{G} \xrightarrow{\tilde{\beta}}$ and $\tilde{\beta}|Act(P_{c,i}) = \tilde{\alpha}$. Thus, $\mathcal{F}_c^* \xrightarrow{\sharp\tilde{\beta}}$ and $X_{P_{c,i}} \xrightarrow{\sharp\tilde{\alpha}}$. (2) Suppose $\mathcal{G} \preccurlyeq P_{c,i}$ and $\mathcal{G} \xrightarrow{\tilde{\beta}}$. Let $\tilde{\beta}|Act(P_{c,i}) = \tilde{\alpha}_i$ for each i. Then $X_{P_{c,i}} \xrightarrow{\tilde{\alpha}_i}$ for each i and hence $\mathcal{F}_c^* \xrightarrow{\tilde{\beta}}$. □

Proof of Theorem 4.5.11. (1) Suppose $\mathfrak{A}$ is complete and $X_{P_{c,i}} \xrightarrow{\tilde{\alpha}}$. Construct a component $\mathcal{E}\langle a \rangle$ of $\mathfrak{A}$ such that

- $Act(\mathcal{E}\langle a \rangle) \subseteq Act(P_{c,i})$, and

- if $\mathcal{E}\langle a\rangle \xrightarrow{\tilde{\alpha}} P$ and $\mathcal{E}\langle a\rangle \xrightarrow{\tilde{\alpha}} P'$ then $P' = P$.

Then, there is $\mathcal{A} = (\mathcal{F} \times \mathcal{E}\langle a\rangle) \parallel \mathcal{G}$ such that $\mathcal{A} \Longrightarrow (R \times P) \parallel Q$ for some R, Q. Thus, $\mathcal{F} \times \mathcal{E}\langle a\rangle \xrightarrow{\tilde{\alpha}}_{Act(P_{c,i})} R \times P$ and $\mathcal{G} \xrightarrow{\sharp\tilde{\alpha}}_{Act(P_{c,i})} Q$. Thus, $\mathcal{G} \succcurlyeq P_{c,i}$.

(2) Suppose $\mathcal{G} \succcurlyeq P_{c,i}$ and $\mathfrak{A}$ is conservative. Let $\mathcal{E}\langle a\rangle$ be a component (of $\mathfrak{A}$) and $P \in Proc(\mathcal{E}\langle a\rangle)$. Let $P_{c,i} \xrightarrow{\tilde{\alpha}} P$ for some i. Thus, $\mathcal{G} \xrightarrow{\sharp\tilde{\alpha}}_{Act(P_{c,i})} Q$ for some Q. The conservation of $\mathfrak{A}$ entails that there are $\mathcal{F}, \tilde{\beta}$ such that $\mathcal{F} \xrightarrow{\tilde{\beta}} R$ and $\tilde{\beta}|Act(P_{c,i}) = \tilde{\alpha}$ for some R. Thus, $(\mathcal{F} \times \mathcal{E}\langle a\rangle) \parallel \mathcal{G} \Longrightarrow (R \times P) \parallel Q$. The completeness of $\mathfrak{A}$ follows. □

4.7 Related Work and Discussions

The approach in this chapter extends the approach in the previous chapter. Based on the conceptual framework of ADLs, the chapter proposes a process algebraic semantic approach to formally describe the dynamic connector-based architecture. For connector-based systems, because the topology (i.e. the binding of components) is fixed, dynamism is equivalent to runtime insertion and removal of components. In ADLs, component specifications are usually separated into the type and instance levels. This chapter presents a formal definition for semantic conformance in component instantiation, which not only subsumes the parameterisation but also some capture certain semantic variances. This chapter studies the possibility of reducing the verification state space for several basic behavioural properties (i.e. deadlock-freedom, non-starvation, conservation, and completeness) of a dynamic connector basic from the entire universe of possible system configurations to a specific configuration, which is fixed according to the type-level description. To this end, a serial of analytic techniques are presented. To conclude this chapter, we present discussions on the related work and the limitations of the approach that partly motivates the subsequent work in this thesis.

Traditionally, different kinds of process algebra have been employed as the underpinning notations of many ADLs. In particular, our process-algebraic model is closely related to the ADLs explicitly specifying component (and connector) behaviors as protocols, such as Wright [Allen and Garlan, 1997], PADL [Bernardo

et al., 2002], LEDA [Canal et al., 2001], PiLar Cuesta et al. [2005], and π-ADL [Oquendo, 2004]. However, our model is independent of any concrete ADLs and aims to facilitate the formal analysis of specific architecture styles called dynamic connector-based architecture styles. Hence, our work is in the middle of works following the process-algebraic approach to architectural analysis. For example, Inverardi et al. [2000] made use of the model CHAM to describe behaviors of components and developed an algorithm using behavioral equivalence to verify the property of deadlock-freedom. The analytic techniques proposed by Bernardo et al. [2002] are grounded on the distinction of causes of deadlocks: single interactions, combinations of interactions forming an acyclic topology, and those forming a cyclic topology. In particular, they reduced cyclic architectures into basic star-topological ones, which are equivalent to our connector-based configurations, and applied compositional checking to the analysis of these basic architectures described in PADL. Andova et al. [2011] presented a method translating the expressions in Paradigm, a coordination modeling language proposed by the same authors, into the process algebra ACP. By so doing, formal reasoning methods are applicable to the analysis of Paradigm models, such as the behavioral equivalence of two components based on the notion of branching bisimulation. However, all the approaches just mentioned do not address the verification problem of component-aspect dynamism. In our previous work Su et al. [2010], the authors developed methods to facilitate the checking of deadlock-freedom and non-starvation for dynamic connector-based architectures. This work extends the previous work in several aspects as discussed in the Introduction of this paper.

To conclude, we reports several matters around the work in this chapter, which may shed light on some further work. To show the practical adequacy of the approach in this chapter, it is helpful to link the proposed semantic model with a working PA-based ADL. Also, in spite of its expressive power in describing a variety of behavioural specifications of components and connectors, some syntactic restrictions are still applied. For example, components are restricted to be $\times$-free. Similar to the approach in Chapter 3, the approach in this chapter is not immune to the following two general shortcomings. The modelling and analytic techniques in this chapter are for the connector-based architecture; PA in general is two low-level to express less abstract data structures and algorith-

mic structures in a convenience way. Chapter 5 presents a new approach, based on session-typed PA, to deal with the non-connector-based architecture, whilst Chapter 6 provides a method to bridge the gap between the low-level syntax of PA and the coarse-grained specifications in architectural description.

Chapter 5

Peer-to-Peer Architecture

5.1 Introduction

This chapter develops a PA notation with session types to deal with the modelling of the peer-to-peer architecture. To enhance the modularisation of session description, the session types are structured into the communication and integration levels. This chapter is organised as follows. The motivation to use session types in architectural description is explained in Sections 5.1.1 and 5.1.2. The PA notation with session establishment primitives is presented in Section 5.2. The two-level structured session types are given in Section 5.3, with their syntax and semantics, and a method of projecting sessions into roles (for processes). A type system and its relevant properties are provided in Section 5.4 and two properties of architectures, i.e. channel privacy and behavioural conformance of processes with respect to sessions, are analysed in Section 5.5. A process slicing method is provided to complement the type checking in Section 5.6. Complementary materials (proofs and examples) are provided in Section 5.7. This chapter is concluded with brief discussions in Section 5.8.

5.1.1 Peer-to-Peer Architectural Description

The previous chapters (i.e. Chapters 3 and 4) present two approaches and various techniques for the modelling and analysis of the connector-based software architecture. In spite of their usefulness, one obvious restriction is their dependence

on the distinguished topological position of the connector in the architecture. Real systems such as peer-to-peer communicating systems on the other hand may not have this special communication topology. A connector, as a first-class entity in architectural description, does not necessarily have the counterpart in the implementation level [Allen and Garlan, 1997], hence, it is indeed possible to describe arbitrary systems in the model of the connector-based architecture. However, this 're-constructive' approach has at least two disadvantages. Firstly, it makes the specifications apart from the implementations. Secondly, it brings in an overhead of protocols. For example, a direct message transit between two components in the implementation needs to be specified as two message transits that pass by the connector. To provide a suitable formal notation for the peer-to-peer architectural modelling, this chapter employs the theory of session types.

5.1.2 Session Types and Session Composition

Informally, a session is a unit of message-based communications with a specific purpose. The suitability of session types for describing distributed computing includes two aspects:

- Session types provide a global descriptive method for the protocols written in PA notations such as π-calculus;
- For protocol modularisation, session types project fragments of the protocols on the usage of private channels for intended classes of participants.

The first feature facilitates the protocol design and verification. As an example, Bernardo et al. [2002] presented the malicious host protocol: when arriving the party, the guest expects to be welcomed by the host and to be asked whether (s)he wants an orange juice or a pineapple juice; then, the guest expects the host to tell the waiter brings the drink he likes; however, the host is a malicious person and tells the waiter a drink different from the guest's desired one. There is certainly a deadlock in the interactions between the three people. With a global description of the protocol, the deadlock can be easily identified. The second feature corresponds to the binding of ports of components in architectural

description: the privacy of channels within components in a session can be enhanced (so that each channel is used by a unique pair of components) to ensure the correct binding of ports between two components.

The session-based specifications of a processes should not be assumed as corresponding to different independent behavioural threads. In the actual distributed computing, there are often meaningful interplays between a number of sessions. For example, the following business protocol consists of four sessions between five agents. A broker and two buyers are in the auction session *Auction* (where *Auction* is seen as a global description of the auction protocol). *Auction* is followed by a transaction session between the auction winner, the broker, and the seller. Two alternative transaction protocols are given for the winner to choose: *DTransaction* is a direct protocol, in which the winner directly transfers money to the seller; *STransaction* is a secure protocol, in which the money is transferred via the broker and an extra (sub-)session *EPay* money transaction between the winner, the bank, and the broker is involved. Therefore, the whole business protocol is the integration of four sessions in the intended ways. It is indeed possible to view the protocol as inseparable, but so-doing violates both the natural understanding of the protocol and the gradual procedure of requirement specification.

As one main purpose of the session type theory is to use the session-based description for reasoning about behaviours of processes (or components), it is helpful to express the aforementioned ways of session composition as sessions themselves. This motivates a two-level session type theory, in which sessions in the communication level (called communicating sessions) specify the end-point communications of multiple components, whilst sessions in the integration level (called integrating sessions) describe the gluing of communicating sessions by concatenation, choice, interleaving composition, and nesting composition.

5.2 Process Algebra $\mathcal{PA}_s$

This section defines a variant of π-calculus, denoted $\mathcal{PA}_s$. In the next two sections, a type discipline based on two-level session types is developed for the calculus. Compared with the existing session type literature, the syntax of our calculus is abstract and close to the original presentation of π-calculus. Our intention is to

$P ::= \pi.P$		prefixing	$\alpha ::= \pi$		action
	$(\nu a)P$	hiding		τ	silence
	$r : P$	labelling	$\pi ::= \bar{a}_{[2..n]}(\tilde{c})$		invitation
	$\mathbf{0}$	inaction		$a_{[k]}(\tilde{c})$	acceptance
	X	variables		$a!v$	sending
	$[\mathrm{rec}\, X]P$	recursion		$a?v$	receiving
	$P + P$	choice			
	$P \mid P$	parallel			

Figure 5.1: Syntax of $\mathcal{PA}_s$

minimise the side techniques (we return to this point in Section **??**).

The basic sets are a set of *channels* (a, b, c, c'), a set of *messages* or *message types* (u, v, v'), and a set of *participant names* $(p, q, r, 1, 2, \text{etc.})$. The syntax of *processes* and *actions* is given in Figure 5.1. Sessions, which are informally understood as units of interactions, are established by shared channels. The key syntactic primitives for channel establishment are of the forms $\bar{a}_{[2..n]}(\tilde{c})$ and $a_{[k]}(\tilde{c})$, which are due to Honda et al. [2008]. These two prefixes are called *session actions* and a is called a *session channel*. $\bar{a}_{[2..n]}(\tilde{c})$ invites participants 2 to n to join in a session whose communicating channels are $\tilde{c}$, whilst $a_{[k]}(\tilde{c})$ accepts a session invitation. By the operational semantics, when the actions $\bar{a}_{[2..n]}(\tilde{c})$ and $a_{[k]}(\tilde{c})$ (for each $2 \leq k \leq n$) are triggered synchronously, a session is established via the session channel a and a sequence of fresh communicating channels $\tilde{c}$ are generated (unlike [Honda et al., 2008], in which the message transport is asynchronous). In $r : P$, r labels P and is seen as the name of P. In some literature, it is also called the location of P [Hennessy, 2007]. Other syntactic primitives and constructions are standard and from π-calculus.

Binders are a in $(\nu a)P$, $\tilde{c}$ in $\bar{a}_{[2..n]}(\tilde{c}).P$ or $a_{[k]}(\tilde{c}).P$, and X in $[\mathrm{rec}\, X]P$. Substitution of channels are standard. In particular, $([\mathrm{rec}\, X]P)\{a/b\} = [\mathrm{rec}\, X](P\{a/b\})$. $(\nu\tilde{a})P$ stands for $(\nu a_1)\ldots(\nu a_n)P$ where $\tilde{a} = a_1, \ldots, a_n$. The left-associative law is adopted when presenting multiple $\mid$ or $+$. We assume the bound name convention for processes, namely, bound names are different from free names in a given process. Let $\mathrm{fc}(P)$ and $\mathrm{fc}(\alpha)$ denote the set of free channels in P and α, re-

$$
\begin{gathered}
P \mid Q \equiv Q \mid P \qquad P \mid \mathbf{0} \equiv P \\
(P \mid Q) \mid R \equiv P \mid (Q \mid R) \\
P + Q \equiv Q + P \qquad P + \mathbf{0} \equiv P \\
(P + Q) + R \equiv P + (Q + R) \\
(\nu a)\mathbf{0} \equiv \mathbf{0} \qquad (\nu a)(\nu b)P \equiv (\nu b)(\nu a)P \\
(\nu a)P \mid Q \equiv (\nu a)(P \mid Q) \ \text{ if } a \notin \mathrm{fc}(Q) \\
[\mathrm{rec}\, X]R \equiv R \ \text{ if } X \notin \mathrm{fv}(R) \\
P \equiv Q \ \text{ if } P =_\alpha Q \\
l' : l : P \equiv l' : P \\
l : P \mid l : Q \equiv l : (P \mid Q) \\
l : (\nu a)P \equiv (\nu a) l : P
\end{gathered}
$$

Figure 5.2: Structural congruence of $\mathcal{PA}_s$

spectively. $\mathrm{fv}(P)$ is the set of free process variables, and $\mathrm{act}(P)$ the set of prefixes in P. Supposing $\tilde{a} = \mathrm{fc}(P)$ and $|\tilde{c}| = |\tilde{a}|$, $P\langle\tilde{c}\rangle$ refers $P\{\tilde{c}/\tilde{a}\}$, the simultaneous substitution of $\tilde{a}$ by $\tilde{c}$.

The structural congruence $\equiv$ is the smallest congruent relation on processes that includes the equations in Figure 5.2. $P =_\alpha Q$ means that P and Q are variants of *alpha-conversion*. Note that we leave out the equi-recursive equation (e.g. $[\mathrm{rec}\, X]P \equiv P\{[\mathrm{rec}\, X]P/X\}$) in the structural laws (it is called recursion-free or replication-free structural congruence in some literature [Engelfriet and Gelsema, 2004]). Consequently, we have the decidability of structural congruence.

Lemma 5.2.1. *For any given P, Q, it is decidable if $P \equiv Q$.*

If $X \in \mathrm{fv}(P)$, we let $P^{[X]}$ be $[\mathrm{rec}\, X]P$; otherwise, $P^{[X]}$ is P. $P \sqsubseteq Q$ means $P + R \equiv Q$ for some R, and $P \sqsubset Q$ means $P + R \equiv Q$ for some $R \not\equiv \mathbf{0}$. $P \sqcup Q$ is defined as follows: if $Q \sqsubseteq P$ then $P \sqcup Q = P$; if $P \sqsubset Q$ then $P \sqcup Q = Q$; otherwise, $P \sqcup Q = P + Q$.

The operational semantics are given through a labelled transition system defined by rules in Figure 5.3. In the session type literature, the semantics of the process calculus is defined as a reduction system instead of a labelled transition one. The advantage of the former over the latter is a simpler presentation. But

$$a!v.P \xrightarrow{a!v} P \qquad \text{[Sed-s]}$$

$$a?v.P \xrightarrow{a?v} P \qquad \text{[Rcv-s]}$$

$$\bar{a}_{[2..n]}(\tilde{c}).P \xrightarrow{\bar{a}_{[2..n]}(\tilde{c})} P \qquad \text{[Inv-s]}$$

$$a_{[k]}(\tilde{c}).P \xrightarrow{a_{[k]}(\tilde{c}')} P\{\tilde{c}'/\tilde{c}\} \qquad \text{[Acc-s]}$$

$$\frac{P \xrightarrow{\alpha} P'}{P \mid Q \xrightarrow{\alpha} P' \mid Q} \qquad \text{[Par-s]}$$

$$\frac{P \xrightarrow{\alpha} P'}{P + Q \xrightarrow{\alpha} P'} \qquad \text{[Sum-s]}$$

$$\frac{P \xrightarrow{a!v} P' \qquad Q \xrightarrow{a?v} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q} \qquad \text{[Com-s]}$$

$$\frac{P \xrightarrow{\alpha} P' \quad \mathrm{fc}(\alpha) \neq a}{(\nu a)P \xrightarrow{\alpha} (\nu a)P'} \qquad \text{[Hid-s]}$$

$$\frac{P_1 \xrightarrow{\bar{a}_{[2..n]}(\tilde{c})} P_1' \quad P_i \xrightarrow{a_{[i]}(\tilde{c})} P_i' \ (\forall\, 2 \leq i \leq n)}{P_1 \mid \ldots \mid P_n \xrightarrow{\tau} (\nu\tilde{c})(P_1' \mid \ldots \mid P_n')} \qquad \text{[Sess-s]}$$

$$\frac{P \xrightarrow{\alpha} P'}{l : P \xrightarrow{\alpha} l : P'} \qquad \text{[Lab-s]}$$

$$\frac{P \xrightarrow{\alpha} P'}{[\mathrm{rec}\, X]P \xrightarrow{\alpha} P'\{[\mathrm{rec}\, X]P/X\}} \qquad \text{[Rec-s]}$$

$$\frac{P \equiv Q \quad Q \xrightarrow{\alpha} Q' \quad Q' \equiv P'}{P \xrightarrow{\alpha} P'} \qquad \text{[Eqv-s]}$$

Figure 5.3: Operational semantics of $\mathcal{PA}_s$

because one of our purposes in the present paper is to study the behavioural relation between processes and their session types, the standard operational semantics are more suitable to this end. The rules [Inv-s], [Acc-s], and [Sess-s] handle the session establishment, and their intuitive meanings have been explained. [Lab-s] is for process labelling. The rest of the semantic rules are standard.

Let $\mathrm{proc}(P) = \{Q \mid P \xrightarrow{\tilde{\alpha}} Q\}$. P *simulates* Q, denoted $P \succ Q$, if there is a relation $\mathcal{R} \subseteq \mathrm{proc}(P) \times \mathrm{proc}(Q)$ such that if $\langle P, Q\rangle \in \mathcal{R}$ and $Q \xrightarrow{\alpha} Q'$ then there is P' such that $P \xrightarrow{\alpha} P'$ and $\langle P', Q'\rangle \in \mathcal{R}$. We use $P \xrightarrow{\alpha}_{\succ} Q$ to mean that there is R such that $P \xrightarrow{\alpha} R$ and $R \succ Q$. We say P is *deterministic* (up to structural congruence) if for each $Q \in \mathrm{proc}(P)$, $Q \xrightarrow{\alpha} R$ and $Q \xrightarrow{\alpha} R'$ entail $R \equiv R'$.

Examples of agent behaviours We provide a detailed, but informal description of the interactions between the five agents in the example from Introduction, and then formulate their individual behaviours in the calculus. In the upmost level, the whole business protocol is divided into two stages. The first stage is for the auction session and the second one includes two alternative transaction sessions and a possible nested sub-session.

At the auction stage, the broker initiates the auction session with two buyers, i.e. buyer_1 and buyer_2. For simplicity, we assume that the buyers already know the base price of the auctioned item. After the auction is initiated, buyer_1 (resp. buyer_2) sends its bid to the broker and the protocol reaches a recursive state. In the recursive state, the broker sends a new quote to the other buyer and the protocol proceeds in the following two alternative branches. (a) If the other buyer does not bid (after some amount of time), then the broker issues an invoice to buyer_1 (resp. buyer_2), finishing the auction. (b) If the other buyer bids, then the broker forwards the latest quote to buyer_1 (resp. buyer_2) and, again, the protocol has two sub-branches: (b1) if buyer_1 (resp. buyer_2) continues to bid, then the protocol returns to the recursive state; (b2) otherwise, the broker issues buyer_2 (resp. buyer_1) an invoice to finish the auction.

At the transaction stage, the buyer that won the auction initiates one of the following two transaction options. (a) If the direct transaction is chosen, then the broker forwards the price to the seller, and the buyer makes the payment to the seller and receives the ordering information. (b) If the secure transac-

tion is chosen, an extra bank transfer session is involved. The buyer authorises the bank to transfer an intended amount of money to the broker. The broker holds the money but informs the seller that the pre-payment is ready, and seller sends the ordering information to the buyer. After receiving the item, the buyer sends a confirmation message to the broker and the broker finalises the deal by transferring the pre-payment to the seller.

The formal description of the broker's behaviour is given by the following process.

$$
\begin{aligned}
P_{\text{broker}} \stackrel{\text{def}}{=} \overline{\text{auc}}_{[2..3]}(a_{1,2}, a_{1,3}).\ \sum_{i\in\{1,2\}} (a_{1,i+1}?\text{bid}.\ [\text{rec}\, X](a_{1,4-i}!\text{quote}.\\
(a_{1,i+1}!\text{invoice}.\ P^{i}_{\text{broker}} + a_{1,4-i}?\text{bid}.\ a_{1,i+1}!\text{quote}.\\
(a_{1,i+1}?\text{bid}.\ X + a_{1,4-i}!\text{invoice}.\ P^{3-i}_{\text{broker}}))))\\
P^{i}_{\text{broker}} \stackrel{\text{def}}{=} \text{dTran}^{i}_{[2]}(b_{1,2}, b_{2,3}).\ b_{2,3}!\text{price}.\ \mathbf{0} + \text{sTran}^{i}_{[2]}(c_{1,2}, c_{1,3}, c_{2,3}).\\
\text{epay}^{i}_{[2]}(d_{1,3}, d_{2,3}).\ d_{1,3}?\text{transfer}.\ c_{2,3}!\text{prepaid}.\\
c_{1,2}?\text{confirm}.\ c_{2,3}!\text{payment}.\ \mathbf{0}
\end{aligned}
$$

As explained before, the session is established via shared channels, i.e. session channels, one of which is auc. The prefix $\overline{\text{auc}}_{[2..3]}(a_{1,2}, a_{1,3})$ initiates a session with another two participants (three in total), which, in this case, are the two buyers. $\text{dTran}^{i}_{[2]}(b_{1,2}, b_{2,3})$, $\text{sTran}^{i}_{[2]}(c_{1,2}, c_{1,3}, c_{2,3})$, and $\text{epay}^{i}_{[2]}(d_{1,3}, d_{2,3})$ $(i \in \{1, 2\})$ are for accepting session establishment. Other prefixes are ordinary prefixes in π-calculus. We use $a_{i,j}$ to denote the channel used by the ith and jth agents in the session. For example, $a_{1,2}$ is the communicating channel between the first and second participants, which, in this case, are the broker and buyer_1. The recursive structure in P_{broker} corresponds to the recursive state of the auction protocol informally described above.

The behaviours of the two buyers in the two stages of the protocol follow. Let

$j \in \{1, 2\}$.

$$P_{\text{buyer}_j} \stackrel{\text{def}}{=} \text{auc}_{[j+1]}(a_{1,2}, a_{1,3}).\ a_{1,j+1}!\text{bid}.\ [\text{rec}\, X_1](a_{1,j+1}?\text{quote}.\ a_{1,j+1}!\text{bid}.\ X + a_{1,j+1}?\text{invoice}.\ P'_{\text{buyer}_i}) + a_{1,j+1}?\text{quote}.\ [\text{rec}\, X_2] (a_{1,j+1}!\text{bid}.\ (a_{1,j+1}?\text{invoice}.\ P'_{\text{buyer}_j} + a_{1,j+1}?\text{quote}.\ X_2))$$

$$P'_{\text{buyer}_j} \stackrel{\text{def}}{=} \overline{\text{dTran}}^{j}_{[2..3]}(b_{1,3}, b_{2,3}).\ b_{1,3}!\text{payment}.\ b_{1,3}?\text{order}.\ \mathbf{0} + \overline{\text{sTran}}^{j}_{[2..3]}(c_{1,2}, c_{1,3}, c_{2,3}).\ \overline{\text{epay}}_{[2..3]}(d_{1,3}, d_{2,3}).\ d_{1,3}!\text{amount}.\ d_{2,3}?\text{transfer}.\ c_{1,3}?\text{order}.\ \text{c}_{1,2}!\text{confirm}.\ \mathbf{0}$$

The seller and the bank only take part in the second part of the protocol, and their behaivours follow.

$$P_{\text{seller}} \stackrel{\text{def}}{=} \sum_{i \in \{1,2\}} (\text{dTran}^{i}_{[3]}(b_{1,3}, b_{2,3}).\ b_{2,3}?\text{price}.\ b_{1,3}?\text{payment}.\ b_{1,3}!\text{order}.\ \mathbf{0} + \text{sTran}^{i}_{[3]}(c_{1,2}, c_{1,3}, c_{2,3}).\ c_{2,3}?\text{prepaid}.\ c_{1,3}!\text{order}.\ c_{1,3}!\text{payment}.\ \mathbf{0})$$

$$P_{\text{bank}} \stackrel{\text{def}}{=} \text{epay}_{[3]}(d_{1,3}, d_{2,3}).\ d_{1,3}?\text{amount}.\ d_{2,3}!\text{transfer}.\ \mathbf{0}$$

The interactions of five processes, i.e. P_{broker}, P_{buyer_1}, P_{buyer_2}, P_{seller}, and P_{bank} according the operational semantics should follow the presented scenario.

5.3 Two-level Session Types

5.3.1 Syntax and Semantics

We provide the syntax of *session types* or *sessions*, and then define the two kinds of sessions studied in the present paper, i.e. communicating and integrating sessions. Communicating session types (B, B') are given in 5.4 and integrating session types (A, A') are given in 5.4. In the following, for the convenience of discussions, we use S, T to denote session types generally.

$\langle p, q : v \rangle \to S$ is a session of *communication* form, meaning that, after the agent p sends the message (type) v to the agent q, the session proceeds as S.

$B ::= \langle p, q : v\rangle \to B$	communication		$\mid$ `end`	termination
$\mid B \oplus B$	union		$\mid \mathbf{t}$	type variable
$\mid B \otimes B$	product		$\mid \mu\mathbf{t}.B$	recursion
$A ::= \langle \tilde{p} : B\rangle\{A\}$	communication		$\mid$ `end`	termination
$\mid A \oplus A$	union		$\mid \mathbf{t}$	type variable
$\mid A \otimes A$	product		$\mid \mu\mathbf{t}.A$	recursion
$\mid A; A$	concatenation			

Figure 5.4: Syntax of communicating and integrating sessions

$\langle \tilde{p} : S\rangle\{T\}$ is a session of *establishment* form, meaning that the agents whose names are given by $\tilde{p}$ establish a session S which nests T. The first item in the sequence $\tilde{p}$ refers to the participant that initiates S. We call $\langle p, q : v\rangle$ and $\langle \tilde{p} : S\rangle$ *event prefixes*. $S; T$ is the *concatenation* of S and T, $S \oplus T$ is their *union*, and $S \otimes T$ their *product*. $\mathbf{t}$ is a type variable and $\mu\mathbf{t}.S$ is a recursive type and binds $\mathbf{t}$ in S in the standard way. A session is *closed* if occurrences of all variables in it are bound. `end` is a terminated session.

If T is contained in (the syntax of) S, we call T a sub-session of S. $\mathrm{pid}(S)$ is the sequence that enumerates the participant names in S. For convenience, we also use $\mathrm{pid}(S)$ to denote set of names belonging to $\mathrm{pid}(S)$. S is *well-formed*, if (1) for each sub-session $\langle p, q : v\rangle \to T$ of S, $p \neq q$, and (2) for each sub-session $\langle \tilde{p} : T\rangle\{T'\}$ of S, $\tilde{p}$ is a sequence of pair-wise different names, and $|\tilde{p}| = |\mathrm{pid}(T)|$. Hence, well-formedness of sessions rules out self interactions such as $\langle p, p : v\rangle$ and multiple participation of sessions such as $\langle p, p : S\rangle$. For each S, we define a set $\mathrm{opid}(S)$ as follows:

- $\mathrm{opid}(\mathbf{t}) = \mathrm{opid}(\mathtt{end}) = \varnothing$; $\mathrm{opid}(\mu\mathbf{t}.S) = \mathrm{opid}(S)$;
- $\mathrm{opid}(\langle p, q : v\rangle \to S) = \{\{p, q\}\}$; $\mathrm{opid}(\langle \tilde{p} : S\rangle\{T\}) = \{\tilde{p}\}$;
- $\mathrm{opid}(S \oplus T) = \mathrm{opid}(S \otimes T) = \mathrm{opid}(S) \cup \mathrm{opid}(T)$;
- – if $\mathrm{opid}(S) \neq \varnothing$, then $\mathrm{opid}(S; T) = \mathrm{opid}(S)$; and
  – if $\mathrm{opid}(S) = \varnothing$, then $\mathrm{opid}(S; T) = \mathrm{opid}(T)$.

Then, *race-free* sessions are recursively defined as follows. (1) $\mathtt{end}$ and $\mathrm{opid}(\mathbf{t})$ are race-free; (2) if S is race-free, then so is $\mu\mathbf{t}.S$; (3) if S is race-free and $\{p,q\}\cap H \neq \varnothing$ for each $H \in \mathrm{opid}(S)$, then $\langle p,q:v\rangle \to S$ is race-free for any v; (4) if S,T are race-free, then $\langle \tilde{p} : S\rangle\{T\}$ is race-free; (5) if S,T are race-free and $H \cap H' \neq \varnothing$ for each $H \in \mathrm{opid}(S), H' \in \mathrm{opid}(T)$, then $S \oplus T$ is race-free; (6) if S,T are race-free then $S \otimes T$ is race-free; (7) if T is race-free and $\mathrm{pid}(S) = \varnothing$, then $S;T$ is race-free; and (8) if S,T are race-free, $\mathrm{pid}(S) \neq \varnothing$, and $\mathrm{pid}(S') \cap H \neq \varnothing$ for each $H \in \mathrm{opid}(T)$ and each sub-session S' of S, then $S;T$ is race-free.

Throughout this chapter, we assume the well-formedness and race-freedom of sessions, both of which are preserved by the laws of the structural congruence and operational semantics (Figures 5.5 and 5.6, defined below). A further requirement, which is need in the role projection (see Section 5.3.3), is as follows: S is *single-threaded*, if for each sub-session $T;T'$ of S, T does not contain $\otimes$. In the sequel, if we need a session to be single-threaded, this is either clear in the context or made explicitly. Single-threadedness of sessions is *not* preserved by the laws of the structural congruence and operational semantics, however, this does not affect the technical results of the paper. A minor remark is that since ; does not appear in communicating sessions, so communicating sessions are always single-threaded.

Let $\tilde{p}$ be a distinct sequence and $|\tilde{p}| = \mathrm{pid}(S)$. We use $S\langle\tilde{p}\rangle$ to denote the simultaneous substitution of $\tilde{p}$ for $\mathrm{pid}(S)$ in S.

For simplicity, for a given communicating session B, we let $\mathrm{pid}(B)$ be a sequence of consecutive integral numbers from 1. However, when writing $B\langle\tilde{p}\rangle$, $\tilde{p}$ is not necessarily a sequence of consecutive integrals. We use $\langle\tilde{p} : S\rangle$ to abbreviate $\langle\tilde{p} : S\rangle\{\mathtt{end}\}$. *Auction*, *DTransaction*, *STransaction*, and *EPay* in Section 5.3.2 below are communicating sessions. *Proto* in Section 5.3.2 below is an integrating session.

The structural congruence of sessions is the smallest congruent relation containing the equations presented in Figure 5.5. These laws of structural congruence have a strong correspondence to those for processes in Figure 5.2. Because the equi-recursive equation $\mu\mathbf{t}.S \equiv S\{\mu\mathbf{t}.S/\mathbf{t}\}$ is left out, the session structural congruence is also decidable.

The operational semantics of sessions are presented in Figure 5.6, where we

$$
\begin{gathered}
(S \oplus S') \oplus S'' \equiv S \oplus (S' \oplus S'') \\
S \oplus S' \equiv S' \oplus S \qquad S \oplus \mathtt{end} \equiv S \\
(S \otimes S') \otimes S'' \equiv S \otimes (S' \otimes S'') \\
S \otimes S' \equiv S' \otimes S \qquad S \otimes \mathtt{end} \equiv S \\
(S; S'); S'' \equiv S; (S'; S'') \\
S; \mathtt{end} \equiv S \qquad \mathtt{end}; S \equiv S \\
\mu \mathbf{t}.S \equiv S \quad \text{if } \mathbf{t} \notin \mathrm{fv}(S) \\
S \equiv T \quad \text{if } S =_\alpha T
\end{gathered}
$$

Figure 5.5: Session structural congruence

use λ to denote $p, q : v$ or $\tilde{p} : B$. [S-COM] describes the ordinary message-based communications between two participants. [S-SESS] is for the session establishment and nesting; when a session is established, it runs interleavingly with its nested session. Other semantic rules are standard. [S-TIMES], [S-SUM], and [S-CON] handle the session production, summation, and concatenation, respectively. Recursive sessions are dealt with by [S-REC] and session equivalence by [S-EQ].

5.3.2 Examples of Sessions

We use the syntax of two-level sessions to formulate our business protocol. *Proto* is the session at the integrating level whilst the remaining four are at the communicating level. Their intuitive explanations have already been given in the last part of Section 5.2. We can also find a correspondence between *Proto* and its

$$\langle p, q : v\rangle \to B \xrightarrow{p,q:v} B \qquad \text{[S-COM]}$$

$$\langle \tilde{p} : B\rangle\{A\} \xrightarrow{\tilde{p}:B} A \otimes B\langle\tilde{p}\rangle \qquad \text{[S-SESS]}$$

$$\frac{S \xrightarrow{\lambda} S'}{S \otimes T \xrightarrow{\lambda} S' \otimes T} \qquad \text{[S-TIMES]}$$

$$\frac{S \xrightarrow{\lambda} S'}{S \oplus T \xrightarrow{\lambda} S'} \qquad \text{[S-SUM]}$$

$$\frac{S \xrightarrow{\lambda} S'}{S; T \xrightarrow{\lambda} S'; T} \qquad \text{[S-CON]}$$

$$\frac{S \xrightarrow{\lambda} S'}{\mu\mathbf{t}.S \xrightarrow{\lambda} S'\{\mu\mathbf{t}.S/\mathbf{t}\}} \qquad \text{[S-REC]}$$

$$\frac{S \equiv T \quad T \xrightarrow{\lambda} T' \quad T' \equiv S'}{S \xrightarrow{\lambda} S'} \qquad \text{[S-EQ]}$$

Figure 5.6: Session operational semantics

rough description in Introduction. $\bigoplus$ is the multiple case of $\oplus$.

$$\begin{aligned}
\mathit{Proto} \stackrel{\text{def}}{=}\ & \langle \text{broker}, \text{buyer}_1, \text{buyer}_2 : \mathit{Auction}\rangle\{\}; \bigoplus_{i\in\{1,2\}} \langle \text{buyer}_i, \\
& \quad \text{broker}, \text{seller} : \mathit{DTransaction}\rangle\{\} \ \oplus \langle \text{buyer}_i, \text{broker}, \\
& \quad\quad \text{seller} : \mathit{STransaction}\rangle\{\langle \text{buyer}_i, \text{broker}, \text{bank} : \mathit{EPay}\rangle\{\}\} \\
\mathit{Auction} \stackrel{\text{def}}{=}\ & \bigoplus_{i\in\{2,3\}} \langle i, 1 : \text{bid}\rangle \to \mu\mathbf{t}.\langle 1, 5-i : \text{quote}\rangle \to \\
& \quad (\langle 1, i : \text{invoice}\rangle \to \texttt{end} \oplus \langle 5-i, 1 : \text{bid}\rangle \to \langle 1, i : \text{quote}\rangle \\
& \quad\quad \to (\langle i, 1 : \text{bid}\rangle \to \mathbf{t} \oplus \langle 1, 5-i : \text{invoice}\rangle \to \texttt{end})) \\
\mathit{DTransaction} \stackrel{\text{def}}{=}\ & \langle 2, 3 : \text{price}\rangle \to \langle 1, 3 : \text{payment}\rangle \to \langle 3, 1 : \text{order}\rangle \to \texttt{end} \\
\mathit{STransaction} \stackrel{\text{def}}{=}\ & \langle 2, 3 : \text{prepaid}\rangle \to \langle 3, 1 : \text{order}\rangle \to \langle 1, 2 : \text{confirm}\rangle \\
& \quad \to \langle 2, 3 : \text{payment}\rangle \to \texttt{end} \\
\mathit{EPay} \stackrel{\text{def}}{=}\ & \langle 1, 3 : \text{amount}\rangle \to \langle 3, 2 : \text{transfer}\rangle \to \texttt{end}
\end{aligned}$$

A comparison of the above session formulation and the behavioural formula-

tion of the five agents in Section 5.2 leads us to see three advantages of session modularisation and integration. First, sessions characterise the interactions between processes globally, facilitating the prevention of deadlocks and race conditions. Also, sessions are free of channels. Finally, following the principle of separation of concerns, communicating sessions partition protocols into independent modules whilst integrating sessions assemble communicating sessions at an adequately abstract level. The last aspect is unique to our theory and its merits are two-fold: it fits the natural understanding of the protocol and the gradual procedure of protocol formulation.

5.3.3 Role Projection

Session roles or *roles* refer to behaviours of participants acting in sessions. In other words, roles are the local description of sessions for participants. Formally, roles are represented as abstract processes. The goal of this sub-section is to develop mechanisms to project communicating and integrating sessions into their roles. The projection forms a basis for the type system developed later.

We first deal with the projection of communicating sessions. First, we mark *each* occurrence of *each* event prefix in a given session by a unique channel name. Then, we map the given session into processes according to the following rules:

- $\mathtt{end}\upharpoonright r = \mathbf{0}$, $\mathbf{t}\upharpoonright r = X_{\mathbf{t}}$, $(\mu\mathbf{t}.B)\upharpoonright r = [\mathrm{rec}\, X_{\mathbf{t}}](B\upharpoonright r)$,
- $(\langle p, q : v\rangle \to B)\upharpoonright r =$

$$\begin{cases} c!v.(B\upharpoonright r) & \text{if } r = p \\ c?v.(B\upharpoonright r) & \text{if } r = q \\ B\upharpoonright r & \text{if } r \notin \{p, q\} \end{cases}$$

  where the leftmost occurrence of $\langle p, q : v\rangle$ in $\langle p, q : v\rangle \to B$ is marked by c,
- $(B \oplus B')\upharpoonright r = B\upharpoonright r + B'\upharpoonright r$, and $(B \otimes B')\upharpoonright r = B\upharpoonright r \mid B'\upharpoonright r$.

After the first two steps, we obtain a set of processes such that the *message flow* between them at the runtime (according to the operational semantics) is *deterministic* (and so channel interference is avoided). We say the message flow

between the P_1 to P_m deterministic if $P_1 \mid \ldots \mid P_m$ is deterministic. However, the number of channels used in sessions many be large. In the third step, we apply a channel substitution to the set of roles to optimise channel usage. The definition of the channel substitution is subject to practical considerations. For example, one may let two agents use the same channel to communicate, just as we did for the processes of five agents in the protocol example. The channel substitution is *legal* as long as the message flow between the resulted processes remains deterministic. Without confusion, when writing $B\upharpoonright r$, we always refer to the optimised $B\upharpoonright r$ and call it the role of B for r. The projection is well-defined based on the well-formedness of communicating sessions.

The projection for integrating sessions is similar, but single-threadedness is required. Also, usually the number of communicating sessions in an integrating session is not very large, therefore we omit the channel optimisation step. First, we mark each *occurrence* of the prefix in a given integrating session by a unique channel name. Then, we map the given session into processes according to the following rules:

- $\mathtt{end}\upharpoonright r = \mathbf{0}$, $\mathbf{t}\upharpoonright r = X_{\mathbf{t}}$, $(\mu\mathbf{t}.A)\upharpoonright r = [\mathrm{rec}\, X_{\mathbf{t}}](A\upharpoonright r)$,
- $\langle\tilde{p} : B\rangle\{A\}\upharpoonright r =$

$$\begin{cases} \bar{a}_{[2..n]}(\tilde{c}).(A\upharpoonright r) & \text{if } r = \tilde{p}[1] \wedge |\mathrm{pid}(B)| = |\tilde{p}| = n \wedge \\ & \quad |\mathrm{fc}(B\upharpoonright r)| = |\tilde{c}| \wedge \tilde{c} \cap \mathrm{fc}(A\upharpoonright r) = \varnothing \\ a_{[k]}(\tilde{c}).(A\upharpoonright r) & \text{if } r = \tilde{p}[k] \wedge |\mathrm{fc}(B\upharpoonright r)| = |\tilde{c}| \wedge \tilde{c} \cap \mathrm{fc}(A\upharpoonright r) = \varnothing \\ A\upharpoonright r & \text{if } r \notin \tilde{p} \end{cases}$$

  where the leftmost occurrence of $\langle\tilde{p} : B\rangle$ in $\langle\tilde{p} : B\rangle\{A\}$ is marked by a,

- $(A; A')\upharpoonright r = (A\upharpoonright r)\{A'\upharpoonright r/\mathbf{0}\}$, $(A \oplus A')\upharpoonright r = A\upharpoonright r + A'\upharpoonright r$, and $(A \otimes A')\upharpoonright r = A\upharpoonright r \mid A'\upharpoonright r$.

The process $A\upharpoonright r$ is the role of A for r. The projection rule $(A; A')\upharpoonright r = (A\upharpoonright r)\{A'\upharpoonright r/\mathbf{0}\}$ is well-defined based on the single-threadedness of integrating sessions, and others are well-defined by the well-formedness of integrating sessions.

The projection is completely automated for integrating sessions, but it presupposes a legal channel substitution for communicating sessions to optimise the

channel usage.

Examples of roles The following set of processes contains roles of *Proto* for five agents (the first five) and roles of all four communicating sessions for the broker (the last four). Let $j \in \{1, 2\}$.

$$R^{\text{all}}_{\text{broker}} \stackrel{\text{def}}{=} \overline{\text{auc}}_{[2..3]}(a_{1,2}, a_{1,3}).\ \sum_{i\in\{1,2\}} (\text{dTran}^i_{[2]}(b_{1,3}, b_{2,3}).\ \mathbf{0} + \text{sTran}^i_{[2]}(c_{1,2}, c_{1,3}, c_{2,3}).\ \text{epay}^i_{[2]}(d_{1,3}, d_{2,3}).\ \mathbf{0})$$

$$R^{\text{all}}_{\text{buyer}_j} \stackrel{\text{def}}{=} \text{auc}_{[j+1]}(a_{1,2}, a_{1,3}).\ (\overline{\text{dTran}}^j_{[2..3]}(b_{1,3}, b_{2,3}).\ \mathbf{0} + \overline{\text{sTran}}^j_{[2..3]}(c_{1,2}, c_{1,3}, c_{2,3}).\ \overline{\text{epay}}^j_{[2..3]}(d_{1,3}, d_{2,3}).\ \mathbf{0})$$

$$R^{\text{all}}_{\text{seller}} \stackrel{\text{def}}{=} \sum_{i\in\{1,2\}} (\text{dTran}^i_{[3]}(b_{1,3}, b_{2,3}).\ \mathbf{0} + \text{sTran}^i_{[3]}(c_{1,2}, c_{1,3}, c_{2,3}).\ \mathbf{0})$$

$$R^{\text{all}}_{\text{bank}} \stackrel{\text{def}}{=} \sum_{i\in\{1,2\}} \text{epay}^i_{[3]}(d_{1,3}, d_{2,3}).\ \mathbf{0}$$

$$R^{\text{auc}}_{\text{broker}} \stackrel{\text{def}}{=} \sum_{i\in\{1,2\}} (a_{1,i+1}?\text{bid}.\ [\text{rec}\, X_1](a_{1,4-i}!\text{quote}.\ (a_{1,i+1}!\text{invoice}.\ \mathbf{0}.\ + a_{1,4-i}?\text{bid}.\ a_{1,i+1}!\text{quote}.\ (a_{1,i+1}?\text{bid}.\ X_1 + a_{1,4-i}!\text{invoice}.\ \mathbf{0}))))$$

$$R^{\text{sTran}}_{\text{broker}} \stackrel{\text{def}}{=} c_{2,3}!\text{prepaid}.\ c_{1,2}?\text{confirm}.\ c_{2,3}!\text{payment}.\ \mathbf{0}$$

$$R^{\text{dTran}}_{\text{broker}} \stackrel{\text{def}}{=} b_{2,3}!\text{price}.\ \mathbf{0} \qquad R^{\text{epay}}_{\text{broker}} \stackrel{\text{def}}{=} d_{2,3}?\text{transfer}.\ \mathbf{0}$$

5.4 Type Discipline for $\mathcal{PA}_s$

5.4.1 Type System

The purpose of the type system below is to efficiently type processes so that the 'illegal' runtime behaviours of processes are prevented by static type checking. The type system is based on the role projection developed earlier.

We define the following syntax:

$$\Gamma ::= \varnothing \mid \Gamma, a \triangleright B \mid \Gamma, X \qquad \Delta ::= \{\tilde{c}_i : Q_i\}_{i\in I}$$

where $\tilde{c}_i \cap \tilde{c}_j = \varnothing$ for each $i \neq j \in I$.

$$\Gamma \vdash \mathbf{0} \triangleright \mathbf{0} \qquad \text{[T-NIL]}$$

$$\Gamma, a \triangleright B \vdash a \triangleright B \qquad \text{[T-CH]}$$

$$\Gamma, X \vdash X \triangleright X \text{ or } \Gamma, X \vdash \mathbf{0} \circ \tilde{c} : X \qquad \text{[T-VAR]}$$

$$\frac{\Gamma \vdash a \triangleright B \quad \Gamma \vdash P \triangleright R \circ \tilde{c} : B\!\upharpoonright\!1\langle\tilde{c}\rangle, \Delta \quad \mathrm{pid}(B) = [1, n]}{\Gamma \vdash \bar{a}_{[2..n]}(\tilde{c}).P \triangleright \bar{a}_{[2..n]}(\tilde{c}).R \circ \Delta} \qquad \text{[T-INV]}$$

$$\frac{\Gamma \vdash a \triangleright B \quad \Gamma \vdash P \triangleright R \circ \tilde{c} : B\!\upharpoonright\!i\langle\tilde{c}\rangle, \Delta \quad 2 \leq i \in \mathrm{pid}(B)}{\Gamma \vdash a_{[i]}(\tilde{c}).P \triangleright a_{[i]}(\tilde{c}).R \circ \Delta} \qquad \text{[T-ACC]}$$

$$\frac{\Gamma \vdash P \triangleright R \circ \Delta \quad \tilde{c} \cap (\mathrm{dom}(\Gamma) \cup \mathrm{ch}(\Delta)) = \varnothing}{\Gamma \vdash P \triangleright R \circ \tilde{c} : \mathbf{0}, \Delta} \qquad \text{[T-TML]}$$

$$\frac{\Gamma \vdash P \triangleright R \circ \Delta \quad \tilde{c} \cap (\mathrm{dom}(\Gamma) \cup \mathrm{ch}(\Delta)) = \varnothing}{\Gamma \vdash P \triangleright R \circ \Delta, \tilde{c} : \mathbf{0}} \qquad \text{[T-TMR]}$$

$$\frac{\Gamma \vdash P \triangleright R \circ \tilde{c} : Q, \Delta \quad b \in \tilde{c}}{\Gamma \vdash b\S v.P \triangleright R \circ \tilde{c} : b\S v.Q, \Delta} \text{ where } \S \in \{!, ?\} \qquad \text{[T-SR]}$$

$$\frac{\Gamma \vdash P \triangleright R \circ \Delta \quad \Gamma \vdash P' \triangleright R' \circ \Delta' \quad \Delta \asymp \Delta'}{\Gamma \vdash P \mid P' \triangleright R \mid R' \circ \Delta \mid \Delta'} \qquad \text{[T-COM]}$$

$$\frac{\Gamma \vdash P \triangleright R \circ \Delta \quad \Gamma \vdash P' \triangleright R' \circ \Delta' \quad \Delta \asymp \Delta'}{\Gamma \vdash P + P' \triangleright R \sqcup R' \circ \Delta \sqcup \Delta'} \qquad \text{[T-SUM]}$$

$$\frac{\Gamma, X \vdash P \triangleright R \circ \tilde{c}_1 : Q_1, \ldots, \tilde{c}_n : Q_n}{\Gamma \vdash [\mathrm{rec}\, X]P \triangleright R^{[X]} \circ \tilde{c}_1 : Q_1^{[X]}, \ldots, \tilde{c}_n : Q_n^{[X]}} \qquad \text{[T-REC]}$$

$$\frac{\Gamma \vdash P \triangleright R \circ \Delta \quad R \equiv R' \quad \Delta \equiv \Delta'}{\Gamma \vdash P \triangleright R' \circ \Delta'} \qquad \text{[T-EQ]}$$

$$\frac{\Gamma \vdash P \triangleright \Delta, R \circ \tilde{c} : Q, \Delta' \quad b \in \tilde{c}}{\Gamma \vdash (\nu b)P \triangleright R \circ \Delta, \tilde{c}\backslash b : Q, \Delta'} \qquad \text{[T-HID]}$$

$$\frac{\Gamma \vdash P \triangleright R \circ \Delta \quad b \notin \mathrm{dom}(\Gamma) \cup \mathrm{ch}(\Delta)}{\Gamma \vdash (\nu b)P \triangleright R \circ \Delta} \qquad \text{[T-VEI]}$$

$$\frac{\Gamma \vdash P \triangleright R \circ \Delta}{\Gamma \vdash l : P \triangleright R \circ \Delta} \qquad \text{[T-LAB]}$$

Figure 5.7: Typing rules

A *type environment* Γ is a function that assigns sessions to some channels (session channels) and *typing* to session variables. A typing is of the form $R \circ \Delta$, where R, called a *session typing*, is projected from an integrating session, and Δ, called a *channel typing*, is a sequence of processes, each of which is labelled by a sequence of channels. We further require that the labelled sequences of channels for any two processes in a channel typing are pair-wise disjoint. The domain of Γ is a set of channels or variables it acts on. If its domain contains channel names only, we say Γ is *pure*. Re-ordering of items in a type environment Γ is permitted, but forbidden in a channel typing Δ.

The *type judgement* $\Gamma \vdash P \triangleright R \circ \Delta$ reads 'the typing of P is $R \circ \Delta$ under Γ.' If $\Delta = \epsilon$, we write $\Gamma \vdash P \triangleright R$. Formally, type judgements are defined by the typing rules in Figures 5.7, a short explanation of which is given below. We also say P is *typed* or *typable* by Γ if there is a typing of P under Γ. A few auxiliary definitions are given. $|\Delta|$ is the length of Δ and $\Delta[i]$ is the ith item of Δ where $1 \leq i \leq |\Delta|$. Δ and Δ' are *compatible*, denoted $\Delta \asymp \Delta'$, if $|\Delta| = |\Delta'|$ and $\Delta[i]$ and $\Delta'[i]$ have the same labelling sequence of channels for each $1 \leq i \leq |\Delta|$. Let $\Delta = \tilde{c}_1 : Q_1, \ldots, \tilde{c}_n : Q_n$ and $\Delta' = \tilde{c}_1 : Q'_1, \ldots, \tilde{c}_n : Q'_n$. $\Delta \equiv \Delta'$ if $Q_i \equiv Q'_i$ for each $1 \leq i \leq n$. Let $\Delta \mid \Delta' = \tilde{c}_1 : Q_1 \mid Q'_1, \ldots, \tilde{c}_n : Q_n \mid Q'_n$ and $\Delta \sqcup \Delta' = \tilde{c}_1 : Q_1 \sqcup Q'_1, \ldots, \tilde{c}_n : Q_n \sqcup Q'_n$. $\mathrm{ch}(\Delta)$ is the set of all labelling channels in Δ.

[T-INV] and [T-ACC] are for session invitation and acceptance, and [T-SR] for the ordinary communication. [T-TML] and [T-TMR] are needed because $\Delta \asymp \Delta'$ is used in the pre-conditions of [T-COM] and [T-SUM]. By [T-VAR], the type variables happen in either the main (i.e. integrating) session or a single communicating session. [T-REC] handles the recursive construction where $P^{[X]}$ is defined in Section 5.2. [T-EQ] is necessary to make the current type system expressive enough but also bring in the infinity of typing for processes. For restricted processes e.g. $(\nu a)P$, if $a \in \mathrm{fv}(P)$, then it is dealt with by [T-HID]; otherwise, by [T-VEI]. [T-LAB] absorbs the labelling in the typing derivation. [T-NIL] are standard.

We construct a (pure) type environment for the five agents in the business protocol and establish type judgements for them.

Example 5.4.1. *Let* $\Gamma_{\mathrm{prt}} = \mathrm{auc} \triangleright \mathit{Auction}$, $\mathrm{dTran} \triangleright \mathit{DTran}$, $\mathrm{sTran} \triangleright \mathit{STran}$,

epay $\triangleright$ *EPay. We have that* $\Gamma_{\text{prt}} \vdash P_{\text{broker}} \triangleright R^{\text{all}}_{\text{broker}}$, $\Gamma_{\text{prt}} \vdash P_{\text{buyer}_i} \triangleright R^{\text{all}}_{\text{buyer}_i}$, $\Gamma_{\text{prt}} \vdash P_{\text{seller}} \triangleright R^{\text{all}}_{\text{seller}}$, *and* $\Gamma_{\text{prt}} \vdash P_{\text{bank}} \triangleright R^{\text{all}}_{\text{bank}}$.

5.4.2 Properties of Typing

We study several key properties of the type system. The decidability of type inference comes first.

Theorem 5.4.2. *Given a process P and a type environment Γ, it is decidable whether there exist R, Δ such that $\Gamma \vdash P \triangleright R \circ \Delta$. If there exist, then there is an algorithm to construct such a pair.*

The proof of Theorem 5.4.2 is by computing a so-called *principal* typing for a given process under some type environment. A principal typing is a particular typing for a process such that the process has the principal typing if and only if it is typable. A standard type checking algorithm can be constructed to (attempt to) compute the principal typing for each process, and the termination of the algorithm is guaranteed by the decidability of the structural congruence for processes (cf. Lemma 5.2.1).

To present the following three properties of the type system, we put forward an auxiliary definition: for a channel typing $\Delta = \tilde{c}_1 : Q_1, \ldots, \tilde{c}_n : Q_n$, let $\lceil \Delta \rceil$ be the multiple parallel-composition process $Q_1 \mid \ldots \mid Q_n$. In general, $\Delta \equiv \Delta'$ is strictly stronger than $\lceil \Delta \rceil \equiv \lceil \Delta' \rceil$.

The Subject Congruence Theorem below implies that if P is typable and $P \equiv Q$ then Q is also typable and their typing have a certain structural relation.

Theorem 5.4.3 (Subject congruence)**.** *If $\Gamma \vdash P \triangleright R \circ \Delta$ and $P \equiv P'$, then there exist R', Δ' such that $\Gamma \vdash P' \triangleright R' \circ \Delta'$, $R \equiv R'$ and $\lceil \Delta \rceil \equiv \lceil \Delta' \rceil$.*

The Subjection Reduction Theorem states that the typability of a process is preserved in an 'expected' way during its evolvement. The theorem rules out the standard type errors. For example, there is no Γ such that $\Gamma \vdash \bar{a}_{[2..n]}(\tilde{c}).P_1 \mid a_{[n+1]}(\tilde{c}).P_2 \mid P_3$ or $\Gamma \vdash a_{[k]}(\tilde{c}).Q_1 \mid a_{[l]}(\tilde{c}').Q_2$ where $|\tilde{c}| \neq |\tilde{c}'|$.

Theorem 5.4.4 (Subject reduction)**.** *If $\Gamma \vdash P \triangleright R \circ \Delta$ and $P \xrightarrow{\alpha} P'$, then $\Gamma \vdash P' \triangleright R' \circ \Delta'$ for some R', Δ' satisfying the following conditions:*

1. *If* $\alpha = a\S v$ *where* $\S \in \{!, ?\}$ *then* $R \succ R'$ *and* $\lceil\Delta\rceil \xrightarrow{a\S v}_{\succ} \lceil\Delta'\rceil$,
2. *If* $\alpha = \bar{a}_{[2..n]}(\tilde{c})$ *and* $\Gamma \vdash a \triangleright B$ *then* $R \xrightarrow{\bar{a}_{[2..n]}(\tilde{c})}_{\succ} R'$ *and* $B\restriction 1\langle\tilde{c}\rangle \mid \lceil\Delta\rceil \succ \lceil\Delta'\rceil$,
3. *If* $\alpha = a_{[k]}(\tilde{c})$ *and* $\Gamma \vdash a \triangleright B$ *then* $R \xrightarrow{a_{[k]}(\tilde{c})}_{\succ} R'$ *and* $B\restriction k\langle\tilde{c}\rangle \mid \lceil\Delta\rceil \succ \lceil\Delta'\rceil$,
4. *If* $\alpha = \tau$ *then*
   (a) *either* $R \succ R'$ *and* $\lceil\Delta\rceil \xrightarrow{\tau}_{\succ} \lceil\Delta'\rceil$,
   (b) *or* $R \xrightarrow{\tau}_{\succ} R'$ *and* $B\restriction 1\langle\tilde{c}\rangle \mid \ldots \mid B\restriction n\langle\tilde{c}\rangle \mid \lceil\Delta\rceil \succ \lceil\Delta'\rceil$ *for some* B, n *such that* $\mathrm{pid}(B) = [1, n]$.

The last property says that the typing of a process under a type environment is unique up to a certain structural relation as in Theorem 5.4.3.

Theorem 5.4.5 (Typing uniqueness)**.** *If* $\Gamma \vdash P \triangleright R \circ \Delta$ *and* $\Gamma \vdash P \triangleright R' \circ \Delta'$, *then* $R \equiv R'$ *and* $\lceil\Delta\rceil \equiv \lceil\Delta'\rceil$.

5.5 Behavioural Analysis

In this section, we use the two-level session types to analyse interactions of distributed program components. Two system properties are dealt with: a channel safety property (also studied in the existing session type literature) and a behavioural conformance between processes and sessions. But we first show how to represent and properly type a distributed system in our formalism.

Informally, a program or a component in a distributed system is a pair of a participant name and a process. Following works in the process algebraic approach to architectural analysis, such as Allen and Garlan [1997]; Bernardo et al. [2002]; Su et al. [2012a], we define (the architecture of) a system as a parallel composition of programs or components. Formally, we define that

Definition 5.5.1 (Architecture)**.** *A program or a component is a labelled process* $r : P$, *where* r *and* P *specify its name and behaviour, respectively. An architecture is a process of the form* $\mathrm{Arch} = r_1 : P_1 \mid \ldots \mid r_n : P_n$.

For example, the following architecture implements the business protocol introduced in the Introduction and formalised in Section 5.3.2.

$$\begin{aligned}\mathrm{Arch}_e \;=\; & \mathrm{broker} : P_{\mathrm{broker}} \mid \mathrm{buyer}_1 : P_{\mathrm{buyer}_1} \mid \\ & \mathrm{buyer}_2 : P_{\mathrm{buyer}_2} \mid \mathrm{seller} : P_{\mathrm{seller}} \mid \mathrm{bank} : P_{\mathrm{bank}}\end{aligned}$$

A session channel a *marks* B in A if *some* occurrence of B in A is marked by a in the projection. The following definition characterises how to use session types to properly type a system.

Definition 5.5.2 (Session well-typedness)**.** Arch *(as defined in Def. 5.5.1) is well-typed by* A_{spc} *under* Γ *if*

- A_{spc} *is singled-threaded,*
- $\mathrm{pid}(A_{\mathrm{spc}}) = (r_1, \ldots, r_n)$*,*
- $\Gamma \vdash a \triangleright B$ *if and only if* a *marks* B *in* A_{spc}*, and*
- $\Gamma \vdash P_i \triangleright A_{\mathrm{spc}}\restriction r_i$ *for each* $1 \leq i \leq n$*.*

If Arch is well-typed by A_{spc}, we call A_{spc} a session for Arch. In general, well-typedness is strictly stronger than typability. In other words, if Arch is well-typed by A_{spc} under Γ then $\Gamma \vdash \mathrm{Arch} \triangleright A_{\mathrm{spc}}$; but the other direction does not necessarily hold. Also, we observe that if Arch is well-typed by some session and Arch (as a process) is closed then Arch is well-typed by some pure session.

The channel safety property below says that channel interference is prevented in the runtime of the system.

Definition 5.5.3 (Channel privacy)**.** *The communicating channels in* Arch *are private if the following holds: if* $\mathrm{Arch} \xrightarrow{\tau}_* (\nu \tilde{b})(r_1 : P_1 \mid \ldots \mid r_n : P_n)$ *and* $c\S v \in \mathrm{act}(P_i)$ *where* $\S \in \{!, ?\}$*, then their exists a unique* r_j *such that* $r_i \neq r_j$ *and* $c \in \mathrm{fc}(P_j)$*.*

The channel privacy of a system is a consequence of well-typedness by a session specification, as the following theorem demonstrates.

Theorem 5.5.4. *If* Arch *is well-typed by* A *under* Γ*, then the communicating channels in* Arch *are private.*

Informally, the theorem is guaranteed by the determinism of message flow in the projected roles and the creation of fresh channels in the session establishment.

Session conformance says that the runtime interactions of the system conform to its session specification.

Definition 5.5.5 (Session conformance)**.** *P conforms to S, if there is a relation $\mathcal{R}$ of processes and sessions such that if $\langle P, S\rangle \in \mathcal{R}$ then the following conditions hold:*

- *If $P \equiv (\nu\tilde{b})(p : Q_1 \mid q : Q_2 \mid R)$, $Q_1 \xrightarrow{a!v} Q_1'$ and $Q_2 \xrightarrow{a?v} Q_2'$, then there exists S' such that $S \xrightarrow{p,q:v} S'$ and $\langle P', S'\rangle \in \mathcal{R}$ where $P' \equiv (\nu\tilde{b})(p : Q_1' \mid q : Q_2' \mid R)$;*
- *If $P \equiv (\nu\tilde{b})(p_1 : Q_1 \mid \ldots \mid p_m : Q_m \mid R)$, $Q_1 \xrightarrow{\bar{a}_{[2..m]}(\tilde{c})} Q_1'$ and $Q_i \xrightarrow{a_{[i]}(\tilde{c})} Q_i'$ for all $2 \leq i \leq m$, then there exist B, S' such that $\Gamma \vdash a \triangleright B$, $S \xrightarrow{p_1,\ldots,p_m:B} S'$ and $\langle P', S'\rangle \in \mathcal{R}$, where $P' \equiv (\nu\tilde{b})((\nu\tilde{c})(p_1 : Q_1' \mid \ldots \mid p_m : Q_m') \mid R)$.*

An alternative explanation of session conformance is behavioural refinement, because Def. 5.5.5 actually defines a behavioural simulation relation between sessions and processes. We observe that if P conforms to S and $P \equiv P'$, then P' conforms to S. The following theorem confirms that session conformance of the system is also a consequence of well-typedness by a session specification.

Theorem 5.5.6. *If* Arch *is well-typed by* A_{spc} *then* Arch *conforms to* A_{spc}*.*

By what we have established so far, we have the following two properties for the system Arch_e:

Example 5.5.7. *(1) The communicating channels are private in* Arch_e*. (2) The behaviour of* Arch_e *conforms to its session specification Proto.*

5.6 Process Slicing

A type inference algorithm computes a typing, if possible, for a process under a type environment (cf. Theorem 5.4.2). However, in the real-life cases, the developers have the session specification in the first place and then implement it, so they

need to check whether a process is typable by the given session specification. A straightforward method to solve this type checking problem consists of two steps: to verify whether $\Gamma \vdash P \triangleright A\upharpoonright r$ for some $r : P, A, \Gamma$, we first compute $\Gamma \vdash P \triangleright A_P\upharpoonright r$ by a type inference algorithm and then check whether $A_P \equiv A$. This algorithm is efficient, pre-supposing we have an efficient type inference algorithm.

However, there is a drawback in the above algorithm: if $\Gamma \vdash P \triangleright A\upharpoonright r$ does not hold, the algorithm does not tell which session or sessions it violates. Since the session specification is modularised, it is desirable to know the violated session or sessions. In the following, we propose an algorithm based on process slicing to improve the type checking. Informally, the key idea of the algorithm is to decompose a process into parts and compare each part with a role projected from a corresponding session.

Suppose each session channel in P is typed by Γ, namely, contained in the domain of Γ. The algorithm consists of two steps. The first step is the process slicing. Because the hiding and labelling operators are unnecessary for processes as the initial (not runtime) behaviours of programs or components, we assume that P is free of these two operators. We call P^{χ_M} the main slice of P and the main slicing function χ_M is formally defined as follows.

$$\begin{aligned}
\mathbf{0}^{\chi_M} &= \mathbf{0} \quad X^{\chi_M} = X \quad ([\mathrm{rec}\, X]P)^{\chi_M} = [\mathrm{rec}\, X](P)^{\chi_M}, \\
(\pi.P)^{\chi_M} &= \begin{cases} \pi.(P)^{\chi_M} & \text{if } \pi \text{ is a session action,} \\ P^{\chi_M} & \text{otherwise;} \end{cases} \\
(P \star Q)^{\chi_M} &= P^{\chi_M} \star Q^{\chi_M} \quad \text{where } \star \text{ is } \mid \text{ or } +.
\end{aligned}$$

We call $P^{\chi_{\tilde{c}}}$ the $\tilde{c}$-slice of P, and the slicing function $\chi_{\chi_{\tilde{c}}}$, which are parametric on $\tilde{c}$, is defined below.

$$\begin{aligned}
\mathbf{0}^{\chi_{\tilde{c}}} &= \mathbf{0} \quad X^{\chi_{\tilde{c}}} = X \quad ([\mathrm{rec}\, X]P)^{\chi_{\tilde{c}}} = [\mathrm{rec}\, X](P)^{\chi_{\tilde{c}}}, \\
(\pi.P)^{\chi_{\tilde{c}}} &= \begin{cases} P^{\chi_{\tilde{c}}} & \text{if } \mathrm{fc}(\pi) \notin \tilde{c}, \\ \pi.(P)^{\chi_{\tilde{c}}} & \text{otherwise;} \end{cases} \\
(P \star Q)^{\chi_{\tilde{c}}} &= P^{\chi_{\tilde{c}}} \star Q^{\chi_{\tilde{c}}} \quad \text{where } \star \text{ is } \mid \text{ or } +.
\end{aligned}$$

After computing the slices of a process, we check whether each slice is struc-

turally congruent to a corresponding role. Specifically, we verify if $A\upharpoonright r \equiv (\pi.P)^{\chi_M}$, $B\upharpoonright 1\langle\tilde{c}\rangle \equiv P^{\chi_{\tilde{c}}}$, and $B\upharpoonright k\langle\tilde{c}\rangle \equiv P^{\chi_{\tilde{c}}}$, where $\Gamma \vdash a \triangleright B$.

By by our bound name convention, a name is not bound twice and does not have free *and* bound occurrences simultaneously in a process. The following theorem says that if P is typed by $A\upharpoonright r$ under Γ then the slicing of P 'coincides' with the role projection of A.

Theorem 5.6.1 (Slicing-projection correspondence)**.** *If* $\Gamma \vdash P \triangleright A\upharpoonright r\langle\tilde{a}\rangle$ *then the following three conditions hold:*

- $A\upharpoonright r\langle\tilde{a}\rangle \equiv P^{\chi_M}$,
- *if* $\bar{a}_{[2..n]}(\tilde{c}) \in \mathrm{act}(P)$ *and* $\Gamma \vdash a \triangleright B$, *then* $B\upharpoonright 1\langle\tilde{c}\rangle \equiv P^{\chi_{\tilde{c}}}$,
- *if* $a_{[k]}(\tilde{c}) \in \mathrm{act}(P)$ *and* $\Gamma \vdash a \triangleright B$, *then* $B\upharpoonright k\langle\tilde{c}\rangle \equiv P^{\chi_{\tilde{c}}}$.

Based on the above theorem, the correctness of the process slicing algorithm for the type checking is established. However, the method is not complete: the other direction of the theorem does not hold, as witnessed by the following counter-example. Thereby, the coincidence of role projection and process slicing does not entail the typability, and a technical implication is that the process slicing method cannot replace the type system in Section 5.4.1.

Example 5.6.2. *Let* $B_1 = \langle p, q : v_1\rangle \to \langle p, q : u_1\rangle \to \mathtt{end}$, $B_2 = \langle q, p : v_2\rangle \to \langle q, p : u_2\rangle \to \mathtt{end}$, $A_0 = B_1; B_2$, $\Gamma_0 \vdash a_i \triangleright B_i$ *where* $i \in \{1, 2\}$, *and* $P_1 = a^1_{[2]}(c_1).\ c_1?v_1.\, a^2_{[2..2]}(c_2).\, c_2!v_2.\, c_1?u_1.\, c_2!u_2.\, \mathtt{end}$. *With a suitable role projection of* B_1 *and* B_2, *we have that* A_0, P_1 *and* Γ_0 *satisfy the three conditions in Theorem 5.6.1 but not* $\Gamma_0 \vdash P_1 \triangleright A_0\upharpoonright p\langle a_1, a_2\rangle$.

5.7 Examples and Proof Details

5.7.1 Complete Set of Roles for The Protocol Example

Roles of Proto for the five agents Let $j \in \{1,2\}$.

$$R^{\text{all}}_{\text{broker}} \stackrel{\text{def}}{=} \overline{\text{auc}}_{[2..3]}(a_{1,2}, a_{1,3}).\ \sum_{i\in\{1,2\}} (\text{dTran}^{i}_{[2]}(b_{1,3}, b_{2,3}).\ \mathbf{0} + \text{sTran}^{i}_{[2]}(c_{1,2}, c_{1,3}, c_{2,3}).\ \text{epay}^{i}_{[2]}(d_{1,3}, d_{2,3}).\ \mathbf{0})$$

$$R^{\text{all}}_{\text{buyer}_j} \stackrel{\text{def}}{=} \text{auc}_{[j+1]}(a_{1,2}, a_{1,3}).\ (\overline{\text{dTran}}^{j}_{[2..3]}(b_{1,3}, b_{2,3}).\ \mathbf{0} + \overline{\text{sTran}}^{j}_{[2..3]}(c_{1,2}, c_{1,3}, c_{2,3}).\ \overline{\text{epay}}^{j}_{[2..3]}(d_{1,3}, d_{2,3}).\ \mathbf{0})$$

$$R^{\text{all}}_{\text{seller}} \stackrel{\text{def}}{=} \sum_{i\in\{1,2\}} (\text{dTran}^{i}_{[3]}(b_{1,3}, b_{2,3}).\ \mathbf{0} + \text{sTran}^{j}_{[3]}(c_{1,2}, c_{1,3}, c_{2,3}).\ \mathbf{0})\ \overline{\text{sTran}}^{i-1}_{[2..3]}(c_{1,2}, c_{1,3}, c_{2,3}).\ \overline{\text{epay}}^{i-1}_{[2..3]}(d_{1,3}, d_{2,3}).\ \mathbf{0})$$

$$R^{\text{all}}_{\text{bank}} \stackrel{\text{def}}{=} \sum_{i\in\{1,2\}} \text{epay}^{i}_{[3]}(d_{1,3}, d_{2,3}).\ \mathbf{0}$$

Roles of the four communicating sessions for the broker

$$R^{\text{auc}}_{\text{broker}} \stackrel{\text{def}}{=} \sum_{i\in\{1,2\}} (a_{1,i+1}?\text{bid}.\ [\text{rec}\, X_1](a_{1,4-i}!\text{quote}.\ (a_{1,i+1}!\text{invoice}.\ \mathbf{0}.\ + a_{1,4-i}?\text{bid}.\ a_{1,i+1}!\text{quote}.\ (a_{1,i+1}?\text{bid}.\ X_1 + a_{1,4-i}!\text{invoice}.\ \mathbf{0}))))$$

$$R^{\text{sTran}}_{\text{broker}} \stackrel{\text{def}}{=} c_{2,3}!\text{prepaid}.\ c_{1,2}?\text{confirm}.\ c_{2,3}!\text{payment}.\ \mathbf{0}$$

$$R^{\text{dTran}}_{\text{broker}} \stackrel{\text{def}}{=} b_{2,3}!\text{price}.\ \mathbf{0} \qquad R^{\text{epay}}_{\text{broker}} \stackrel{\text{def}}{=} d_{2,3}?\text{transfer}.\ \mathbf{0}$$

Roles of the four communicating sessions for the buyers Let $j \in \{1,2\}$.

$$R^{\text{auc}}_{\text{buyer}_j} \stackrel{\text{def}}{=} a_{1,j+1}!\text{bid}.\ [\text{rec}\, X_1](a_{1,j+1}?\text{quote}.\ a_{1,j+1}!\text{bid}.\ X + a_{1,j+1}?\text{invoice}.\ \mathbf{0}) + a_{1,j+1}?\text{quote}.\ [\text{rec}\, X_2](a_{1,j+1}!\text{bid}.\ (a_{1,j+1}?\text{invoice}.\ \mathbf{0} + a_{1,j+1}?\text{quote}.\ X_2))$$

$$R^{\text{dTran}}_{\text{buyer}_j} \stackrel{\text{def}}{=} b_{1,3}!\text{payment}.\ b_{1,3}?\text{order}.\ \mathbf{0}$$

$$R^{\text{sTran}}_{\text{buyer}_j} \stackrel{\text{def}}{=} c_{1,3}?\text{order}.\ c_{1,2}!\text{confirm}.\ \mathbf{0} \qquad R^{\text{epay}}_{\text{buyer}_j} \stackrel{\text{def}}{=} d_{1,3}!\text{amount}.\ \mathbf{0}$$

Roles of DTransaction and DTransaction for the seller

$$R^{\text{dTran}}_{\text{seller}} \stackrel{\text{def}}{=} b_{2,3}?\text{price}.\ b_{1,3}?\text{payment}.\ b_{1,3}!\text{order}.\ \mathbf{0}$$
$$R^{\text{sTran}}_{\text{seller}} \stackrel{\text{def}}{=} c_{2,3}?\text{prepaid}.\ c_{1,3}!\text{order}.\ c_{1,3}!\text{payment}.\ \mathbf{0}$$

The role of EPay for the bank

$$R^{\text{epay}}_{\text{bank}} \stackrel{\text{def}}{=} d_{1,3}?\text{amount}.\ d_{2,3}!\text{transfer}.\ \mathbf{0}$$

5.7.2 Supplemented Examples

The following presents two more examples to show the utility of our two-level session types in expressing the scenario-based specification of systems.

Client-server system The first one is a client-server system, which consists of one client, two servers and a configurator. The client attempts to make requests to the servers and the configurator co-ordinates the client and two servers such that the client can only call the available servers. Both servers have two states: they are either in the normal working order or preparing to their data bases and shutting down the service temporarily. The servers inform the configurator of their states in the session in their conversations. Before the client calls the servers, the configurator tells them whether the servers are ready to take requests.

The following is a formulation of the session specification in our two-level session types, where C is the client, S_1, S_2 are two servers, F is the configurator, *CSsystem* is an integrating session type, and *Control*, *Initi* and *Service* are

communicating session types.

$$
\begin{aligned}
\mathit{CSsystem} &\stackrel{\text{def}}{=} \langle F, S_1, S_2 : \mathit{Control}\rangle\{\} \\
&\qquad \otimes \mu\mathbf{t}.(\bigoplus_{i\in\{1,2\}} \langle C, F : \mathit{Initi}\rangle\{\}; \langle C, S_i : \mathit{Service}\rangle\{\}; \mathbf{t}) \\
\mathit{Control} &\stackrel{\text{def}}{=} \mu\mathbf{t}.(\bigoplus_{j\in\{2,3\}} \langle j, 1 : \text{update}\rangle \to \langle j, 1 : \text{ready}\rangle \to \mathbf{t}) \\
\mathit{Initi} &\stackrel{\text{def}}{=} \bigoplus_{k\in\{1,2\}} \langle 1, 2 : \text{ping}_k\rangle \to (\langle 1, 2 : \text{yes} \to \texttt{end}\rangle \oplus \langle 1, 2 : \text{no}\rangle \to \texttt{end}) \\
\mathit{Service} &\stackrel{\text{def}}{=} \langle 1, 2 : \text{request}\rangle \to \langle 2, 1 : \text{return}\rangle \to \texttt{end}
\end{aligned}
$$

The formulate captures the intuitive and coarse-grained understanding of the conversations between the four components. First, the conversations consists of three parts, recorded by three communicating sessions. Second, the relationship of these sessions is described by *CSsystem*, revealing the most essential design decisions of the system. For example, *Service* happens after *Initi* and together they form a recursive session. *Control* is also recursive and proceeds independent of the other two communicating sessions. Of course, many design details are to be worked out in the later development stage. For example, if the configurator replies 'no' to the client's pinging action in *Initi* , then the client may not initiate *Service*. Also, the messages received by the configurator in *Control* should affect its replies to the client's pinging action in *Init*.

Quote request The second example is a quote request protocol which is modified and simplified from the one in Yoshida et al. [2010]. The protocol involves three agents, i.e. a buyer, a supplier, and a manufacturer, and consists of two parts: the first part is conversation between the buyer and the supplier to negotiate the price of some item or good with; the second part, which is nested within in the first part, is between the supplier and the manufacturer to confirm the price.

As before, the formulate consists of one integrating session and several (here is two) communicating sessions. B stands the buyer, S the supplier, and M the

manufacturer.

$$\begin{aligned} \mathit{QuoteReq} &\stackrel{\text{def}}{=} \langle B, S : \mathit{Negotn}\rangle\{\langle S, M : \mathit{Confirm}\rangle\{\}\} \\ \mathit{Negotn} &\stackrel{\text{def}}{=} \langle 1,2 : \text{item}\rangle \to \langle 2,1 : \text{quote}\rangle \to (\langle 1,2 : \text{accepted}\rangle \to \texttt{end} \oplus \\ &\qquad \langle 1,2 : \text{newquote}\rangle \to (\langle 2,1 : \text{accepted}\rangle \to \texttt{end} \oplus \\ &\qquad\quad \langle 2,1 : \text{rejected}\rangle \to \texttt{end})) \\ \mathit{Confirm} &\stackrel{\text{def}}{=} \langle 1,2 : \text{quote}\rangle \to (\langle 1,2 : \text{yes} \to \texttt{end}\rangle \oplus \langle 1,2 : \text{no}\rangle \to \texttt{end}) \end{aligned}$$

This example shows the necessity to distinguish protocol calling and protocol nesting (c.f. discussions in Section **??**). As far as the protocol is concerned, it suffices to indicate the nesting relationship between *Negotn* and *Confirm*. Moreover, without specifying the nesting position of *Confirm* in *Negotn*, *Negotn* describe a complete conversation between the buyer and the supplier.

5.7.3 Derivation of $\Gamma_{\text{prt}} \vdash P_{\text{broker}} \triangleright R^{\text{all}}_{\text{broker}}$

This part of the appendix is dedicated to detailing a derivation of the type judgement $\Gamma_{\text{prt}} \vdash P_{\text{broker}} \triangleright R^{\text{all}}_{\text{broker}}$ in Proposition 5.4.1. Derivations of other type judgements in Proposition 5.4.1 can be constructed in a similar way.

1. by [T-NIL][T-TML]:

$$\Gamma_{\text{prt}} \vdash \mathbf{0} \triangleright \mathbf{0} \circ (c_{1,2}, c_{1,3}, c_{2,3}) : \mathbf{0}$$

2. by [T-SR]:

$$\Gamma_{\text{prt}} \vdash R^{\text{sTran}}_{\text{broker}} \triangleright \mathbf{0} \circ (c_{1,2}, c_{1,3}, c_{2,3}) : R^{\text{sTran}}_{\text{broker}}$$

3. by [T-TML][T-SR]:

$$\Gamma_{\text{prt}} \vdash d_{1,3}?\text{transfer}.\ R^{\text{sTran}}_{\text{broker}} \triangleright \mathbf{0} \circ (d_{1,3}, d_{2,3}) : R^{\text{epay}}_{\text{broker}}, (c_{1,2}, c_{1,3}, c_{2,3}) : R^{\text{sTran}}_{\text{broker}}$$

4. let $i \in \{1,2\}$ and

$$P_1^i \stackrel{\text{def}}{=} \text{sTran}^i_{[2]}(c_{1,2}, c_{1,3}, c_{2,3}).\ \text{epay}^i_{[2]}(d_{1,3}, d_{2,3}).\ d_{1,3}?\text{transfer}.\ R^{\text{sTran}}_{\text{broker}}$$

5. by [T-CH][T-ACC]:

$$\Gamma_{\mathrm{prt}} \vdash P_1^i \triangleright \mathrm{sTran}^i_{[2]}(c_{1,2}, c_{1,3}, c_{2,3}).\ \mathrm{epay}^i_{[2]}(d_{1,3}, d_{2,3}).\ \mathbf{0}$$

6. by [T-NIL][T-TML][T-SR][T-CH][T-ACC]:

$$\Gamma_{\mathrm{prt}} \vdash \mathrm{dTran}^i_{[2]}(b_{1,2}, b_{2,3}).\ b_{2,3}!\mathrm{price}.\ \mathbf{0}\ \triangleright \mathrm{dTran}^i_{[2]}(b_{1,2}, b_{2,3}).\ \mathbf{0}$$

7. let

$$P_2^i = \mathrm{dTran}^i_{[2]}(b_{1,2}, b_{2,3}).\ \mathbf{0} + \mathrm{sTran}^i_{[2]}(c_{1,2}, c_{1,3}, c_{2,3}).\ \mathrm{epay}^i_{[2]}(d_{1,3}, d_{2,3}).\ \mathbf{0}$$

8. by (5)(6)[T-SUM]:

$$\Gamma_{\mathrm{prt}} \vdash P^i_{\mathrm{broker}} \triangleright P_2^i \qquad \Gamma_{\mathrm{prt}} \vdash P^{3-i}_{\mathrm{broker}} \triangleright P_2^{3-i}$$

9. by [T-TML][T-SR]:

$$\Gamma_{\mathrm{prt}} \vdash a_{1,4-i}!\mathrm{invoice}.\ P^{3-i}_{\mathrm{broker}} \triangleright P_2^{3-i} : (a_{1,2}, a_{1,3}) : a_{1,4-i}!\mathrm{invoice}.\ \mathbf{0}$$

10. by [T-VAR][T-TML][T-SR]:

$$\Gamma_{\mathrm{prt}} \vdash a_{1,i+i}?\mathrm{bid}.\ X \triangleright \mathbf{0} \circ (a_{1,2}, a_{1,3}) : a_{1,i}?\mathrm{bid}.\ X$$

11. let

$$P_3^i = a_{1,4-i}?\mathrm{bid}.\ a_{1,i+1}!\mathrm{quote}.\ (a_{1,i+i}?\mathrm{bid}.\ X + a_{1,4-i}!\mathrm{invoice}.\ \mathbf{0})$$

12. by (9)(10)[T-SUM][T-SR][T-EQ]:

$$\Gamma_{\mathrm{prt}} \vdash P_3^i\{P^{3-i}_{\mathrm{broker}}/\mathbf{0}\} \triangleright P_2^{3-i} \circ (a_{1,2}, a_{1,3}) : P_3^i$$

13. by [T-TML][T-SR]:

$$\Gamma_{\text{prt}} \vdash a_{1,i+1}!\text{invoice}.\ P^i_{\text{broker}} \triangleright P^i_2 \circ (a_{1,2}, a_{1,3}) : a_{1,i+1}!\text{invoice}.\ \mathbf{0}$$

14. let

$$P^i_4 = a_{1,4-i}!\text{quote}.\ (a_{1,i+1}!\text{invoice}.\ P^i_{\text{broker}} + P^i_3\{P^{3-i}_{\text{broker}}/\mathbf{0}\})$$
$$P^i_5 = a_{1,4-i}!\text{quote}.\ (a_{1,i+1}!\text{invoice}.\ \mathbf{0} + P^i_3)$$

15. by (12)(13)[T-SUM][T-SR]:

$$\Gamma_{\text{prt}} \vdash P^i_4 \triangleright P^i_2 + P^{3-i}_2 \circ a_{1,4-i}!\text{quote}.\ (a_{1,i+1}!\text{invoice}.\ \mathbf{0} + P^i_3)$$

16. by [T-REC][T-SR]:

$$\Gamma_{\text{prt}} \vdash a_{1,i+i}?\text{bid}.\ [\text{rec}\, X]P^i_4 \triangleright P^i_2 + P^{3-i}_2 \circ a_{1,i+1}?\text{bid}.\ [\text{rec}\, X]P^i_5$$

17. by (16)(4)[T-SUM] ($P^1_2 + P^{3-1}_2 \equiv P^2_2 + P^{3-2}_2$):

$$\Gamma_{\text{prt}} \vdash \sum_{i\in\{1,2\}} P^i_4 \triangleright P^1_2 + P^2_2 \circ R^{\text{auc}}_{\text{broker}}$$

18. by [T-INV]:

$$\Gamma_{\text{prt}} \vdash P_{\text{broker}} \triangleright R^{\text{all}}_{\text{broker}}$$

This finishes the derivation.

5.7.4 Proof Details of Theorems

Proof of Theorem 5.4.2

Proof. The proof of is a standard proof of decidability of type inference. Because of [T-EQ], a process has infinite many typing, but we show that we can compute a 'principal' typing for each process such that the process has a 'principal' typing if and only if it is typable.

First, for each P, we compute a set $\mathrm{sub}_{|}(P)$ (resp. $\mathrm{sub}_{\sqcup}(P)$) which is the smallest set such that

1. if $\langle P_1, P_2\rangle \in \mathrm{sub}_{|}(P)$ (resp. $\mathrm{sub}_{\sqcup}(P)$) then $P_1 \mid P_2 \equiv P$ (resp. $P_1 \sqcup P_2 \equiv P$), and

2. if $Q_1 \mid Q_2 \equiv P$ (resp. $Q_1 \sqcup Q_2 \equiv P$) then there are P_1, P_2 such that $P_1 \equiv Q_1$, $P_2 \equiv Q_2$ and $\langle P_1, P_2\rangle \in \mathrm{sub}_{|}(P)$ (resp. $\mathrm{sub}_{\sqcup}(P)$).

The two sets are decidable because $\equiv$ is decidable (Lemma 5.2.1). $\mathrm{sub}_{|}(\Delta)$ is defined as follows: $\langle \Delta_1, \Delta_2\rangle \in \mathrm{sub}_{|}(\Delta)$ if and only if $|\Delta_1| = |\Delta_2| = |\Delta|$ and, for each $1 \leq i \leq |\Delta|$, $\langle P_1^i, P_2^i\rangle \in \mathrm{sub}_{|}(P^i)$ where $\Delta_1[i] = \tilde{c} : P_1^i$, $\Delta_2[i] = \tilde{c} : P_2^i$ and $P^i = \tilde{c} : \Delta[i]$ for some $\tilde{c}$. $\mathrm{sub}_{\sqcup}(\Delta)$ is defined similarly. Note that if $\langle \Delta_1, \Delta_2\rangle \in \mathrm{sub}_{|}(\Delta)$ (resp. $\mathrm{sub}_{\sqcup}(\Delta)$), then $\Delta_1 \asymp \Delta_2$.

A *principal* typing of P under Γ is a typing derived by the rules in Figures 5.7 except [T-EQ], and plus the following two rules:

$$\frac{\Gamma \vdash P \triangleright R \circ \Delta \quad \Gamma \vdash P' \triangleright R' \circ \Delta' \quad \langle R, R'\rangle \in \mathrm{sub}_{|}(R'') \quad \langle \Delta, \Delta'\rangle \in \mathrm{set}_{|}(\Delta'')}{\Gamma \vdash P \mid P' \triangleright R'' \circ \Delta''} \quad [\text{T-COM}+]$$

$$\frac{\Gamma \vdash P \triangleright R \circ \Delta \quad \Gamma \vdash P' \triangleright R' \circ \Delta' \quad \langle R, R'\rangle \in \mathrm{sub}_{\sqcup}(R'') \quad \langle \Delta, \Delta'\rangle \in \mathrm{set}_{\sqcup}(\Delta'')}{\Gamma \vdash P + P' \triangleright R'' \circ \Delta''} \quad [\text{T-SUM}+]$$

We have the following lemma:

Lemma 5.7.1. *$\Gamma \vdash P \triangleright R \circ \Delta$ if and only if P has a principal typing under Γ.*

The right-to-left direction of the lemma is obvious. For the other direction, we suppose $\Gamma \vdash P \triangleright R \circ \Delta$. In the derivative procedure, if commutative and associative laws for $\mid$ and $+$ are applied, we have the same derivation by the additional two derived rules, and whenever other structural laws in Figure 5.2 are applied, we just omit them. In this manner, we will obtain a principal typing for P. Therefore, the type inference of the type system is decidable. □

Note that if $R' \circ \Delta'$, say, is the principal typing of P, then by Theorem 5.4.5 (to be proved) $R \equiv R'$ and $\lceil \Delta \rceil = \lceil \Delta' \rceil$.

Proof of Theorem 5.4.3

Proof. Suppose $\Gamma \vdash P$ and $P \equiv P'$. The proof is by induction on the derivation of $P \equiv P'$. The proof is divided into two parts. First, we show that the each rule in Figure 5.2 and its symmetric form respect the above theorem. Here we detail one of the most tricky rules:

$$(\nu a)P_1 \mid P_2 \equiv (\nu a)(P_1 \mid P_2) \text{ if } a \notin \mathrm{fc}(P_2)$$

(1) We first suppose $P = (\nu a)P_1 \mid P_2$, $\Gamma \vdash P \triangleright R \circ \Delta$ and $a \notin \mathrm{fc}(P_2)$. Because $\Gamma \vdash P \triangleright R \circ \Delta$ is derived by [T-COM] and possibly by [T-EQ], [T-TML] and/or [T-TMR] (for one or more times), it can be verified that $\Gamma \vdash (\nu a)P_1 \triangleright R_1 \circ \Delta_1$, $\Gamma \vdash P_2 \triangleright R_2 \circ \Delta_2$, $\Delta_1 \asymp \Delta_2$, $R \equiv R_1 \mid R_2$, and $\lceil \Delta \rceil \equiv \lceil \Delta_1 \mid \Delta_2 \rceil$ for some $R_1, R_2, \Delta_1, \Delta_2$. Here we have two possibilities. (1.1) Suppose $a \in \mathrm{ch}(\Delta_1)$. Thus, $\Gamma \vdash (\nu a)P_1 \triangleright R_1 \circ \Delta_1$ is derived by [T-HID] and possibly by [T-EQ][T-TML][T-TMR], and, we have that $\Gamma \vdash P_1 \triangleright R_1 \circ \Delta_1', \tilde{c} : Q, \Delta_1''$, $a \in \tilde{c}$, and $\Delta_1 \equiv \tilde{c}_1 : \mathbf{0}, \ldots, \tilde{c}_m : \mathbf{0}, \Delta_1', \tilde{c}\backslash a : Q, \Delta_1'', \tilde{c}_{m+1} : \mathbf{0}, \ldots, \tilde{c}_{m+n} : \mathbf{0}$. No matter $a \in \bigcup_{i=1}^{m+n} \tilde{c}_i$ or not, we can rewrite the processes of type derivations for P_1 and P_2 to obtain type judgements $\Gamma \vdash P_1 \triangleright R_1 \circ \Delta_3$ and $\Gamma \vdash P_2 \triangleright R_2 \circ \Delta_2$ such that $\Delta_3 \asymp \Delta_4$ and $\lceil \Delta_3 \mid \Delta_4 \rceil = \lceil \Delta_1 \mid \Delta_2 \rceil$ (when applying [T-TML] or [T-TMR] to prefix $\tilde{b}$ for some $\tilde{b}$, we prefix $\tilde{b}/a$ or some $\tilde{b}'$ such that $\tilde{b}'\backslash a = \tilde{b}$ instead). Note that the rewritten derivations are based on $a \in \mathrm{fc}(P_2)$ and the channel assumption (cf. Section 5.2). Then, we apply [T-HID] to type $(\nu a)(P_1 \mid P_2)$ and obtain the desired result. (1.2) Suppose $a \notin \mathrm{ch}(\Delta_1)$. Thus, $\Gamma \vdash (\nu a)P_1 \triangleright R_1 \circ \Delta_1$ is derived by [T-VEI] and possibly by [T-EQ][T-TML][T-TMR], and we have that $\Gamma \vdash P_1 \triangleright R_1 \circ \Delta_1'$ and $\Delta_1 \equiv \tilde{c}_1 : \mathbf{0}, \ldots, \tilde{c}_m : \mathbf{0}, \Delta_1', \tilde{c}_{m+1} : \mathbf{0}, \ldots, \tilde{c}_{m+n} : \mathbf{0}$. Similarly, we rewrite the type derivation for P_2 and obtain $\Gamma \vdash P_2 \triangleright R_2 \circ \Delta_2'$ such that $\Delta_1 \asymp \Delta_2'$ and $\lceil \Delta_2' \rceil \equiv \lceil \Delta_2 \rceil$. Then, apply [T-VEI] to $\Gamma \vdash P_2 \triangleright R_2 \circ \Delta_2'$ and obtain the desired result. (2) Then, we suppose $P = (\nu a)(P_1 \mid P_2)$, $\Gamma \vdash P \triangleright R \circ \Delta$ and $a \notin \mathrm{fc}(P_2)$. The treatment is similar to the first case.

The second part of the proof is to show that the laws of congruence respect

the theorem. We choose to deal with the following rule:

$$\frac{P_1 \equiv P_2}{[\mathrm{rec}\, X]P_1 \equiv [\mathrm{rec}\, X]P_2}$$

We suppose $P = [\mathrm{rec}\, X]P_1$, $P' = [\mathrm{rec}\, X]P_2$, $P_1 \equiv P_2$, and $\Gamma \vdash P \triangleright R \circ \Delta$ where $\Delta \equiv \tilde{c}_1 : Q_1, \ldots, \tilde{c}_n : Q_n$. Because $\Gamma \vdash P \triangleright R \circ \Delta$ is derived by [T-REC] (and [T-EQ][T-TML][T-TMR], possibly), we have $\Gamma, X \vdash P_1 \triangleright R_1 \circ \tilde{c}_1 : Q'_1, \ldots, \tilde{c}_n : Q'_n$ such that $R_1 \equiv R^{[X]}$ and $Q'_i \equiv Q_i^{[X]}$ for each $1 \leq i \leq n$. Since $P_1 \equiv P_2$, by induction hypotheses, $\Gamma, X \vdash P_2 \triangleright R_2 \circ \tilde{c}'_1 : Q''_1, \ldots, \tilde{c}'_n : Q''_n$ for some R_2, $\tilde{c}'_i$ and Q''_i for each $1 \leq i \leq n$ such that $R_2 \equiv R_1$ and $Q'_1 \mid \ldots \mid Q'_n \equiv Q''_1 \mid \ldots Q''_n$. By [T-REC], we have that $\Gamma \vdash [\mathrm{rec}\, X]P_2 \triangleright R \circ \Delta'$ where $R \equiv R_2^{[X]}$ and $\Delta' \equiv \tilde{c}'_1 : Q''^{[X]}_1, \ldots \tilde{c}'_n : Q''^{[X]}_n$. Therefore, we have that $R \equiv R'$ and $\lceil \Delta \rceil \equiv \lceil \Delta' \rceil$. □

Proof of Theorem 5.4.4

Proof. The proof is by induction on the derivation of $P \xrightarrow{\alpha} P'$ according to rules in Figure 5.3 and depends on the value of α. The following only covers the most typical cases.

(1) Suppose $P = \alpha.P'$. In this case, α has three possible forms: $a\S v$, $\bar{a}_{[2..n]}(\tilde{c})$ or $a_{[k]}(\tilde{c})$ where $\S \in \{?, !\}$. First, we let $\alpha = a\S v$. Because $\Gamma \vdash P \triangleright R \circ \Delta$ is derived by [T-SR] and possibly by [T-EQ][T-TML][T-TMR] (for one or more times), we have that $\Gamma \vdash P' \triangleright R' \circ \Delta'$ for some R', Δ' such that $R \equiv R'$, $\lceil \Delta' \rceil \equiv \lceil \tilde{c} : Q, \Delta'' \rceil$, and $\lceil \Delta \rceil \equiv \lceil \tilde{c} : a\S v.Q, \Delta'' \rceil$ for some $\tilde{c}, Q, \Delta''$. We have that $\lceil \Delta \rceil \xrightarrow{\alpha}_{\succ} \lceil \Delta' \rceil$. Then, let $\alpha = \bar{a}_{[2..n]}(\tilde{c})$. Because $\Gamma \vdash P \triangleright R \circ \Delta$ is derived by [T-INV] (and possibly [T-EQ][T-TML][T-TMR]), we have that $\Gamma \vdash P' \triangleright R' \circ \Delta'$, $R \equiv \bar{a}_{[2..n]}(\tilde{c}).R'$ (thus $R' \xrightarrow{\alpha} R$) and $\lceil \Delta \rceil \equiv \lceil \Delta' \rceil \mid B \restriction 1\langle \tilde{c} \rangle$ where $\Gamma \vdash a \vdash B$. Lastly, let $\alpha = a_{[k]}(\tilde{c})$. Because $\Gamma \vdash P \triangleright R \circ \Delta$ is derived by [T-ACC] (and possibly [T-EQ][T-TML][T-TMR]), we have that $\Gamma \vdash P' \triangleright R' \circ \Delta'$, $R \equiv a_{[k]}(\tilde{c}).R'$ (thus $R' \xrightarrow{\alpha} R$), and $\lceil \Delta \rceil \equiv \lceil \Delta' \rceil \mid B \restriction 1\langle \tilde{c} \rangle$ where $\Gamma \vdash a \vdash B$ and $2 \leq k \in \mathrm{pid}(B)$.

(2) Let $\alpha = a\S v$ and suppose $P = P_1 + P_2$ and $P \xrightarrow{\alpha} P'$ is derived from $P_1 \xrightarrow{\alpha} P'$ (the treatment is similar of it derived from $P_2 \xrightarrow{\alpha} P'$). By [T-SUM] (and possibly by [T-EQ][T-TML][T-TMR]), we have that $\Gamma \vdash P_1 \triangleright R_1 \circ \Delta_1$, $\Gamma \vdash P_2 \triangleright R_2 \circ \Delta_2$, $R \equiv R_1 \sqcup R_2$, and $\lceil \Delta \rceil \equiv \lceil \Delta_1 \sqcup \Delta_2 \rceil$. By induction hypotheses, $\Gamma \vdash P' \triangleright R' \circ \Delta'$, $R_1 \succ R'$ and $\lceil \Delta_1 \rceil \xrightarrow{\alpha}_{\succ} \lceil \Delta' \rceil$. Hence, $R \succ R'$ and $\lceil \Delta \rceil \xrightarrow{\alpha}_{\succ} \lceil \Delta' \rceil$.

(3) Let $\alpha = \tau$ and $P = P_1 \mid P_2$. Here we have two subcases. (3.1) Suppose $P \xrightarrow{\tau} P'$ is derived from $P_1 \xrightarrow{\tau} P'$ (or $P_2 \xrightarrow{\alpha} P'$). The treatment for this subcase is relatively simple and similar to the last case and thus we omit it. (3.2) $P \xrightarrow{\tau} P'$ is derived from $P_1 \xrightarrow{a!v} P_1'$ and $P_2 \xrightarrow{a?v} P_2'$ and $P' = P_1' \mid P_2'$. Because $\Gamma \vdash P \triangleright R \circ \Delta$ is derived by [T-COM] (and possibly [T-EQ][T-TML][T-TMR]), we have that $\Gamma \vdash P_1 \triangleright R_1 \circ \Delta_1$ and $\Gamma \vdash P_2 \triangleright R_2 \circ \Delta_2$ for some $R_1, R_2, \Delta_1, \Delta_2$ such that $R \equiv R_1 \mid R_2$, and $\lceil \Delta \rceil \equiv \lceil \Delta_1 \mid \Delta_2 \rceil$. By induction hypotheses, $\Gamma \vdash P_1' \triangleright R_1' \circ \Delta_1'$ and $\Gamma \vdash P_2' \triangleright R_2' \circ \Delta_2'$ for some $R_1', R_2', \Delta_1', \Delta_2'$ such that $R_1 \succ R_1'$, $R_2 \succ R_2'$, $\lceil \Delta_1 \rceil \xrightarrow{a!v}_{\succ} \lceil \Delta_1' \rceil$ and $\lceil \Delta_2 \rceil \xrightarrow{a?v}_{\succ} \lceil \Delta_2' \rceil$. Also, $\Delta_1' \asymp \Delta_2'$. Hence, $\Gamma \vdash P' \triangleright R_1' \mid R_2' \circ \Delta_1' \mid \Delta_2'$, $R_1 \mid R_2 \succ R_1' \mid R_2'$ and $\lceil \Delta_1 \mid \Delta_2 \rceil \xrightarrow{\tau}_{\succ} \lceil \Delta_1' \mid \Delta_2' \rceil$. (3.3) $P \xrightarrow{\tau} P'$ is derived from $P_1 \xrightarrow{\bar{a}_{[2..n]}(\tilde{c})} P_2'$ and $P_i \xrightarrow{a_{[i]}(\tilde{c})} P_i'$ for each $2 \leq i \leq n$. Suppose $\Gamma \vdash a \triangle B$. Because $\Gamma \vdash P \triangleright R \circ \Delta$ is derived by [T-COM] for $n-1$ times (and possibly [T-EQ][T-TML][T-TMR]), we have that $\Gamma \vdash P_1 \triangleright R_1 \circ \Delta_1$ and $\Gamma \vdash P_i \triangle R_i \circ \Delta_i$ $(2 \leq i \leq n)$ for some $R_1, R_i, \Delta_1, \Delta_i$ such that $R \equiv R_1 \mid R_2 \mid \ldots \mid R_n$ and $\lceil \Delta \rceil \equiv \lceil \Delta_1 \rceil \mid \lceil \Delta_2 \rceil \mid \ldots \mid \lceil \Delta_n \rceil$. By induction hypotheses, $\Gamma \vdash P_1' \triangleright R_1' \circ \Delta_1'$ and $\Gamma \vdash P_i' \triangle R_i' \circ \Delta_i'$ $(2 \leq i \leq n)$ for some $R_1', R_i', \Delta_1', \Delta_i'$ such that $R_1 \xrightarrow{\bar{a}_{[2..n]}(\tilde{c})} R_1'$, $R_i \xrightarrow{a_{[i]}(\tilde{c})} R_i'$, $\lceil \Delta_1' \rceil \equiv \lceil \Delta_1 \rceil \mid B \restriction 1\langle \tilde{c} \rangle$, and $\lceil \Delta_i' \rceil \equiv \lceil \Delta_1 \rceil \mid B \restriction i\langle \tilde{c} \rangle$ for each $2 \leq i \leq n$. Also, $\Delta_1' \asymp \Delta_2' \asymp \ldots \asymp \Delta_n'$. Therefore, $\lceil \Delta_1' \mid \ldots \mid \Delta_n' \rceil \equiv B \restriction 1\langle \tilde{c} \rangle \mid \ldots \mid B \restriction n\langle \tilde{c} \rangle \mid \lceil \Delta_1 \mid \ldots \mid \Delta_n \rceil$.

(4) Let $\alpha = a \S v$. Suppose $Q \equiv P$, $Q' \equiv P'$, and $P \xrightarrow{\alpha} P'$ is derived from $Q \xrightarrow{\alpha} Q'$. By Theorem 5.4.3, $\Gamma \vdash Q \triangleright R_1 \circ \Delta_1$ such that $R_1 \equiv R$ and $\lceil \Delta_1 \rceil \equiv \lceil \Delta \rceil$. By induction hypotheses, $\Gamma \vdash Q' \triangleright R_1' \circ \Delta_1'$ such that $R_1 \succ R_1'$ and $\lceil \Delta_1 \rceil \xrightarrow{\alpha}_{\succ} \lceil \Delta_1' \rceil$. Then, by Theorem 5.4.3 again, we have the desired result. □

Proof of Theorem 5.4.5

Proof. The proof is by induction on the derivation of $\Gamma \vdash P \triangleright R \circ \Delta$ and $\Gamma \vdash P \triangleright R' \circ \Delta'$ according to rules in Figure 5.7. We detail two cases. (1) Suppose $P = \bar{a}_{[2..n]}(\tilde{c}).P_1$ and $\Gamma \vdash P \triangleright R \circ \Delta$ is derived by [T-INV]. Let $\Gamma \vdash a \triangleright B$ and $|\mathrm{pid}(B)| = n$. Thus, $\Gamma \vdash P_1 \triangleright R_1 \circ \Delta_1$ for some R_1, Δ_1 such that $R = \bar{a}_{[2..n]}(\tilde{c}).R_1$ and $\Delta_1 = \tilde{c} : B \restriction 1, \Delta$. Also, $\Gamma \vdash P \triangleright R' \circ \Delta'$ and possibly [T-EQ][T-TML][T-TMR] for one or more times. Thus, $\Gamma \vdash P_1 \triangleright R_1' \circ \Delta_1'$ for some R_1', Δ_1' such that $R' \equiv \bar{a}_{[2..n]}(\tilde{c}).R_1'$ and $\lceil \Delta_1' \rceil \equiv B \restriction 1 \mid \lceil \Delta' \rceil$. By induction hypotheses, $R_1 \equiv R_1'$ and

$\lceil \Delta_1 \rceil = \lceil \Delta_1' \rceil$. Therefore, $R \equiv R'$ and $\lceil \Delta \rceil = \lceil \Delta' \rceil$. (2) Suppose $P = [\mathrm{rec}\, X]P'$, and $\Gamma \vdash P \triangleright R \circ \Delta$ and $\Gamma \vdash P \triangleright R' \circ \Delta'$ are derived by [T-REC]. Let $\Gamma \vdash P' \triangleright R_1 \circ \tilde{c}_1 : Q_1^1, \ldots, \tilde{c}_n : Q_n^1$ where $R_1^{[X]} = R$ and $\tilde{c}_1 : Q_1^{1[X]}, \ldots, \tilde{c}_n : Q_1^{n[X]} = \Delta$, and $\Gamma \vdash P' \triangleright R_2 \circ \tilde{c}_1 : Q_1^2, \ldots, \tilde{c}_n : Q_n^2$ where $R_2^{[X]} = R'$ and $\tilde{c}_1 : Q_2^{1[X]}, \ldots, \tilde{c}_n : Q_2^{n[X]} = \Delta'$. By induction hypotheses, $R_1 \equiv R_2$ and $\lceil Q_1^1 \mid \ldots \mid Q_n^1 \rceil \equiv \lceil Q_1^2 \mid \ldots \mid Q_n^2 \rceil$ for each $1 \leq i \leq n$. Thus, we have $R \equiv R'$ and $\lceil \Delta \rceil \equiv \lceil \Delta' \rceil$. □

Proof of of Theorem 5.5.4

Proof. (Sketch) This lemma is guaranteed by the projection of sessions into roles and the generation of fresh channels in the session establishment. A formal proof is by induction on Sys $\xrightarrow{\tau}_* (\nu \tilde{a})(1 : P_1 \mid \ldots \mid n : P_n)$. □

Proof of Theorem 5.5.6

Proof. We first put forward two lemmas.

Lemma 5.7.2. *(1) If* $B \xrightarrow{p,q:v} B'$ *then* $B \restriction p\langle \tilde{c} \rangle \xrightarrow{b?v} B' \restriction p\langle \tilde{c}' \rangle$ *and* $B \restriction q\langle \tilde{c} \rangle \xrightarrow{b!v} B' \restriction q\langle \tilde{c}' \rangle$ *for some* $b \in \tilde{c} \supseteq \tilde{c}'$. *(2) If* $B \restriction p\langle \tilde{c} \rangle \xrightarrow{b?v} P$ *and* $B \restriction q\langle \tilde{c} \rangle \xrightarrow{b!v} Q$ *then there are* $B', \tilde{c}'$ *such that* $B \xrightarrow{p,q:v} B'$, $P \equiv B' \restriction p\langle \tilde{c}' \rangle$, $Q \equiv B' \restriction q\langle \tilde{c}' \rangle$ *and* $\tilde{c}' \subseteq \tilde{c}$.

Lemma 5.7.3. *Let* $\tilde{p} = p_1, \ldots, p_m$ *and* b *marks* B *in* C. *(1) If* $C \xrightarrow{\tilde{p}:B} C' \otimes B\langle \tilde{p} \rangle$ *then* $C \restriction p_1 \xrightarrow{\bar{b}_{[2..m]}(\tilde{c})} C' \restriction p_1$ *and* $C \restriction p_i \xrightarrow{b_{[i]}(\tilde{c})} C' \restriction p_i$ $(2 \leq i \leq m)$. *(2) If* $C \restriction p_1 \xrightarrow{\bar{b}_{[2..m]}(\tilde{c})} P_1$ *and* $C \restriction p_i \xrightarrow{b_{[i]}(\tilde{c})} P_i$ $(2 \leq i \leq m)$ *then there is* C' *such that* $P_j \equiv C' \restriction p_j$ $(1 \leq j \leq m)$ *and* $C \xrightarrow{\tilde{p}:B} C' \otimes B\langle \tilde{p} \rangle$.

Suppose Sys is well-typed by A_{spc}. We construct an $\mathcal{R}$ such that $\langle P, S \rangle \in \mathcal{R}$ if and only if

- Sys $\xrightarrow{\tau}_* P = (\nu \tilde{b})(1 : P_1 \mid \ldots \mid n : P_n)$,
- $A_{\mathrm{spc}} \xrightarrow{\tau}_* S = C \otimes B_1 \otimes \ldots \otimes B_k$,
- for each $1 \leq i \leq n$, $\Gamma \vdash P_i \triangleright R_i \circ \tilde{c}_1 : Q_1^i, \ldots, \tilde{c}_k : Q_k^i$ where
  - $C \restriction i\langle \tilde{a} \rangle \succ R_i$,
  - $B_j \restriction i\langle \tilde{c}_j \rangle \succ Q_j^i$ for each $1 \leq j \leq k$.

First, we have that $\langle \mathrm{Sys}, A_{\mathrm{spc}} \rangle \in \mathcal{R}$. Then, let $\langle P, S \rangle \in \mathcal{R}$ and suppose the above five induction hypotheses. Without loss of generality, we suppose (1) $P_1 \xrightarrow{b!v} P_1'$ and $P_2 \xrightarrow{b?v} P_2'$ or (2) $P_1 \xrightarrow{\bar{b}_{[2..m]}(\tilde{c}_0)} P_1'$ and $P_i \xrightarrow{b_{[i]}(\tilde{c}_0)} P_i'$ $(2 \leq i \leq m)$ and b marks B_0.

(1) Suppose $b \in \tilde{c}_j$. By Theorem 5.4.4, $Q_j^1 \xrightarrow{b!v} Q_j^{1\prime}$ and $Q_j^2 \xrightarrow{b?v} Q_j^{2\prime}$. Thus, by Lemma 5.7.2, $B_j \restriction 1\langle \tilde{c}_j \rangle \xrightarrow{b!v} B_j' \restriction 1\langle \tilde{c}_j' \rangle$ and $B_j \restriction 2\langle \tilde{c}_j \rangle \xrightarrow{b?v} B_j' \restriction 2\langle \tilde{c}_j' \rangle$ for some $B_j', \tilde{c}'$ such that $B_j \xrightarrow{1,2:v} B_j'$, $B_j' \restriction 1\langle \tilde{c}_j' \rangle \succ Q_j^{1\prime}$ and $B_j' \restriction 2\langle \tilde{c}_j' \rangle \succ Q_j^{2\prime}$. Also, $B_j \restriction i\langle \tilde{c}_j \rangle = B_j' \restriction i\langle \tilde{c}_j' \rangle$ for each $3 \leq i \leq n$. Therefore, let $S' = C \otimes B_1' \otimes B_2 \ldots \otimes B_k$ and $P' = (\nu \tilde{b})(1 : P_1' \mid 2 : P_2' \mid 3 : P_3 \mid \ldots \mid P_n)$. We have that $\langle P', S' \rangle \in \mathcal{R}$. (2) By Theorem 5.4.4, $R_1 \xrightarrow{\bar{b}_{[2..m]}(\tilde{c}_0)} R_1'$ and $R_i \xrightarrow{b_{[i]}(\tilde{c}_0)} R_i'$ $(2 \leq i \leq m)$. Thus, by Lemma 5.7.3, $C \restriction \langle \tilde{a} \rangle \xrightarrow{\bar{b}_{[2..m]}(\tilde{c}_0)} C' \restriction \langle \tilde{a} \rangle$ and $C \restriction \langle \tilde{a} \rangle \xrightarrow{b_{[i]}(\tilde{c}_0)} C' \restriction \langle \tilde{a} \rangle$ $(2 \leq i \leq m)$ for some $C', \tilde{a}, \tilde{a}'$ such that $b \in \tilde{a} \supseteq \tilde{a}'$, $C \xrightarrow{1,\ldots,m:B_0} C'$, $C' \restriction l\langle \tilde{a}' \rangle \succ R_l'$ $(2 \leq l \leq m)$. Also, $C \restriction l\langle \tilde{a} \rangle \equiv C' \restriction l\langle \tilde{a}' \rangle$ for each $m+1 \leq l \leq n$. Therefore, let $S' = C' \otimes B_0 \otimes B_1 \otimes \ldots \otimes B_k$ and $P' = (\nu \tilde{b})(1 : P_1' \mid \ldots \mid m : P_m' \mid P_{m+1} \ldots \mid P_n)$. We have that $\langle P', S' \rangle \in \mathcal{R}$. □

Proof of Theorem 5.6.1

Proof. We prove a more general proposition: if $\Gamma \vdash P \triangleright R \circ \Delta$ then

- $P^{\chi_M} \equiv R$,
- if $\bar{a}_{[2..n]}(\tilde{c}) \in \mathrm{act}(P)$ and $\Gamma \vdash a \triangleright B$, then $B \restriction 1 \equiv P^{\chi_{\tilde{c}}}$,
- if $a_{[k]}(\tilde{c}) \in \mathrm{act}(P)$ and $\Gamma \vdash a \triangleright B$, then $B \restriction k \equiv P^{\chi_{\tilde{c}}}$,
- if $\Delta[i] = \tilde{c} : Q$, then $Q \equiv P^{\chi_{\tilde{c}}}$.

We observe that if the above proposition holds then Theorem 5.6.1 immediately follows. The proof of the proposition is by induction on the derivation of $\Gamma \vdash P \triangleright R \circ \Delta$ according to Figure 5.7. The basic cases are simple. For the non-basic cases, we choose to deal with two typical cases. (1) $P = a_{[2..n]}(\tilde{c}).P'$ and $\Gamma \vdash P \triangleright R \circ \Delta$ is derived by [T-INV]. Let $R = \bar{a}_{[2..n]}(\tilde{c}).R'$ and $\Delta = \tilde{c} : B \restriction 1, \Delta'$. By induction hypotheses, $P'^{\chi_M} \equiv R'$ and, thus, $P^{\chi_M} \equiv R$. Suppose $\bar{b}_{[2..m]}(\tilde{c}') \in \mathrm{act}(P)$, $\Gamma \vdash b \triangleright B$, and $\tilde{c}' \cap \mathrm{ch}(\Delta) = \varnothing$. If $b = a$ (and thus $\tilde{c}' = \tilde{c}$, then by induction hypotheses and the rule [T-INV], $P'^{\chi_{\tilde{c}}} = B \restriction 1$. If

$b \neq a$ ..., by induction hypotheses, we have the same result. The case of $b_{[l]}(\tilde{c}')$ is similar. Suppose $\Delta[i] = \tilde{c}' : Q$. Then, $\tilde{c}' \cap \tilde{c} = \varnothing$ and $\Delta'[i+1] = \tilde{c} : Q$. Thus, $P^{\chi_{\tilde{c}'}} = P'^{\chi_{\tilde{c}'}}$ and by induction hypotheses $Q \equiv P^{\chi_{\tilde{c}'}}$. (2) $P = P_1 \mid P_2$ and $\Gamma \vdash P \triangleright R \circ \Delta$ is derived by [T-COM]. Let $R = R_1 \mid R_2$, $\Delta = \Delta_1 \mid \Delta_2$ and $\Gamma \vdash P_i \triangleright R_i \circ \Delta_i$ where $i \in \{1, 2\}$. By induction hypotheses and the rule [T-COM], we can obtain the four propositions above. (N.B. we have suppose P does not contain the hiding operator, so the rules [T-HID] and [T-VEI] are not applicable.) □

5.8 Related Work and Discussions

This chapter presents a session type theory, in which sessions are structured into the communication and integration levels, to deal with the peer-to-peer architectural modelling and analysis. A two-level type system is developed for π-calculus with session establishment primitives and several key properties of the type system are explored. In the analytic aspect, it is demonstrated that the well-typedness of a system by the type system guarantees a channel safety property and the behavioural conformance of components with respect to their session type-based specifications. Moreover, a process slicing method is presented to complement the type checking. To conclude, we present the related work of this chapter and discuss the shortcomings of the approach that are left to be addressed in the next chapter or in future.

Our work is rooted in the forgoing theories of session types, especially the global description of interactions and multiparty sessions. Carbone et al. [2007] presented two calculi to describe the communication behaviours from the global and local perspectives, respectively, and several principles to establish a sound and complete projection of the former to the latter. Some of the ideas behind the syntactic restrictions that we set up for the two-level sessions are related to their projection principles. The process calculus in the present paper is from Honda et al. [2008], in which the authors extended the traditional binary session types to the multiparty asynchronous context and solved several technical channels (as the result of the loss of two-party duality and the asynchrony) such that several fundamental properties of the session type discipline also hold by linearity

analysis. The syntax for the calculus is abstract (e.g. messages are treated as message types) and does not contain some syntactic features that are considered as essential to session type theories (e.g. the distinction of internal and external choices and message-based branching behaviours, as argued by Castagna and Padovani [2009]). Our intention is to focus on the two-level separation of session syntax and minimise the side techniques when studying relevant properties. We leave the work on enriching the syntax of session and calculi alike according to the existing session type theories in the future.

The subsequent work on session types witnesses a trend of increment on the expressive power to characterise richer conversation structures. For example, Deniélou and Yoshida [2011] extended the multiparty session types to accommodate the runtime change of session participants, i.e. the joining or leaving of participants, after a session is initiated. Yoshida et al. [2010] introduced a finite recursive type constructors into the multiparty session types to express a wide range of processes whose specification structures are parameterised whilst keeping the type checking for the resulting type system decidable. To improve protocol modularisation of session types, Demangeon and Honda Demangeon and Honda [2012] introduced a way to define abstract nested protocols independent of their host protocols such that the host protocols can call the nested ones by passing them arguments such as values, roles, and even (names of) other protocols. In these studies, the enrichment of the session type construction leads to the increment of syntactic primitives in the process calculi. In contrast, the separation of two-level sessions in our work does not complicate the syntax of the calculus. An interesting point is to compare the concept of nested protocols by Demangeon and Honda [2012] with that in the present paper. Their protocol calling is comparable to the procedure calling in the sequential programming, in which the exact position of the involvement must be specified to make sense of the main program. Our protocol nesting is more general in the sense that 'being nested by' just means 'occurring within'. Also, in our work, the meaning of the host protocol is complete with or without its nested protocol(s). Padovani [2012] proposed a backward approach to session types, in which session types are defined as projected fragments of processes. More specifically, a process is sliced as per channels it uses and session types are a type approximation of the channel-sliced

fragments of the process. There are two connecting points between his work and ours: first, both make use of process slicing, in spite of different purposes; second, both investigate sessions semantically.

The approach in this chapter provides a suitable formal method in the modelling and analysis of the peer-to-peer architecture, which complements the approaches for the connector-base architecture in the previous chapters. However, the syntax of the calculus is not rich enough to capture some primitives that are important to distributed computing, such as the distinction of internal and external choices and message-based branching behaviours [Allen and Garlan, 1997; Castagna and Padovani, 2009]. The intention is to focus on the two-level separation of session syntax and minimise the side techniques when studying the session types as behavioural approximations of processes. The enrichment of the syntax of the session types and the calculus to according to the existing session type work is left in the future work. Again, another disadvantage, shared by the all PA-based approaches and outlined throughout the thesis, is the gap between the abstract syntax of PA and the coarse-grained specifications in architectural description. Chapter 6 presents an approach that aims to partially address this problem by bridging PA notations and the (high-level) constraint-style specifications.

Chapter 6

Specification Merging

6.1 Introduction

In spite of the convenience for formal reasoning, a problem of PA, if used in architectural description directly, is the gap between its low-level syntax and the coarse-grained system specifications. This gap is bridged by the high-level syntax and the semantic translation of various process algebraic ADLs. In a similar vein, in Chapter 4, the constraint-style specifications (i.e. pre- and post-conditions of actions) of the architecture type and instance are written in (a fragment of) the ASM language and then translated into a PA notation in the semantic-level analysis. The chapter is towards a method to enhance the utility of PA as *ad hoc* architectural notations, by merging a PA notation which is equivalent to the model of Modal Transition Systems (MTSs) and the constraint-style specification language.

Recall that the operational semantics of PA compiles processes into LTSs and each LTS can be formulated as a process in a specific PA notation. In the modal case, we present a modal version of PA and show that it is logically equivalent to the MTS model, which is an extension of the LTS model by separating the necessary and possible transitions – an MTS without necessary transitions logically equals to an LTS. However, the MTS model rather than the modal PA is the primary formalism to be dealt with in this chapter. This is based on the consideration that, due to the extensive recent work on connection of them with

	Typed variables
User	$clt0$: Int, sev: $\{a, b\}$, wok: $2^{\{a,b\}}$
Server1	$clt1$: Int, $upd1$: Bool
Server2	$clt2$: Int, $upd2$: Bool

	Actions
User-Server1	$req1()$, $rtn1()$, $stp1()$, $rdy1()$
User-Server2	$req2()$, $rtn2()$, $stp2()$, $rdy2()$

Table 6.1: Typed (state) variables and actions of *AlterCS*

the scenario-based specification languages (such as MSC [ITU, 2000], LSC [Harel and Marelly, 2003]), MTSs are more notable in the software engineering research.[1] The complexity of the problem of merging arbitrary MTSs and (predicate) constraints are very high, therefore, the algorithm in this chapter assumes that the MTSs of interest are deterministic.

The remainder of this chapter is organised as follows. A specific research problem that motivates this chapter is reported in the next subsection. Definitions of MTSs and dynamic predicate logic are given in the Preliminaries (Section 6.2). Specifications of the running example in Section 6.1.1 are formally interpreted in Section 6.3. The model merging algorithm is presented in Section 6.4, together with a case study based on the running example. In Section 6.5, this chapter is concluded with remarks and discussions.

6.1.1 Problem: Specification Merging

The running example in this chapter is a client-server system called *AlterCS*, the main feature of which is that the users enjoy alternative services from two servers which may stop working occasionally. Because in this chapter, we do not consider verification techniques between the architecture type and instance, *AlterCS* is described in the architecture-instance level only. *AlterCS* consist of three components, i.e. *User*, *ServerA*, and *ServerB*. State variables (or variables) of the components and actions performed by them are presented in Table 6.1. *User* has three variables: $clt0$ indicates its control state, sev is the name of the

[1] A detailed survey for this research area is beyond the scope of this chapter.

server that it prepares to request, and *wok* restores the name of the working servers. Both servers have two variables: *clti* and *updi* indicates the control state and the working condition of *Serveri* for each $i \in \{1, 2\}$. *User* sends $req1()$ to *Server1* to make a request and waits for the result $rtn1()$; on the other hand, *Server1* informs *User* whether it is ready to work by actions $rdy1()$ and $stp1()$ ($rdy1()$ means 'ready to work' and $stp1()$ means 'stop working'). $req2()$, $rtn2()$, $stp2()$, and $rdy2()$ have the similar functionalities between *User* and *Server2*. The typing information of the variables are also provided. In particular, *wok*: $2^{\{a,b\}}$ means that the value of *wok* ranges over all subsets of $\{a, b\}$. We let a and b stand for *Server1* and *Server2*, respectively.

The behavioural specifications *AlterCS* are also provided. There are two fragments of the behavioural specifications, one presented as an MTS and the other as a set of constraints. The formal definition of MTSs is provided in Section 6.2.1; for the time being, the MTS for *AlterCS* is documented in a semi-formal way below:

- The states of the MTS for *AlterCS* are subsets of $\{req1(), req2(), stp1(), stp2()$ and the initial state is $\varnothing$, the empty set;
- The possible and necessary transitions of the MTS for *AlterCS* are specified as follows: let s be a state and $i \in \{1, 2\}$,
  - if $reqi() \notin s$ then there is a possible transition between s and $s' = s \cup \{reqi()\}$,
  - if $reqi() \in s$ then there is a necessary transition between s and $s' = s \backslash \{rtni()\}$,
  - if $stpi() \in s$ then there is a possible transition between s and $s' = s \cup \{stpi()\}$,
  - if $stpi() \notin s$ then there is a necessary transition between s and $s' = s \backslash \{stpi()\}$.

The constraints for *AlterCS* are provided in Figure 6.1. The set of constraints defines an LTS via dynamic predicate logic, which is presented in Section 6.2.2. For now, it is sufficient to informally illustrate the problem that motivates this

```
// pre/post-conditions //
  req1()::
    pre: clt1=1 & upd1=0 & clt0=1 & sev=a
    post: clt1=2 & clt0=2
  req2()::
    pre: clt2=1 & upd2=0 & clt0=1 & sev=b
    post: clt2=2 & clt0=2
  stp1()::
    pre: upd1=0 & wok=SET(a,b)
    post: upd1=1 & wok=SET(b) & sev=b
  stp2()::
    pre: upd2=0 & wok=SET(a,b)
    post: upd2=1 & wok=SET(a) & sev=a
// initial values //
  clt0=clt1=clt2=1
  upd1=upd2=0
  sev=a
// invariant //
  if (a In wok) then sev=a
```

Figure 6.1: Constraints for $CSsystem_2$

chapter: *how to merge the specification of* $\mathcal{M}_{cs}$ *and the constraints?* The merging has at least two purposes. The first purpose is to check whether two kinds of specification are consistent. The second one is to make use of information in one kind to elaborate the other kind, together yielding a unified specification which narrows the gap between the early specifications and the mature design.

6.2 Preliminaries

6.2.1 Behavioural Models

Definition 6.2.1 (Labelled Transition System)**.** *An LTS is the tuple*

$$\mathcal{S} = (S_{\mathcal{S}}, ini_{\mathcal{S}}, A_{\mathcal{S}}, \longrightarrow_{\mathcal{S}})$$

where $S_{\mathcal{S}}$ is a finite set of states, $s_{\mathcal{S}} \in S_{\mathcal{S}}$ is the initial state, $A_{\mathcal{S}} \subseteq Act$ is a set of actions such that the internal action $\tau \in A_{\mathcal{S}}$, and $\longrightarrow_{\mathcal{S}} \subseteq S_{\mathcal{S}} \times A_{\mathcal{S}} \times S_{\mathcal{S}}$ is the transition relation.

Given two $\mathcal{S}, \mathcal{S}'$, we write $\mathcal{S} \sqsubseteq \mathcal{S}'$ if $S_{\mathcal{S}} \subseteq S_{\mathcal{S}'}$, $ini_{\mathcal{S}} = ini_{\mathcal{S}'}$, $A_{\mathcal{S}} \subseteq A_{\mathcal{S}'}$, and $\longrightarrow_{\mathcal{S}} \subseteq \longrightarrow_{\mathcal{S}'}$.

Definition 6.2.2 (Modal Transition System)**.** *An MTS is the tuple*

$$\mathcal{M} = (S_{\mathcal{M}}, ini_{\mathcal{M}}, A_{\mathcal{M}}, \longrightarrow_{\Box\mathcal{M}}, \longrightarrow_{\Diamond\mathcal{M}})$$

where $S_{\mathcal{M}}$ is a finite set of states, $s_{\mathcal{M}} \in S_{\mathcal{M}}$ is the initial state, $A_{\mathcal{M}} \subseteq Act$ is a set of actions such that the internal action $\tau \in A_{\mathcal{M}}$, and $\longrightarrow_{\Box\mathcal{M}} \subseteq \longrightarrow_{\Diamond\mathcal{M}} \subseteq S_{\mathcal{M}} \times A_{\mathcal{M}} \times S_{\mathcal{M}}$ are the necessary and possible transition relations, respectively.

Let $s \xrightarrow{\widetilde{\alpha}}_{\Diamond\mathcal{M}} s'$ (resp. $s \xrightarrow{\widetilde{\alpha}}_{\Box\mathcal{M}} s'$, $s \xrightarrow{\widetilde{\alpha}}_{\mathcal{S}} s'$) if and only if there are $s_1, \ldots, s_{n+1}$ such that $s = s_1$, $s' = s_{n+1}$, and $s_i \xrightarrow{\alpha_i}_{\Diamond\mathcal{M}} s_{i+1}$ (resp. $s_i \xrightarrow{\alpha_i}_{\Box\mathcal{M}} s_{i+1}$, $s_i \xrightarrow{\alpha_i}_{\mathcal{S}} s_{i+1}$) for each $1 \leq i \leq n+1$. For each $\widetilde{\alpha}$, we use $\widetilde{\alpha}|_{A_{\mathcal{S}}}$ (resp. $\widetilde{\alpha}|_{A_{\mathcal{M}}}$) to denote the projection of $\widetilde{\alpha}$ into $A_{\mathcal{S}}$ (resp. $A_{\mathcal{M}}$). Let $|\mathcal{S}|$, the size of $\mathcal{S}$, is $|S_{\mathcal{S}}| + |\longrightarrow_{\mathcal{S}}|$, and $|\mathcal{M}| = |S_{\mathcal{M}}| + |\longrightarrow_{\Diamond\mathcal{M}}|$.

Definition 6.2.3 (Refinement)**.** *$\mathcal{S}$ weakly refines $\mathcal{M}$, denoted $\mathcal{M} \preccurlyeq \mathcal{S}$, if the following holds: there is a relation $\sim \subseteq S_{\mathcal{M}} \times S_{\mathcal{S}}$ such that $ini_{\mathcal{M}} \sim ini_{\mathcal{S}}$, and*

- *if $w \sim s$ and $s \xrightarrow{\alpha}_{\mathcal{S}} s'$, then*
  - *if $\alpha \in A_{\mathcal{M}} \backslash \tau$ then there is w' such that $w \xrightarrow{\alpha}_{\Diamond\mathcal{M}} w'$ and $w' \sim s'$,*
  - *otherwise, $w' \sim s$;*
- *if $w \sim s$, $w \xrightarrow{\alpha}_{\Box\mathcal{M}} w'$, then*
  - *if $\alpha \neq \tau$, then there are $\widetilde{\gamma}, s'$ such that $s \xrightarrow{\widetilde{\gamma}}_{\mathcal{S}} s'$, $\widetilde{\gamma}|_{A_{\mathcal{S}}} = \alpha$, $w' \sim s'$,*
  - *otherwise, $w \sim s'$.*

The relation $\sim$ is called a refinement relation for $\mathcal{M}, \mathcal{S}$.

Definition 6.2.4 (Determinism)**.** *$\mathcal{S}$ is deterministic if $s \xrightarrow{\alpha} s'$ and $s \xrightarrow{\alpha} s''$ for any α imply $s' = s''$. $\mathcal{M}$ is deterministic if $w \xrightarrow{\alpha} w'$ and $w \xrightarrow{\alpha} w''$ for any α imply $w' = w''$.*

When using LTSs or MTSs to represent specifications in this chapter, states in $S_{\mathbb{S}}$ or $S_{\mathbb{M}}$ may be structured rather than just names of nodes. For example, informally speaking[1], in the MTS of *AlterCS*, a state is a set of actions, and in the LTS translated from the constraints of *AlterCS*, a state is an evaluation of state variables. In the literature, LTSs with structured states are sometimes called Statecharts or Kripke Transition System.

Modal PA A modal version of PA denoted $\mathcal{PA}_m$, which adds modality into the action prefixes of PA, and its connection to the MTS model is presented. Note that the presented modal PA is not itself an object of study, and the main techniques in the chapter are based on the MTS model.

The following grammars define the syntax of the modal PA:

$$P, Q ::= \mathbf{0} \mid X \mid \alpha_{\Box}.P \mid \alpha_{\diamond}.P \mid P + Q$$

where $\alpha \in Act$, a finite set of actions, and $\Box$ and $\diamond$ respectively stand for the possibility and necessity. As before, for each process variable X, an equation $X \stackrel{\text{def}}{=} P$ is defined. The operational semantics of are given as follows:

$$\alpha_{\Box}.P \xrightarrow{\alpha}_{\Box} P \qquad \alpha_{\heartsuit}.P \xrightarrow{\alpha}_{\diamond} P$$

$$\frac{P \xrightarrow{\alpha}_{\heartsuit} P'}{P + Q \xrightarrow{\alpha}_{\heartsuit} P'} \qquad \frac{Q \xrightarrow{\alpha}_{\heartsuit} Q'}{P + Q \xrightarrow{\alpha}_{\heartsuit} Q'}$$

$$\frac{P \xrightarrow{\alpha}_{\heartsuit} P' \quad X \stackrel{\text{def}}{=} P}{X \xrightarrow{\alpha}_{\heartsuit} P'}$$

where $\heartsuit \in \{\Box, \diamond\}$. By the semantic rules, each P of $\mathcal{PA}_m$ is corresponded to an MTS $\mathcal{M}_P$, with $Proc(P)$ (the set of processes reachable from P) being the its set of states. If the set of process variables is finite, then $Proc(P)$ is finite. On the other hand, given $\mathcal{M}$, a process $P_{\mathcal{M}}$ can be defined as follows: (1) for each $w \in W$, select a new process variable X_w; (2) let $W' = \{w' \in W \mid w \xrightarrow{\alpha}_{\heartsuit} w'\}$, then $X_w \stackrel{\text{def}}{=} \sum_{w' \in W'} \alpha_{\heartsuit}.X'_w$; and (3) let $P_{\mathcal{M}}$ be X_{w_0}.

[1]Formal interpretation is given in Section 6.3.

6.2.2 Dynamic Predicate Logic

A dynamic predicate logic $\mathcal{L}_d$ is defined. Bear in mind that the syntax of $\mathcal{L}_d$ is tailored to interpret the constraints of *AlterCS*. Terms, formulas, and programs of $\mathcal{L}_d$ are given by the following syntax.

$$\begin{aligned} t, r &::= x \mid \mathrm{SET}(t_1, \ldots, t_n) && \text{terms} \\ \varphi, \psi &::= t \equiv r \mid t \text{ In } r \mid \varphi \wedge \psi \mid \neg\varphi && \text{formulas} \\ \delta, \theta &::= \varphi? \mid \alpha \mid x \leftarrow t \mid \delta; \theta && \text{programs} \end{aligned}$$

where x is variable, SET is a finite-ary function, $\equiv$ is the equality, In is also a two-ary predicate, $\varphi?$ is a testing, α denotes an action, $x \leftarrow t$ is an update assignment, and $\delta; \theta$ is the concatenation of δ, θ. The implication connective $\rightarrow$ is defined by $\neq, \wedge$ in the standard way.[1]

Given a set $\mathfrak{D}$, called the domain of $\mathcal{L}_d$, an assignment $\mathbf{I}_\mathfrak{D}$ of $\mathcal{L}_d$ based on $\mathfrak{D}$ is function mapping each variable to an element in $\mathfrak{D}$. The set of all assignments based on $\mathfrak{D}$ of $\mathcal{L}_d$ is denoted $\mathbb{I}_\mathfrak{D}$. . Semantics rules for terms and formulas of $\mathcal{L}_d$ follow.

- $\mathrm{SET}(t_1, \ldots, t_n)^\mathbf{I} = \{t_1^\mathbf{I}, \ldots, t_n^\mathbf{I}\}$
- $\mathbf{I} \vDash t \equiv r$ if $t = r$
- $\mathbf{I} \vDash t \text{ In } r$ if $t \in r$
- $\mathbf{I} \vDash \neg\varphi$ if not $\mathbf{I} \vDash \varphi$
- $\mathbf{I} \vDash \varphi \wedge \psi$ if $\mathbf{I} \vDash \varphi$ and $\mathbf{I} \vDash \psi$

The semantics of programs are update functions on $\mathbb{I}_\mathfrak{D}$, i.e. *partial* functions f from $\mathbb{I}_\mathfrak{D}$ into itself. Specify f_i as the total identity function

- $\varphi?^{\mathbb{I}_\mathfrak{D}}$ is $f_i \restriction \{\mathbf{I} \in \mathbb{I}_\mathfrak{D} \mid \mathbf{I} \vDash \varphi\}$,
- $(x \leftarrow t)^{\mathbb{I}_\mathfrak{D}}$ is $f : \mathbf{I} \rightarrow \mathbf{I}[x := t^\mathbf{I}]$ for each $\mathbf{I} \in \mathbb{I}_\mathfrak{D}$,

[1] For convenience, in this chapter, $\wedge, \longrightarrow, \neg$ are used both as logical symbols in $\mathcal{L}_d$ and as first-order meta-logical symbols (together with $\vee, \exists, \forall$) and no confusion is expected to be caused.

- $(\delta;\theta)^{\mathbb{I}_{\mathfrak{D}}}$ is the function combination $\theta^{\mathbb{I}_{\mathfrak{D}}} \circ \delta^{\mathbb{I}_{\mathfrak{D}}}$.

6.3 Specification Interpretation

The MTS specification for *AlterCS* is directly corresponded to a deterministic MTS model denoted $\mathcal{M}_{cs}$ where

- $S_{\mathcal{M}_{cs}} = 2^{\{reqi(),stpi() \mid i=1,2\}}$,
- $ini_{\mathcal{M}_{cs}} = \varnothing$.
- $A_{\mathcal{M}_{cs}} = \{reqi(), rtni(), stpi(), rdyi() \mid i = 1, 2\}$,
- let $i \in \{1, 2\}$, for each $s, s' \in S_{\mathcal{M}_{cs}}$:
  - $s \xrightarrow{reqi}_{\Box\mathcal{M}_{cs}} s'$ if $reqi() \notin s$ and $s' = s \cup \{reqi()\}$,
  - $s \xrightarrow{rtni}_{\Diamond\mathcal{M}_{cs}} s'$ if $reqi() \in s$ and $s' = s \backslash \{reqi()\}$,
  - $s \xrightarrow{stpi}_{\Box\mathcal{M}_{cs}} s'$ if $stpi() \notin s$ and $s' = s \cup \{stp()\}$,
  - $s \xrightarrow{rdyi}_{\Diamond\mathcal{M}_{cs}} s'$ if $rdyi() \in s$ and $s' = s \backslash \{reqi()\}$,
  - $s \xrightarrow{\alpha}_{\Diamond\mathcal{M}_{cs}} s'$ if $s \xrightarrow{\alpha}_{\Box\mathcal{M}_{cs}} s'$ for each $\alpha \in S_{\mathcal{M}_{cs}}$.

The structural specification and the constraints for *AlterCS* defines an LTS $\mathcal{S}_{cs}$ where

- $S_{\mathcal{S}_{cs}}$ is the set of assignments based on $\mathfrak{D}_{\text{cs}} = \text{Int} \cup \text{Bool} \cup \{a, b\} \cup 2^{\{a,b\}}$,
- $ini_{\mathcal{S}_{cs}}$ is the assignment $\mathbf{I}_{\text{cs}}$ such that
  - $\mathbf{I}_{\text{cs}}(a) = a$, $\mathbf{I}_{\text{cs}}(b) = b$,
  - $\mathbf{I}_{\text{cs}}(wok) = \{a, b\}$, $\mathbf{I}_{\text{cs}}(sev) = a$,
  - $\mathbf{I}_{\text{cs}}(clt0) = \mathbf{I}_{\text{cs}}(clt1) = \mathbf{I}_{\text{cs}}(clt2) = 1$,
  - $\mathbf{I}_{\text{cs}}(upd1) = \mathbf{I}_{\text{cs}}(upd2) = 0$,

  ($\mathbf{I}_{\text{cs}}$ is induced by the initial values in Figure 6.1)

- for each $s, s' \in S_{\mathcal{S}}$, $s \xrightarrow{\alpha}_{\mathcal{S}} s'$ if and only if $f_\alpha(s) = s'$ where f_α is

$$(\varphi_{inv}?; \varphi_\alpha?; \delta_\alpha)^{\mathbb{I}_{\mathfrak{D}_{\text{cs}}}} \quad .$$

(φ_{inv} is the invariant, φ_α is the pre-condition of α, and δ_α is the post-condition of α)

6.4 The Merging Algorithm

The problem raised in Section 6.1.1 can be rephrased as follows: *given a deterministic MTS* $\mathcal{M}$ *and a Kripke tuple* $(\mathcal{S}, \rho)$*, how to generate another Kripke tuple* $(\mathcal{S}^*, \rho)$ *where such that* $\mathcal{S} \sqsubseteq \mathcal{S}^*$ *and* $\mathcal{M} \preccurlyeq \mathcal{S}^*$*?* An algorithm to fulfil this task is the goal of this section. In this end of this section, an outcome of an algorithm for *AlterCS* is provided.

Several preliminary data structures are generated before presenting the main part of the model merging algorithm. If there are $s_0, \ldots, s_{n+1}$ such that $s = s_0$, $s' = s_n$, $s_i \xrightarrow{\tau}_{\heartsuit\mathcal{M}} s_{i+1}$ for each $1 \leq i \leq n$, and $s_n \xrightarrow{\alpha}_{\heartsuit\mathcal{M}} s'$ for some $\alpha \neq \tau$, we write $s \Longrightarrow_{\heartsuit\mathcal{M}} s'$ and let $\text{Pass}(s \overset{\alpha}{\Longrightarrow}_{\heartsuit\mathcal{M}} s') = \{s_1, \ldots, s_n\}$. Informally, $\Longrightarrow_{\diamondsuit\mathcal{M}}$ and $\Longrightarrow_{\square\mathcal{M}}$ are respectively the observable concatenations of $\longrightarrow_{\diamondsuit\mathcal{M}}$ and $\longrightarrow_{\square\mathcal{M}}$, which absorb invisible internal transitions. The set $\text{Pass}(w \overset{\alpha}{\Longrightarrow}_{\heartsuit\mathcal{M}} w')$ contains the passed states (except for the beginning and terminal ones, i.e. w and w') of $w \overset{\alpha}{\Longrightarrow}_{\heartsuit\mathcal{M}} w'$. Let $\text{Pass}_{\mathcal{M}} = \{\text{Pass}(w \overset{\alpha}{\Longrightarrow}_{\diamondsuit\mathcal{M}} w') \mid w, w' \in S_{\mathcal{M}}, \alpha \in A_{\mathcal{M}}\}$. A depth-first search on $\mathcal{M}$ suffices to generate $\Longrightarrow_{\diamondsuit\mathcal{M}}$, $\Longrightarrow_{\square\mathcal{M}}$ and $\text{Pass}_{\mathcal{M}}$. The complexity of the generation of three data structures is the linear size of $\mathcal{M}$, i.e. $O(|\mathcal{M}|)$. Let $s \xrightarrow{\widetilde{\alpha}}_{\mathcal{S}/A_{\mathcal{M}}} s'$ if $s \xrightarrow{\widetilde{\alpha}}_{\mathcal{S}} s'$ and $\widetilde{\alpha} \cap A_{\mathcal{M}} = \varnothing$. Similarly, a depth-first search on $\mathcal{S}$ can generate $\longrightarrow_{\mathcal{S}/A_{\mathcal{M}}}$ and the complexity is $O(|\mathcal{S}|)$.

The model merging algorithm is presented in Algorithm 3 (with two procedures *Moves*() and *Shrink*()).

Let $S_{\mathcal{S}^*} = S_{\mathcal{S}} \uplus S_{\mathcal{M}}$ and $A_{\mathcal{S}^*} = A_{\mathcal{S}} \cup A_{\mathcal{M}}$. For readability, in the sequel, if not specified explicitly, let's use s, s' to denote elements in $S_{\mathcal{M}}$, w, w' to denote elements in $S_{\mathcal{S}}$, and v, v' elements of $S_{\mathcal{S}^*}$. In general, Algorithm 3 visits elements in $S_{\mathcal{S}^*} \times S_{\mathcal{M}}$ and attempts to generate a transition relation $\longrightarrow_{\mathcal{S}^*}$. If no such $\longrightarrow_2$ can be generated, it returns a negative answer. Besides the $\longrightarrow_{\mathcal{S}^*}$, several

Algorithm 3: Model generation

Input : $\mathcal{M}$, $\mathcal{S}$, $\Longrightarrow_{\Diamond\mathcal{M}}$, $\Longrightarrow_{\Box\mathcal{M}}$, $\longrightarrow_{\mathcal{S}/\mathcal{A}_{\mathcal{M}}}$
Output: $\longrightarrow_{\mathcal{S}_*}$ or 'no'

1 **subset** of $S_{\mathcal{S}_*} \times S_{\mathcal{M}}$: $\sim := \varnothing$, $\chi := \varnothing$
2 **subset** of $S_{\mathcal{S}_*} \times A_{\mathcal{S}_*} \times S_{\mathcal{S}_*}$: $\longrightarrow_{\mathcal{S}_*} := \varnothing$
3 **bool** $bool := 0$
4 **let** $\chi := \{\langle ini_{\mathcal{S}}, ini_{\mathcal{M}}\rangle\}$
5 **while** $\chi \neq \varnothing \vee bool \neq 1$ **do**
6 **let** $\iota = Moves(\chi, \sim, \longrightarrow_{\mathcal{S}_*})$
7 **case** $\iota =$ 'false'
8 $bool := 1$
9 **case** $\iota = \langle \Phi, \Psi \rangle$
10 $\longrightarrow_{\mathcal{S}_*} := \longrightarrow_{\mathcal{S}_*} \cup \Psi$
11 $\chi := \Phi - \sim$
12 $\sim := \sim \cup \Phi$
13 **if** $bool = 1$ **then return** 'no'
14 **else return** $Shrink(\longrightarrow_{\mathcal{S}_*}, \sim)$

variables are manipulated by Algorithm 3, including $\sim$, χ, ι, Φ, Ψ, $bool$, and $bool'$. $\sim$ records the visited elements and is refinement-to-be between $\mathcal{S}_*$ and $\mathcal{M}$. We call elements in $\sim$ *corresponded pairs*. χ, which is a subset of $\sim$, records the elements visited for the first time. χ, which is a subset of $\sim$, records the elements visited for the first time. ι refers to the results returned from the procedure $Moves()$ that is either the string "false" or $\langle \Phi, \Psi \rangle$, where $\Phi \subseteq S_{\mathcal{S}_*} \times S_{\mathcal{M}}$ and $\Psi \subseteq S_{\mathcal{S}_*} \times A_{\mathcal{S}_*} \times S_{\mathcal{S}_*}$. $bool$ and $bool'$ are boolean variables to indicate whether an intended $\mathcal{S}_*$ can be generated successfully.

After the initiation of relevant variables, the execution of Algorithm 3 visits the singleton $\{\langle w_0, s_0\rangle\}$ (Line 4) and then enters a "while" loop (Line 5 to Line 12), whose exit condition is either $\chi = \varnothing$, meaning that the execution of Algorithm 3 succeeds, or $bool = 1$, meaning that it fails. In the body of the loop, the execution involves the procedure $Moves()$ (Line 6).

The purpose of $Moves()$, which takes $\chi, \sim, \longrightarrow_{\mathcal{S}_*}$ as parameters, is to attempt to generate a pair $\langle \Phi, \Psi \rangle$ that eventually leads to a transition relation of the intended $\mathcal{S}_*$ and a refinement relation between the intended $\mathcal{S}_*$ and $\mathcal{M}$. $Moves()$

Procedure $Moves(\chi, \sim, \longrightarrow_{\mathbb{S}_*})$

subset of $S_{\mathbb{S}_*} \times S_{\mathcal{M}}$: $\Phi := \varnothing$
subset of $S_{\mathbb{S}_*} \times A_{\mathbb{S}_*} \times S_{\mathbb{S}_*}$: $\Psi := \varnothing$
bool $bool' := 0$
foreach $\langle v, s\rangle \in \chi$, $\alpha \in A_{\mathbb{S}}$ **do**
  if $\exists v'(v \xrightarrow{\alpha}_{\mathbb{S}} v' \vee v \xrightarrow{\alpha}_{\mathbb{S}_*} v' \vee v \overset{\alpha}{\Longrightarrow}_{\Box\mathcal{M}} v')$ **then**
    if $\alpha \in A_{\mathcal{M}}$ **then**
      if $\exists s'(s \overset{\alpha}{\Longrightarrow}_{\Diamond\mathcal{M}} s')$ **then**
        $\Phi := \Phi \cup \{\langle v, s''\rangle \mid s'' \in \mathrm{Pass}(s \overset{\alpha}{\Longrightarrow}_{\Diamond\mathcal{M}} s')\} \cup \{\langle v', s'\rangle\}$
        $\Psi := \Psi \cup \{\langle v, \alpha, v'\rangle\}$
      else $bool' := 1$
    else $\Phi := \Phi \cup \{\langle v', s\rangle\}$, $\Psi := \Psi \cup \{\langle v, \alpha, v'\rangle\}$
  if $\alpha \neq \tau \wedge \exists s'(s \xrightarrow{\alpha}_{\Box\mathcal{M}} s')$ **then**
    $\Psi := \Psi \cup \{\langle v, \alpha, s'\rangle\}$
    foreach $s_1 \in S_{\mathcal{M}}(s_1 \neq s \wedge v \sim s_1)$ **do**
      if $\exists s_1'(s_1 \overset{\alpha}{\Longrightarrow}_{\Diamond\mathcal{M}} s_1')$ **then**
        $\Phi := \Phi \cup \{\langle v, s_1''\rangle \mid s_1'' \in \mathrm{Pass}(s_1 \overset{\alpha}{\Longrightarrow}_{\Diamond\mathcal{M}} s_1')\} \cup \{\langle v', s_1'\rangle\}$
      else $bool' := 1$
  if $\alpha = \tau \wedge \exists s'(s \xrightarrow{\alpha}_{\Box\mathcal{M}} s')$ **then**
    $\Phi := \Phi \cup \{\langle v, s'\rangle\}$
if $bool' = 0$ **then return** $\langle \Phi, \Psi\rangle$
else return 'false'

guarantees that Φ, Ψ satisfy the following conditions: for each $\alpha \in A_{\mathbb{S}_*}$,

$$\frac{v \sim s \wedge s \xrightarrow{\alpha}_{\Box\mathcal{M}} s' \wedge \alpha \neq \tau}{\exists v'(\langle v', s'\rangle \in \Phi \wedge \langle v, \alpha, v', \rangle \in \Psi)} \qquad \frac{v \sim s \wedge s \xrightarrow{\tau}_{\Box\mathcal{M}} s'}{\langle s, s'\rangle \in \Phi}$$

$$\frac{v \sim s \wedge (v \xrightarrow{\alpha}_{\mathbb{S}} v' \vee v \xrightarrow{\alpha}_{\mathbb{S}_*} v' \vee v \overset{\alpha}{\Longrightarrow}_{\Box\mathcal{M}} v')}{\exists s'(\langle v, s, \alpha, v', s'\rangle \in \Psi)}$$

$$\frac{v \sim s \wedge \langle v, \alpha, v'\rangle \in \Psi \wedge \alpha \in A_{\mathcal{M}}}{\exists s'(s \overset{\alpha}{\Longrightarrow}_{\Diamond\mathcal{M}} s' \wedge \langle v', s'\rangle \in \Phi) \wedge \forall s'' \in \mathrm{Pass}(s \overset{\alpha}{\Longrightarrow}_{\Diamond\mathcal{M}} s')(\langle v, s''\rangle \in \Phi)}$$

$$\frac{v \sim s \wedge \langle v, \alpha, v'\rangle \in \Psi \wedge \alpha \notin A_{\mathcal{M}}}{\langle v', s'\rangle \in \Phi}$$

If no such Ψ can be generated, $Moves()$ returns the string "false".

Back to the main of Algorithm 3, depending on the return type of $Moves()$,

Procedure $Shrink(\longrightarrow_{\mathcal{S}_*}, \sim)$

foreach $w \in S_{\mathcal{S}}$, $\alpha \in A_{\mathcal{M}}$ **do**
  if $\exists s, w', \widetilde{\beta}(w \xrightarrow{\alpha}_{\mathcal{S}_*} s \wedge w \xrightarrow{\widetilde{\beta}}_{\mathcal{S}/A_{\mathcal{M}}} w' \wedge w' \xrightarrow{\alpha}_{\mathcal{S}_*} s)$ **then**
    $\longrightarrow_{\mathcal{S}_*} := \longrightarrow_{\mathcal{S}_*} \setminus \langle w, \alpha, s \rangle$

foreach $s \in S_{\mathcal{M}}$ **do**
  if $\exists w \neq s (s \sim s \wedge w \sim s)$ **then**
    $\longrightarrow_{\mathcal{S}_*} := \longrightarrow_{\mathcal{S}_*} \{w/s\}$

return $\longrightarrow_{\mathcal{S}_*}$

i.e. the type of ι, the execution of Algorithm 3 separates into two branches. If the first branch is chosen, execution assigns the value 1 to *bool* (Line 8) and is ready to exit the loop. If the second branch is chosen, the execution manipulates $\sim$, χ and $\longrightarrow_{\mathcal{S}_*}$ in accordance with the value of ι. The manipulation of $\sim$, χ and $\longrightarrow_{\mathcal{S}_*}$ consists of three commands: new transitions are added into the on-the-fly $\longrightarrow_{\mathcal{S}_*}$ (Line 10); new corresponding pairs is packed in χ (Line 11); and they are further added into $\sim$ (Line 12). Hence, when the "while" loop is exited on the condition that $\chi = \varnothing$ (in this case, $\chi = \Phi - \sim = 0$ in the last execution of the loop), we already have that $\mathcal{M} \preccurlyeq \mathcal{S}_*$ with $\sim$ being the refinement relation, and and $\mathcal{S} \sqsubseteq \mathcal{S}_*$ (for $\longrightarrow_{\mathcal{S}} \subseteq \longrightarrow_{\mathcal{S}_*}$). After exiting the loop, the execution returns results according to different situations and $Shrink()$, which takes $\longrightarrow_{\mathcal{S}_*}$ and $\sim$ as parameters, is possibly involved (Line 14). Here an auxiliary definition is made: $\longrightarrow_{\mathcal{S}_*} \{w/s\}$ is the transition relation obtained from $\longrightarrow_{\mathcal{S}_*}$ by replacing all occurrences of s by v. $Shrink()$ sharps $\longrightarrow_{\mathcal{S}_*}$ by cutting off and merging some transitions. The utility of $Shrink()$ will be discussed shortly. For now, it should be noticed that $Shrink()$ invalidates neither $\mathcal{M} \preccurlyeq \mathcal{S}_*$ nor $\mathcal{S} \sqsubseteq \mathcal{S}_*$.

Theorem 6.4.1. (Correctness of Algorithm 3) *If Algorithm 3 returns* $\longrightarrow_{\mathcal{S}_*}$, *then* $\mathcal{S}_* \preccurlyeq \mathcal{M}$ *and* $\mathcal{S} \sqsubseteq \mathcal{S}_*$.

The proof of the above theorem, i.e. the correctness of Algorithm 3, is contained in the explanation of the algorithm. The algorithm is also complete, as stated by the theorem below.

We say $\mathcal{M}$ and $\mathcal{S}$ are *consistent*, if there exists an LTS $\mathcal{S}'$ such that $\mathcal{M} \preccurlyeq \mathcal{S}'$ and $\mathcal{S} \sqsubseteq \mathcal{S}'$.

Theorem 6.4.2. (Completeness of Algorithm 3) *If* $\mathcal{M}$ *and* $\mathcal{S}$ *are consistent then Algorithm 3 returns* $\longrightarrow_{\mathcal{S}_*}$.

A proof for completeness requires to show that, if such an $\mathcal{S}_*$ exists, in each time the value of $bool'$ in the procedure $Moves()$ is 0. The informal idea of the proof is as follows. Let $\sim$ be the $\sim$ after Algorithm 3 returns $\longrightarrow_{\mathcal{S}_*}$. $bool' := 1$ is executed in the procedure $Moves()$ if and only if one of the two cases happens: there are $v \in S_1, s \in S$ such that $v \sim s$, but there is no refinement relation containing the pair $\langle v, s\rangle$; there are $v \in S_{\mathcal{S}}, s, s' \in_{\mathcal{M}}$ such that $v \sim s$, $v \sim s'$, but s, s' do not refine each other. However, the existence of a correct refinement excludes the possibility of both cases.

Let us now analyze the complexity of Algorithm 3. The maximum time required to query a given tuple in or insert it into any of the transition relations $\Longrightarrow_{\Diamond\mathcal{M}}$, $\Longrightarrow_{\Box\mathcal{M}}$, and $\Longrightarrow_{\mathcal{S}/A_{\mathcal{M}}}$ or $\sim$ is certain. The procedure $Moves()$ takes no more than $k \cdot |\chi| \cdot |A_{\mathcal{S}_*}| \cdot |S_{\mathcal{M}}|$ steps for some integer k. Because it is ensured that the values of χ in any two executions of the "while" loop do not overlap, the complexity of "while-loop" is $k' \cdot |S_{\mathcal{S}_*}| \cdot |S_{\mathcal{M}}|^2 \cdot |A_{\mathcal{S}_*}|$ for some integer k'. The procedure $Shrink()$ does not increase the complexity. Hence, the complexity of Algorithm 3 is $O(|S_{\mathcal{S}_*}| \cdot |S_{\mathcal{M}}|^2 \cdot |A_{\mathcal{S}_*}|)$ where $|S_{\mathcal{S}_*}| = |S_{\mathcal{M}}| + |S_{\mathcal{S}}|$ and $|A_{\mathcal{S}_*}| = |A_{\mathcal{M}}| + |A_{\mathcal{S}}|$.

6.4.1 Model Characterisation

The procedure $Shrink()$ leads to some properties of $\mathcal{S}_*$. The first property of $\mathcal{E}$ is determinism.

Theorem 6.4.3. $\mathcal{S}_*$ *is deterministic.*

The main purpose of $Shrink()$ is to reduce the size of $\mathcal{S}_*$. To characterise the size of $\mathcal{S}_*$ with respect to all correct refinement models, it needs several auxiliary definitions. Let $\sim$ be the $\sim$ after Algorithm 3 returns $\longrightarrow_{\mathcal{S}_*}$. Let $\sim_{\mathcal{M}} = \{\langle s, s'\rangle \in S_{\mathcal{M}} \times S_{\mathcal{M}} \mid v \sim s, v \sim s', v \in S_{\mathcal{S}_*}\}$. Let $\mathcal{S}' = (S_{\mathcal{S}'}, \iota_{\mathcal{S}'}, A_{\mathcal{S}'}, \longrightarrow_{\mathcal{S}'})$. Suppose $\mathcal{S}' \preccurlyeq \mathcal{M}$ with the refinement relation $\sim'$ and $\mathcal{S} \subseteq \mathcal{S}'$. We say $\sim'$ *respects* $\sim_{\mathcal{M}}$, if $u \sim' s, u \sim' s'$ implies $s \sim_{\mathcal{M}} s'$ for each $u \in S_{\mathcal{S}'}$, $s, s' \in S_{\mathcal{M}}$.

Theorem 6.4.4. *If* $\sim'$ *respects* $\sim_{\mathcal{M}}$, *then* $| \longrightarrow_{\mathcal{S}_*} | \leq | \longrightarrow_{\mathcal{S}'} |$.

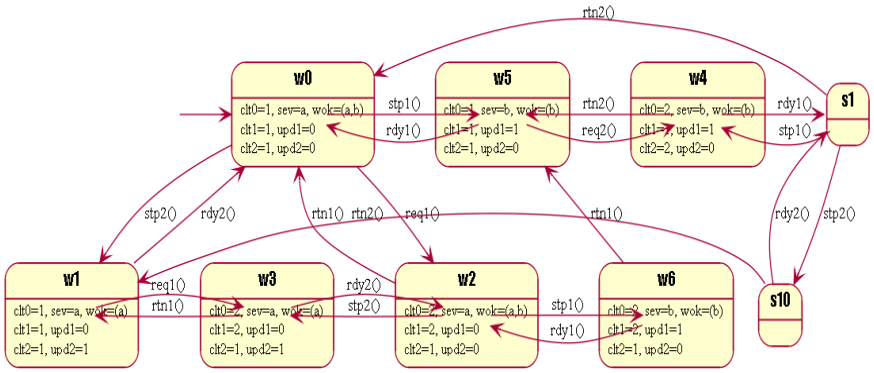


Figure 6.2: Final behavioural specification $\mathcal{S}_{cs*}$ for *AlterCS*

A natural question is that whether Theorem 6.4.4 can be generalised by removing the condition that $\sim'$ respects $\sim_{\mathcal{M}}$. The answer is, unfortunately, negative. In other words, Algorithm 3 does not necessarily generate a minimal model in all possible refinement models of the given MTS w.r.t. the given constraints. To compute such a minimal model, an algorithm needs to check whether any two nodes in the given MTS refine each other. But this checking is not performed in Algorithm 3.

6.4.2 A Merging Example

In the following, the application of the merging algorithm to *AlterCS* is illustrated. The algorithm takes $\mathcal{M}_{cs}$ and $\mathcal{S}_{cs}$ as input and produces an LTS, say, $\mathcal{S}_{*cs}$, which is depicted in Fig. 6.2. In general, the refinement algorithm does not necessarily generate a unique model up to the difference of node names. But the several observations below do not depend on the uniqueness of $\mathcal{S}_{cs*}$.

Firstly, $\mathcal{M}_{cs} \preccurlyeq \mathcal{S}_{cs*}$, as only a part of transitions in $\mathcal{M}_{cs}$ are inherited by $\mathcal{S}_{cs*}$. Secondly, the majority of nodes are annotated with information about the values of state variables, which is identified to $\mathcal{S}_{cs}$. Thirdly, some transitions in $\mathcal{S}_{cs*}$, e.g. $\langle w_5, rdy1(), w_0\rangle$ and $\langle w_2, rtn1(), w_0\rangle$, are not derivable from $\mathcal{S}_{cs}$, because PPCs of some actions, e.g. $rdy1()$ and $rtn1()$, are not contained in the constraints for *AlterCS* (Fig. 6.1). Transitions such as the previous two are added into $\mathcal{S}_{cs*}$ based on the transitions in $\mathcal{M}_{cs}$. Finally, as two nodes remain non-annotated, a good opportunity is given to practitioners to further elaborate the specification

by filling in those blanks.

6.5 Related Work and Discussions

This chapter aims to complement the use of process algebraic or state-machine-based formalisms in architectural description, by bridging the gap between their low-level formal syntax and the coarse-grained system specifications. The solution in this chapter is an automated method to merge deterministic MTSs and the constraint-style specifications. In general, the method elaborates the MTS by information about state variables extracted from the set of constraints and produces a refinement of the MTS.

The practical significance of this paper assumes the applicability of behaviour model synthesis from scenarios. Among different synthesis approaches, the MTS synthesis is extensively exploited. For example, Uchitel et al. [2007] proposed an approach to synthesising MTSs from both scenarios and safety properties expressed in a three-value temporal logic. In particular, they ground synthesis for scenarios on existing LTS synthesis and restrict the scenarios being existential as opposed to the universal ones presented in Fig. 2 for our running example. Sibay et al. [2008] introduced a novel scenario description language called epLSC that is able to express not only the existential and universal scenarios and also the universal existential ones, and also developed an MTS synthesis algorithm for this new language.

In order to perform the MTS synthesis effectively, both aforementioned works need to merge two or more MTSs. The merging methods of MTSs, i.e. the produce of a minimum common refinement model of two MTS, was studied by Uchitel and Chechik [2004]. A restriction in their work is the supposition of two merged MTSs must share the same alphabet. A more general approach to MTS merging that overcomes this restriction was studied in detail by Fischbein et al. [2012]. Both model merging and our approach address refinement problem for MTSs. But we refine an MTS by constraints instead of another MTS and our finial product is a complete behaviour model eLTS. Alrajeh et al. [2011] proposed another formal method for refining MTSs. Their method assumes an MTS synthesised from safety properties expressed in a temporal logic called FLTL

and a set of positive and negative traces that belong to the possible traces of the MTS. Then, it learns an FLTL safety property from the traces and uses this property in conjunction with the original ones to generate a new MTS which is a strong refinement of the previous one.

Incorporation of PPCs into the behaviour model synthesis was studied by Whittle and Schumann [2000] because, as they argued, PPCs provide semantic information that facilitate the interpretation of scenarios, conflict detection and loop identification. But their algorithm generates a final LTS and fails to support the partiality of specification. Krka et al. [2009] proposed an algorithm generating component-level rather than system-level MTSs from basic forms of scenarios and PPCs. There are a few differences between their use of PPCs and ours. First, the use of PPCs by Krka et al is based on the necessary perspective, in which the satisfaction of pre-conditions must be guaranteed in order to trigger the actions or events. By contrast, we adopt the sufficient perspective for PPCs, in which the satisfaction of pre-conditions guarantees to trigger the guarded actions or events. Also, PPCs are the basis of their synthesis algorithm in their work, but are integrated with MTSs rather than scenarios in our approach. Finally, the form of PPCs treated in this paper are more general than that in theirs.

A merging algorithm that handles arbitrary MTSs is possible but left in the future work. It is informally claimed that this can be achieved at the price of an algorithm with higher complexity. Moreover, determinism should not be seen as a very serious restriction, because one can refine a non-deterministic MTS to obtain a deterministic one before combining it with constraints.

Chapter 7

Conclusions

Motivated by the diversity of ADLs and the popularity of PA in the area of software architectural description, this thesis investigates the use of PA as formal notations in the modelling and analysis of the connector-based and peer-to-peer architectures, independent of the high-level specification languages. This goal is fulfilled in three parts, i.e. Chapters 3 to 5. A complementary part (Chapter 6) of this thesis is towards a method to bridge the gap between the low-level PA syntax and the coarse-grained architectural-level specifications. The remainder of this conclusive chapter is organised as follows. First, the approaches in Chapters 3 to 6 are reviewed. Then, the limitations of the approaches are discussed and several directions for further work are outlined.

Chapter 3 is dedicated to the connector-based architecture, in which the components communicate with each other via the central connector. Therefore, the connector-based architecture always is an instance of the star topology. For the reusability and scalability of specifications, it is advantageous to describe the connector in a way such that it is regardless of the number of the attached components. This motivates an ADL-like language, called ACDL. The most important feature of ACDL is its separation of ports of components into the data ports and the control ports, the former of which are for the data transit between components, whilst the latter of which are for controlling the exact transit topology. The semantics of ACDL is version of PA and a formal translation from ACDL to the PA notation is provided. Based on the PA notation, a variety of analytic techniques are developed to facilitate the checking of the properties of deadlock-

freedom and non-starvation of the architecture. The significance of these techniques, some of which are abased on additional assumptions, is to show how the verification of the two properties can be carried out in a compositional way and leveraged from the instance-layer description to the type-layer description in ACDL.

Chapter 4 also deals with the connector-based architecture. Instead of proposing an ADL-like language, the approach in Chapter 4 is purely semantic. More specifically, a process-algebraic semantic model is presented for the same architecture. The semantic model is free of the syntactic restriction imposed by ACDL and the semantic restriction imposed by the translation from ACDL into its underpinning PA notation, and hence, more flexible than ACDL in the modelling aspect. For example, the semantic model can be used to describe the dynamic connector-based architecture, in which the runtime reconfiguration of the architecture is equivalent to the insertion and removal of the components with respect to the connector (this feature is failed to be captured by ACDL). Another aspect of extensions is that the component conformance in the semantic model is defined in terms of not only parameterisation but also behavioural conformance. In the analysis, the developed techniques demonstrate that the verification state spaces of the aforementioned two properties (i.e. deadlock-freedom and non-starvation) and two additional important properties (i.e. conservation and completeness) can be reduced from the universal of the possible runtime configurations to a fixed configuration. Also, some compositional checking techniques are developed for conservation and completeness.

Chapter 5 introduces an approach based on the theory of session types for PA to deal with the peer-to-peer architecture, which the previous two approaches cannot handle. As the modularised behavioural types for PA, session types are used as the protocol-based interaction specifications of the components in the architecture. Session types describe the interactions of the components from the global point of view and facilitate the design and verification of the specifications. Also, session types project fragments of the protocols as the usage of private channels and ensure the correctness of channel (or port) binding between the components in architectural description. This chapter presents a PA notation with session establishment primitives and develops a session type discipline

for the notation. It investigates several key properties of the type discipline and establishes a channel privacy property (for correct binding), which states that each channel is shared to a unique pair of processes (or components), and a behavioural conformance property, which states that the behaviours of processes (or components) conform to the session type-based specifications. Finally, a process slicing method is presented to improve the type checking.

The work in Chapter 6 complements the previous chapters. This chapter aims to bridge the gap between the low-level syntax of PA notations and the coarse-grained specifications in architectural description and, thereby, enhances the use of PA notions as *ad hoc* architectural modelling formalisms. More specifically, this chapter presents an automated method to merge Modal Transition Systems (MTSs), which is equivalent to some modal version of PA, and the constraint-style specifications.

Overall, this thesis conducts an *in-depth* study on the use of various versions of PA as formal notions to describe and analyse two basic architectural models, i.e. the connector-based architecture and the peer-to-peer architecture. Main contributions of the study include a group of techniques that facilitate checking the behavioural properties of the architecture and a new perspective to specify and analyse the peer-to-peer architecture. Also, a merging algorithm for MTSs and constraints is proposed to enhance the adequacy of PA as architectural modelling notations.

Nevertheless, the approaches in this thesis suffer several shortcomings and limitations, for which further improvement is needed. Firstly, although the integrity of the whole thesis is based on the investigation of PA in architectural modelling and analysis, one limitation is the lack of a unified approach to deal with both the connector-based architecture and the peer-to-peer architecture. From Chapter 3 to Chapter 5, different PA notations are used, and the selection of these notations is based on either capturing particular aspects of the modelled architecture or enabling particular analytic techniques, or both. Another shortcoming is that, in spite of a variety of analytic techniques presented in each chapter, it is hard to coin a conclusive and technical definition for the properties that have been analysed. In particular, two basic properties (deadlock-freedom and non-starvation) are treated in Chapter 3; in Chapter 4, three more important

properties (behavioural conformance, conservation, and completeness) are tackled; but only behavioural conformance and a channel safety property are dealt with in Chapter 5. It is desirable to have a unified approach, which contains a single PA notation (perhaps with session types) that is able to capture all features of interest of the modelled architectures in the way that different PA notations in this thesis do, and enables the analysis of all properties of interest in the way that these PA notations do. The optimistic fact is that, although not demonstrated formally, there is no incompatibility between these notations and, therefore, it can be possible to construct such a single PA notation in a further work.

Finally, in Chapter 6, a modal version of PA, which is equivalent to the model of MTSs, is presented and merged with constraints. Although MTSs are extensively studied in the literature, MTSs do not subsume *all* PA notations in the previous chapters. This is also a limitation of the merging method, which aims to improve the use of PA in architectural description. Hence, a direction of future work is to improve the method to deal with more expressive modal PA.

Bibliography

Gregory D. Abowd, Robert Allen, and David Garlan. Formalizing style to understand descriptions of software architecture. *ACM Transaction on Software Engineering Methodology*, 4(4):319–364, 1995. ISSN 1049-331X.

Alessandro Aldini and Marco Bernardo. On the usability of process algebra: An architectural view. *Theoretical Computer Science*, 335(2-3):281–329, 2005. 34

Robert Allen and David Garlan. A formal basis for architectural connection. *ACM Trans. Softw. Eng. Methodol.*, 6:213–249, July 1997. ISSN 1049-331X. 6, 33, 63, 67, 85, 104

Dalal Alrajeh, Jeff Kramer, Alessandra Russo, and Sebastian Uchitel. An inductive approach for modal transition system refinement. In Michael Gelfond John P. Gallagher, editor, *Technical Communications of the 27th International Conference on Logic Programming, ICLP 2011*, pages 106–116. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2011. URL http://publicaciones.dc.uba.ar/Publications/2011/AKRU11. ICLP 2011, July 6-10, 2011, Lexington, Kentucky, USA. 119

Suzana Andova, Luuk Groenewegen, and Erik P. de Vink. Dynamic consistency in process algebra: From paradigm to acp. *Sci. Comput. Program.*, 76(8): 711–735, 2011. 9, 64

F. Arbab. Reo: a channel-based coordination model for component composition. *Mathematical Structures in Computer Science*, 14(3):1–38, 2004.

M. Ali Babar, C. Cuesta, J. Savolainen, and T. Männistö, editors. *2012 Joint Working IEEE/IFIP Conference on Software Architecture and European Con-*

ference on Software Architecture, WICSA/ECSA'12 2012, 2012. IEEE. ISBN 978-1-4673-2809-8. 1

J. C. M. Baeten. A brief history of process algebra. *Theor. Comput. Sci.*, 335 (2-3):131–146, May 2005. ISSN 0304-3975. 2, 5

Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008. ISBN 026202649X, 9780262026499.

Olivier Barais, Anne Franoise, Le Meur Laurence Duchien, and Julia Lawall. Software architecture evolution. In *Software Evolution*, pages 233–262, 2008.

H.P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic 103*. North-Holland, Amsterdam, the second, revised edition edition, 1984. 5

J. A. Bergstra, A. Ponse, and Scott A. Smolka, editors. *Handbook of Process Algebra*. Elsevier Science Inc., New York, NY, USA, 2001. ISBN 0444828303. 2, 5, 127, 132

Jan A. Bergstra and Jan Willem Klop. Algebra of communicating processes with abstraction. *Theor. Comput. Sci.*, 37:77–121, 1985. 5

Marco Bernardo, Paolo Ciancarini, and Lorenzo Donatiello. Architecting families of software systems with process algebras. *ACM Transactions on Software Engineering and Methodology*, 11(4):386–426, 2002. ISSN 1049-331X. 6, 8, 25, 33, 63, 64, 67, 85

Tommaso Bolognesi and Ed Brinksma. Introduction to the iso specification language lotos. *Computer Networks*, 14:25–59, 1987. 5

Grady Booch, James Rumbaugh, and Ivar Jacobson. *Unified Modeling Language User Guide (2nd Edition)*. Addison-Wesley, 2005. ISBN 0321267974. 12

E. Börger and R. F. Stärk. *Abstract State Machines – A Method for High-Level System Design and Analysis*. Springer-Verlag, 2003. 42, 46

R. Bruni, I. Lanese, and U. Montanari. A basic algebra of stateless connectors. *Theoretical Computer Science*, 366(1):98–120, 2006.

Carlos Canal, Ernesto Pimentel, and José M. Troya. Specification and refinement of dynamic software architectures. In *WICSA*, pages 107–126, 1999. 7

Carlos Canal, Ernesto Pimentel, and Jos M. Troya. Compatibility and inheritance in software architectures. *Science of Computer Programming*, 41:2001, 2001. 8, 64

Marco Carbone, Kohei Honda, and Nobuko Yoshida. Structured communication-centred programming for web services. In *ESOP*, pages 2–17, 2007. 10, 102

Giuseppe Castagna and Luca Padovani. Contracts for mobile processes. In *CONCUR*, pages 211–228, 2009. 103, 104

R. Cleavelandy and O. Sokolsky. Equivalence and preorder checking for finite-state systems. In Bergstra et al. [2001], chapter 6, pages 391–425. ISBN 0444828303. 5

Carlos E. Cuesta, Pablo de la Fuente, Manuel Barrio-Solórzano, and M. Encarnación Beato. An 'abstract process' approach to algebraic dynamic architecture description. *The Journal of Logic and Algebraic Programming*, 63(2):177 – 214, 2005. 8, 64

Luca de Alfaro and Thomas A. Henzinger. Interface automata. In *Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*, ESEC/FSE-9, pages 109–120, New York, NY, USA, 2001. ACM. ISBN 1-58113-390-1.

Romain Demangeon and Kohei Honda. Nested protocols in session types. In Maciej Koutny and Irek Ulidowski, editors, *CONCUR'12*, volume 7454 of *Lecture Notes in Computer Science*, pages 272–286. Springer Berlin Heidelberg, 2012. doi: 10.1007/978-3-642-32940-1_20. 10, 103

Pierre-Malo Deniélou and Nobuko Yoshida. Dynamic multirole session types. In *POPL*, pages 435–446, 2011. 10, 103

Joost Engelfriet and Tjalling Gelsema. The decidability of structural congruence for replication restricted pi-calculus processes. *LIACS Technical Report*, 2004. 70

Jos Luiz Fiadeiro, Antnia Lopes, and Michel Wermelinger. A mathematical semantics for architectural connectors. In *Proceedings of Colloquium on Formal Approaches in Software Engineering, Joint Conference on Theory and Practice of Software Development*, pages 505–519, 1997.

Dario Fischbein, Nicolás D'Ippolito, Greg Brunet, Marsha Chechik, and Sebastián Uchitel. Weak alphabet merging of partial behavior models. *ACM Trans. Softw. Eng. Methodol.*, 21(2):9, 2012. 11, 119

David Garlan and Mary Shaw. An introduction to software architecture. Technical report, Pittsburgh, PA, USA, 1994.

Simon J. Gay and Malcolm Hole. Types and subtypes for client-server interactions. In *ESOP*, pages 74–90, 1999. 10

Luuk Groenewegen and Erik P. de Vink. Operational semantics for coordination in paradigm. In *COORDINATION*, pages 191–206, 2002. 9

Jan Friso Groote, Aad Mathijssen, Michel A. Reniers, Yaroslav S. Usenko, and Muck van Weerdenburg. The formal specification language mcrl2. In *MMOSS*, 2006. 9

David Harel and Rami Marelly. *Come, Let's Play: Scenario-Based Programming Using LSC's and the Play-Engine*. Springer-Verlag, 2003. ISBN 3540007873. 106

Matthew Hennessy. *A distributed Pi-calculus*. Cambridge University Press, 2007. 69

C. A. R Hoare. *Communicating Sequential Processes*. Prentice-Hall, NJ, USA, 1985. 5

Kohei Honda, Vasco Thudichum Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In *ESOP*, pages 122–138, 1998. 9

Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. In *POPL*, pages 273–284, 2008. 10, 69, 102

Hans Hüttel and Kim Guldstrand Larsen. The use of static constructs in a modal process logic. In *Logic at Botik*, pages 163–180, 1989. 11

IEEE Recommended Practice for Architectural Description of Software-Intensive Systems. IEEE, 2000. 1

Paola Inverardi, Alexander L. Wolf, and Daniel Yankelevich. Static checking of system behaviors using derived component assumptions. *ACM Transactions on Software Engineering and Methodology*, 9(3):239–272, 2000. ISSN 1049-331X. 7, 34, 64

ITU. *Recommendation z.120: Message sequence charts*, 2000. 12, 106

Ivo Krka, Yuriy Brun, George Edwards, and Nenad Medvidovic. Synthesizing partial component-level behavior models from system specifications. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ESEC/FSE '09, pages 305–314, New York, NY, USA, 2009. ACM. 13, 120

K.G. Larsen and B. Thomsen. A modal process logic. In *Logic in Computer Science, 1988. LICS '88., Proceedings of the Third Annual Symposium on*, pages 203 –210, 0-0 1988. doi: 10.1109/LICS.1988.5119. 11

Jeff Magee, Naranker Dulay, Susan Eisenbach, and Jeff Kramer. Specifying distributed software architectures. In *Proceedings of the 5th European Software Engineering Conference*, pages 137–153, 1995. ISBN 3-540-60406-5. 8, 34, 37

Jeff Magee, Jeff Kramer, and Dimitra Giannakopoulou. Behaviour analysis of software architectures. In *WICSA*, pages 35–50, 1999. 8

N. Medvidovic and R.N. Taylor. A classification and comparison framework for software architecture description languages. *Software Engineering, IEEE Transactions on*, 26(1):70 –93, jan 2000. 1

Nenad Medvidovic and Richard N. Taylor. Software architecture: foundations, theory, and practice. In *ICSE (2)*, pages 471–472, 2010. 1

Nenad Medvidovic, David S. Rosenblum, and Richard N. Taylor. A language and environment for architecture-based software development and evolution. In *Proceedings of the 21st international conference on Software engineering*, ICSE '99, pages 44–53, New York, NY, USA, 1999. ACM. ISBN 1-58113-074-0. 37

Nenad Medvidovic, David S. Rosenblum, David F. Redmiles, and Jason E. Robbins. Modeling software architectures in the unified modeling language. *ACM Transaction on Software Engineering Methodology*, 11(1):2–57, 2002. ISSN 1049-331X. doi: http://doi.acm.org/10.1145/504087.504088.

Nikunj R. Mehta, Nenad Medvidovic, and Sandeep Phadke. Towards a taxonomy of software connectors. In *Proceedings of the 22nd international conference on Software engineering*, ICSE '00, pages 178–187, New York, NY, USA, 2000. ACM. ISBN 1-58113-206-9.

R. Milner. *Communication and Concurrency*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989. ISBN 0-13-115007-3. 5

Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, i. *Inf. Comput.*, 100(1):1–40, 1992a. 5

Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, ii. *Inf. Comput.*, 100(1):41–77, 1992b. 5

Flavio Oquendo. 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. ISSN 0163-5948. 8, 34, 64

Luca Padovani. On projecting processes into session types. *Mathematical Structures in Computer Science*, 22(2):237–289, 2012. 10, 103

Davide Sangiorgi and David Walker. *π-calculus: A Theory of Mobile Processes*. Cambridge University Press, NY, USA, 2001. ISBN 0521781779. 5

M. Shaw. Procedure calls are the assembly language of system interconnection: Connectors deserve first-class status. In *Proceedings of the Workshop on Studies of Software Design*, 1993. 14

M. Shaw and P. Clements. The golden age of software architecture. *IEEE Software*, 23(2):31–39, 2006. ISSN 0740-7459. doi: 10.1109/MS.2006.58. 1

Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall, NJ, USA, 1996.

Junrong Shen, Xi Sun, Gang Huang, Wenpin Jiao, Yanchun Sun, and Hong Mei. Towards a unified formal model for supporting mechanisms of dynamic component update. In *ESEC/SIGSOFT FSE*, pages 80–89, 2005. 9

G. Sibay, S. Uchitel, and V. Braberman. Existential live sequence charts revisited. In *Software Engineering, 2008. ICSE '08. ACM/IEEE 30th International Conference on*, pages 41 –50, may 2008. 119

J. Sifakis. A framework for component-based construction. In *Proceedings of the 3rd IEEE International Conference on Software Engineering and Formal Methods*, 2005.

B. Spitznagel and D. Garlan. A compositional formalization of connector wrappers. In *Proceedings of the 25th International Conference on Software Engineering*, 2003.

Guoxin Su, Mingsheng Ying, and Chengqi Zhang. An adl-approach to specifying and analyzing centralized-mode architectural connection. In *Proceedings of the 4th European conference on Software architecture*, ECSA'10, pages 8–23, Berlin, Heidelberg, 2010. Springer-Verlag. 4, 14, 57, 64

Guoxin Su, Mingsheng Ying, and Chengqi Zhang. Semantic analysis of component-aspect dynamism for connector-based architecture styles. In *WICSA/ECSA*, pages 151–160, 2012a. 4, 85

Guoxin Su, Mingsheng Ying, and Chengqi Zhang. Session communication and integration. *CoRR*, abs/1210.2125, 2012b. 4

Massimo Tivoli and Paola Inverardi. Failure-free coordinators synthesis for component-based architectures. *Science of Compututer Programming*, 71(3): 181–212, 2008. ISSN 0167-6423. doi: http://dx.doi.org/10.1016/j.scico.2008.03.001. 34

Sebastián Uchitel and Marsha Chechik. Merging partial behavioural models. In *SIGSOFT FSE*, pages 43–52, 2004. 11, 119

Sebastián Uchitel, Jeff Kramer, and Jeff Magee. Behaviour model elaboration using partial labelled transition systems. In *ESEC / SIGSOFT FSE*, pages 19–27, 2003. 13

Sebastian Uchitel, Greg Brunet, and Marsha Chechik. Behaviour model synthesis from properties and scenarios. In *Proceedings of the 29th international conference on Software Engineering*, ICSE'07, pages 34–43, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2828-7. 119

R. J. van Glabbeek. The linear time - branching time spectrum i. - the semantics of concrete, sequential processes. In Bergstra et al. [2001], chapter 1, pages 3–100. ISBN 0444828303. 5

Jos Warmer and Anneke Kleppe. *The Object Constraint Language: Getting Your Models Ready for MDA*. Addison-Wesley, Boston, MA, USA, 2 edition, 2003. ISBN 0321179366. 12

Jon Whittle and Johann Schumann. Generating statechart designs from scenarios. In *ICSE*, pages 314–323, 2000. 12, 13, 120

Daniel M. Yellin and Robert E. Strom. Protocol specifications and component adaptors. *ACM Trans. Program. Lang. Syst.*, 19:292–333, March 1997. ISSN 0164-0925.

Nobuko Yoshida, Pierre-Malo Deniélou, Andi Bejleri, and Raymond Hu. Parameterised multiparty session types. In *FOSSACS*, pages 128–145, 2010. 10, 92, 103