Università degli Studi di Pisa

Dipartimento di Informatica
Dottorato di Ricerca in Informatica

Ph.D. Thesis

A Theory of Interface Modeling of Component-Based Software

Ruzhen Dong

Supervisor	Supervisor
Zhiming Liu	Carlo Montangero

May 26, 2015

Pisa, Italy

Abstract

In component-based software engineering, large software systems are decomposed into components with clearly described interfaces in order to make developers from different teams work effectively together. The interfaces specify the assumptions and guarantees of the interaction behavior by declaring functional and non-functional properties. More specifically, we work within the Refinement Calculus of Component and Object Systems (rCOS), a formal framework and methodology for component-based development, which supports separation of concern, component design at different levels, and model-driven development. The components we consider are open, in the sense that components may require services from the environment in order to provide their own services.

Deadlock is difficult to check and avoid when composing components, because the open components are composed based on the assumption/guarantee relation of their provided and required services. Game semantics may be used to provide intuitive understanding for the assumption/guarantee relation, but there is little support for implementation of components specified in game semantics. A well established denotational model of components is needed to describe what the components can provide and require, and also which services may cause deadlock. In order to improve the techniques for checking whether two components can be composed without causing deadlocks, we present a new automata-based approach to model the dependency relation of the invocations to provided and required services. We develop an interface model, called *input deterministic automata*, which defines all the unblockable sequences of invocation to the services provided by a component. We also present an algorithm that, for any given component automaton, generates the interface model that has the same input deterministic behaviors. Correctness of the algorithm is proved.

The automata-based model provides an operational and intuitive description of the interaction behaviors of components. A denotational trace-based model of components is also given, inspired by the failure divergence semantics in CSP. Similar to CSP, a failure in the trace-based model is a pair of an alternating sequence of invocation to provided and required services, and a set of invocations to the provided services that may not be blocked after the alternating sequence.

Components need to be adapted for reuse in different contexts. A coordinator component is introduced to adapt a component by filtering out the provided services. The composition operation, called coordination, is also defined. An algorithm is developed to synthesize a coordinator, for any given component automaton, to obtain the interface model of the coordinator-automaton composition.

Components are often replaced by new, better ones, with more provided and less required services. To guarantee the correctness of such a substitution, we introduce a suitable refinement notion, for both the automata-based and trace-based models. Finally, we prove that the automata-based refinement entails the trace-inclusion one.

Contents

1 Introduction **11**

1.1 Motivation 12

1.2 Related Work 16

1.3 Contribution 19

1.4 Organization of the Thesis 20

1.5 Origins of the Chapters 21

2 rCOS **23**

2.1 Unified Semantics of Sequential Programming 24

2.1.1 **Designs of Sequential Programs** 24

2.1.2 Reactive Systems and Reactive Designs 29

2.2 Closed Components 35

2.2.1 Specification Notation for Primitive Closed Components . . . 37

2.2.2 Labeled Transition Systems of Primitive Closed Components . 38

2.2.3 Component Contracts 41

2.2.4 Refinement between Closed Components 42

2.3 Open Components 44

2.3.1 Specification of Open Components 45

2.3.2 Semantics and Refinement of Open Components 46

2.3.3 Transition Systems 48

2.4 Processes 49

2.4.1 Specification of Processes 50

2.4.2 Contracts of Processes 50

2.4.3 Transition Systems 52

2.5 Summary and Discussion 53

3 Automata-based Model of Components 55

3.1 Background 56

3.1.1 Finite State Machine 56

3.1.2 Regular sets 57

3.2 Component Automata 58

3.3 Component Interface Automata 66

3.3.1 Motivation 66

3.3.2 Non-blockableness 66

3.3.3 Component Interface Automata 70

3.4 Composition Operation 80

3.4.1 Product of Component Automata and Finite State Machine . 81

3.4.2 Product of Component Automata 85

3.4.3 Plugging 89

3.4.4 Composition of Component Interface Automata 91

3.4.5 Hiding and Renaming 93

3.5 Refinement 93

3.6 Discussion and Conclusion 103

3.6.1 Internal Transitions 103

3.6.2 Substitution 105

3.6.3 Summary 105

4 Trace-based Model of Components 107

4.1 Trace-based Model 107

4.1.1 Failure Model of Components 109

4.1.2 Plugging Operation 111

4.2 Refinement 112

4.3 Summary 113

5 Coordination 115

5.1 Process Component 115

5.1.1 Process Automata 116

5.1.2 Coordination 116

5.2 Coordinator Component 117

5.2.1 Coordination Operation 120

5.2.2 Synthesizing Interface Coordinator for Component Automata . 121

5.3 Summary 124

6 Conclusion and Future Work 127

Bibliography 129

Notations

These following notations are used through the thesis.

For any $w_1, w_2 \in \mathcal{L}^*$, the sequence concatenation of w_1 and w_2 is denoted as $w_1\hat{\ }w_2$ an $A \circ B$ is $\{w_1\hat{\ }w_2 \mid w_1 \in A,\ w_2 \in B\}$ where $A, B \subseteq \mathcal{L}^*$ are two sets of sequences of elements from $\mathcal{L}$.

Given a sequence of sets of sequences $\langle A_1, \ldots, A_k\rangle$ with $k \geq 0$, we denote $A_1 \circ \cdots A_k$ as $conc(\langle A_1, \ldots, A_k\rangle)$. ϵ is used as notion of empty sequence, that is, $\epsilon\hat{\ }w = w\hat{\ }\epsilon = w$.

Let ℓ be a pair (x, y), we denote $\pi_1(\ell) = x$ and $\pi_2(\ell) = y$. Given any sequence of pairs $tr = \langle \ell_1, \ldots, \ell_k\rangle$ and a set of sequences of pairs T, it is naturally extended that $\pi_i(tr) = \langle \pi_i(\ell_1), \ldots, \pi_i(\ell_k)\rangle$, $\pi_i(T) = \{\pi_i(tr) \mid tr \in T\}$ where $i \in \{1, 2\}$.

Let $tr \in A$ and $\Sigma \subseteq \mathcal{L}$, $tr \upharpoonleft \Sigma$ is a sequence obtained by removing all the elements that are not in Σ from tr. And we extend this to a set of sequences $T \upharpoonleft \Sigma = \{tr \upharpoonleft \Sigma \mid tr \in T\}$. Similarly, $tr \downharpoonleft \Sigma$ is a sequence obtained by removing all the elements in Σ and $T \downharpoonleft \Sigma = \{tr \downharpoonleft \Sigma \mid tr \in T\}$.

Given a sequence of pairs tr, $tr \upharpoonleft 1P$ is a sequence obtained by removing the elements whose first entry is not in P. For a sequence of elements $\alpha = \langle a_1, \cdots, a_k\rangle$, $pair(\alpha) = \langle (a_1, \{a_1\}), \cdots, (a_k, \{a_k\})\rangle$.

Chapter 1

Introduction

There is an increasing demand for software systems in modern societies. They are widely used in our daily life and work, for example, control systems of aircraft, trains and cars, e-banking, office system, and various applications on PCs and mobile devices, etc. Despite a lot of progress made in programming techniques and tools, developing reliable software and delivering the implementation in time is still a challenge, due to the growing complexity of software systems. Component-based and model-driven methodologies integrated with sound systematic formal semantics and automatic tools are promising approaches to tackle these problems.

Model-Driven Development In the model-driven architecture (MDA) approaches [Obj01, WBHR08, Szy02], software artifacts are treated as *models* in every stage of software development, such as requirements, design, coding, testing, deploying, and maintenance. The focus is shifted from program codes to behavioral models, which are easier to be specified and understood by domain experts and software developers. The model-driven methodology is to develop software with models as basis and automatic model transformation techniques to refine high level specifications into machine code stepwise. Especially, there is great tool support of MDA for the modeling, developing, validation and verification. In a software development, the models are defined as *platform independent* and *platform specific* . A PIM is a model of a software system, independent of programming languages, operating systems, and hardware, while a PSM is a model of a software system, that is built for a specific platform. Separating PIM and PSM facilitates model reuse for different platforms by transforming PIMs to PSMs with automatic model transformation techniques.

Component-Based Software Engineering In component-based software engineering (CBSE) [Szy02], software systems or components are built by composing smaller components which may be developed by different teams of developers or reused from existing components. To support this aim, clearly defined interfaces describing how the components can be used are the key for component reuse. The MDA of component-based software with sound formal semantics support is an effective solution to handling the software complexity, predictability, and correctness. CBSE is widely accepted and used in industry on popular platforms like JavaBean [BMH06], Common Object Request Broker Architecture (CORBA) [Gro06], and Microsoft Component Object Model [Box98].

Recent survey papers [LW07, CSVC11] on software components show that, up to now, a formal common and sufficiently precise definition of software components is still a problem in CBSE community. The widely used definition given by Szyperski [Szy02] states that

> *" A Software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties."*

This definition implies that the key characteristic of a software component is the contractually specified interfaces that support third-party composition.

In order to develop reliable and reusable components, independent development, and compositional reasoning, in this thesis, we would develop an interface theory to support modeling, composition, refinement, and coordination.

1.1 Motivation

The interfaces play the key role in component-based development for third party composition, component reuse, independent development and substitutability. To fulfill this aim, interfaces should be enhanced with contracts. As in [HLL$^+$12], contracts can be classified into the following four levels:

1. Syntactic level: describes signatures of the methods or services assuring they communicate through allowable methods and permissible data types of possible parameters, and type-checking techniques can verify and guarantee the compatibility at this level [Pie02].

2. Functionality level: describes what the method does functionally, which can be checked assured by pre/post-conditions and invariants [HLL$^+$12].

3. Interaction level, or called communication level: shows the temporal order and dependence relation of service or method invocations, which can be modeled by labeled transitions systems, finite state machine, or process algebras for different abstract levels and verification techniques.

4. Extra-functional level: or called quality-of-service, describes the performances of services, which can be modeled in queue theory and timed automata [AD94].

In this thesis, we consider the interfaces at the interaction level, modeling the protocols of communication between components, for instance, the order of method invocation. Violation of this protocol would cause incompatibility, for example, the method *readFile()* cannot be called before *openFile()* in a file system and the *withDraw()* service must be disabled to the clients before the credit card is checked to be *valid* in e-banking component. Programming libraries, like Java APIs, are kinds of components that are widely used for building software systems.

The above examples are components that only provide methods and this kind of components are called *closed components.* However, when components provide services to the environment, during which the component may need to require services from the environment too. Thus, components are called *open* if they provide methods and also require methods of other components. Hence, closed components are special cases of open components. We will call the open components as components in this thesis. Components act as service providers in the sense that the aim is to specify a set of methods that can be called by others and make the requirement as less as possible. In general, a component has *provided* and *required* interfaces which specify what the component provides to and requires from the environment, which is the context for the component, such as the client and other components. A component can be viewed as is in Fig.1.1. The provided interface specifies methods, which are called *provided services* in this thesis. The required interface specifies methods that the component needs, which are called *required services.* Thus, a component interact with the environment by providing services while requiring services and this is also the basis for the component composition. *Interaction protocol* describes the order of the service invocations, and given any allowable order of provided services, it is important to know the requirement in order to support independent development. And the sequence of provided methods specified in the interaction protocols should never be blocked during run-time.

Next, we will introduce the techniques for the interface theory we have developed in this thesis.

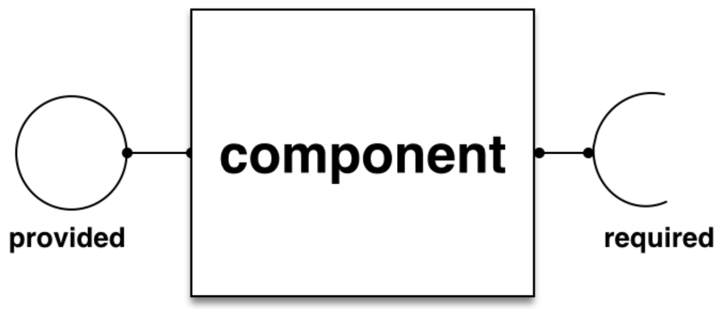


Figure 1.1: Component Interfaces

Automata-based models We adopt automata-based techniques for modeling interaction behaviors of components. The automata-based models are intuitive and can be directly used in model checking for verification.

In this thesis, we consider two models of components, that are *abstract model* and *interface model*. An abstract model describes possible interaction behaviors of the component, that is, sequences of invocations to the provided and required services of the components. However, a sequence of invocation to services of the abstract model may be blocked due to non-determinism, that is, the next allowable service is not determined after a certain sequence of service invocations. The automata-based abstract model is called *component automaton*. Invocation to provided and required services are modeled as *provided* and *required events*. The sequence of alternating provided and required events is *trace* of component automaton, and called *provided*,and *required trace* when the trace is projected into the set of provided and required events, respectively. A provided event is *non-blockable* at given state, if the component can always provide this service at this state.

Interface models The interface model is for the third party composition, so the interaction protocols specified in interface models should not contain sequences that may be blocked. In this thesis, we call this property as *interface property*, as it would support the non-blockable composition. A abstract model is also called an *interface model* of components, if it satisfies the interface property. The automata-based interface model is called *component interface automaton* in this thesis. A component automaton is *input-deterministic*, if after any sequence of invocation to provided services, the component will always reach states at which the set of non-blockable provided events are same. Input-determinism is then proved to be equivalent to the interface property. An algorithm is presented to check whether an abstract model satisfies the interface property. And another algorithm is developed to produce an interface model for any given abstract while preserving all the non-blockable behaviors.

Coordination The components we have discussed so far are *passive* in the sense that they act as service providers. They specify a set of provided services that can be invoked by the environment and the required services are enclosed inside the service body of provided services which are called by the component when the corresponding provided services are invoked. In order to make component reuse more flexible and adaptive, components should be able to be coordinated and constrained.

Consider a one-place buffer component which provides *put* and *get* and the interaction protocol is $(put \cdot get)^*$, which means *get* can be called after *put* recursively. In order to build a buffer with a larger capacity, i.e., two-place buffer, an active component is needed to invoke *get* from the first buffer and *put* of the second buffer internally. Such kinds of components are called *process components*(processes, for simplicity). Automata-based model of process is called *process automaton* and the alphabet of the label is *action*. Coordination of a component automaton by a process automaton is defined by internalizing the provided events of the component following control flow of the process. The result component automaton contains internal and anonymous events, which is a extension of the previous component automaton.

However, some components provide some sequence of provided services that are non-blockable, but in practice, these sequences should not be allowed. For instance, if an existing buffer component provides *put* and *get*, and it is developed in the way without any constraining on the temporal order of calling *put* and *get*. The interaction protocol is $(put + get)^*$. In order to build a more reasonable buffer based on this one, a component which constrains the sequence of provided services is needed. We call such component as *coordinator component*(coordinator for simplicity). Coordination is defined by filtering certain transitions of the component automaton by the coordinator.

Substitutability Component substitutability is another important issue in component based development for maintenance and incremental development. Trivial case is that a component can be replaced by a new implementation as long as the interfaces do not change, because components interact with each other via interfaces. A more complicated case is that a component is replaced by a component with a "better" interface model. In abstract models, we use term *refinement* for "better than" relation. It is challenging, especially for open component with both provided and required services. The intuitive idea is that a better interface model should be able to provide more services and require less. Alternating simulation technique [AHKV98, DAH01] is used to define such refinement relation for interface automata [DAH01]. The idea is that a refined interface simulates the other on the input and be simulated on the output by the other. We use simulation technique to define the refinement relation, but the difference is that the refined model should be better at avoiding deadlock and providing less blockable services.

Trace models Existing models describing interaction behaviors of components by capturing the operational steps of executions. During building the automata-based interface models, inspired by the failure-divergence semantics of CSP [Hoa85], we propose to describe interface models of components by traces, which are alternating sequences of invocation to provided/required traces and refusal sets of provided services. We aim to show the interface of components in a denotational perspective and some primary results are obtained, such as plugging composition is consistent with that of automata-based models.

1.2 Related Work

There are a lot of languages describing the interaction protocols of components such as ADLs (architecture description languages) [MT00], behavioral protocols of SOFA components [PV02], behavioral models of FRACTAL components [BHM05] and BIP [BMP$^+$07]. These models focus on describing the order of method invocation and support for composition and compatibility checking, however, the guarantee/assumption of provided and required methods between a component and its environment is implicit.

Interface models The most closely to work of interface models are the Input/Output(I/O) Automata [LT89, LT87] and the Interface Automata [DH01, DAH01, DAH05]. Their target focus on distributed embedded systems. While we focus on building interfaces to facilitate component compositions in the provided/required relation.

Our approach is positioned in between these existing approaches. I/O automata are defined by Lynch and Tuttle to model concurrent and distributed discrete event systems. The main features of the I/O automata are *input-enabledness* and *pessimistic* compatibility. The input-enabledness requires that all the input actions should be available at any state. The pessimism is that two components are compatible if there is no deadlock in the composition for any environment. On the contrary, our interface model does not require that all inputs are always enabled, because there are guards for provided services in software components, while the interface model is input-deterministic to guarantee that all the sequences of provided services with possible internal behaviors interleaved can never be blocked when the environment calls.

Alfaro and Henzinger introduce interface automata to model reactive components that have an assume-guarantee relation with the environment. Two components are compatible in an optimistic way in the sense that two components are compatible if there exists one environment that can make the composition avoid deadlock. This compatibility condition may be too relaxed since the usability of the provided service of a component depends not only on the components to be composed with and also the environment for the composition. To this end, Giannalopoulou et al [EGP08, GPB02] develop assume-guarantee rules and tools to reason about compatibility and weakest assumptions for any given interface automaton, while Larson et al [LNW06] present interface models for any given I/O automaton by splitting assumptions from guarantees. In contrast to these approaches, we present an interface model that directly specifies the unblockable sequences of provided services independent of the environment and develop an algorithm to generate such interface model based on the execution model of any given component.

In work [YS97], protocols specify the constraints of messages passing and method invocations, quite similar to interface automata. Compatibility is about deadlock free and subprotocols are defined based on similar idea of alternating simulation. However, internal events are not considered in this paper. Other loosely related work are architecture description languages (ADLs), Wright [AG97], a formal basis for architectural connection is defined by CSP expressions. Software architecture describes the elements from which system are built, interactions among these elements, pattern that guides their composition, and constraints [SG96]. ADLs (architectural description language) [AG97], most based on CSP [Hoa85] and CCS [Mil95], have been proposed as modeling components, interfaces,interaction, and constraints to support architecture-based development. A survey of ADLs (architecture description languages) can be found in [MT00]. ADLs model components and interactions, but they don't provide enough mechanism to support the third party composition for components.

The behavioral mode in paper [BHM05] presents hierarchical components of Fractal. The model describes both functional behavior and also life-cycle management of components. It will be growing complicated as components are composed and internal hierarchical structure is also described. This behavioral is difficult in supporting component reuse. The behavior protocols, expressed as a regular language like language, for Sofa components [PV02] describes the hierarchical structure of components. It contains framework protocol specifies the external behaviors and architecture protocol for internal synchronization between sub-components. It is mainly used to formal verification instead of compositions.

In the Behavior, Interaction, Priority (BIP) framework [BB13, BBS06, BS08, BSS09, Sif05, GS05, GGMC^{+}07, GS12], components into three layers: behaviors specified as a set of transitions, interaction about communication between behaviors, and priorities used to choose possible interactions. It mainly targets for the heterogeneous component. We share the idea of correctness by construction that certain properties, such as deadlock-freedom and liveness, should be preserved in the composition. However, our model focus on the provided/required relations between components and aim at solving the deadlock caused by non-determinism.

Coordination Reo is a channel-based framework for coordinating interactions among components [Arb04, JA12, BSAR06]. Port automata [Kra11][KC09] is the abstract version of constraints automata [BSAR06] without data constraints for the Reo. Transitions of the port automata models connector for synchronization behaviors of components by synchronizing the ports. And the transition with none port means the internal transition. Whether components are already implemented or in the design phrase, the constraints on the sequence of provided or required service invocations naturally exists. On the other hand, in the theory of interface-based design, software component is development by refining the interface into implementation according to the topdown development idea. Our model aims at showing how the components can be used due to business logic or detailed programming structure. The process automata coordinate component automata by actively invoke provided services and internalize the service. The coordinator automata is a filter to constrain the provided services.

Refinement There are two main techniques to define refinement for formal models. Failure-divergence refinement technique of CSP [Hoa85] and simulation techniques from CCS [Mil95]. The closest idea is alternating simulation in [DH01], which is a game semantics that a component and its environment. Besides a refined component should provide more and require less, we enhanced the simulation relation with *refusals* to trim out the transitions that are blockable. In models with modalities [LNW07b, LNW07a, RBB^{+}09, RBB^{+}11, LV12], transitions are differentiated by *may* and *must*, which may or must be implemented in the system. In summary, we define the refinement relation not only considering the alternating simulation between provided and required events, but also requiring that a refined interface should be better at avoiding failure and provide less services that are blockable.

1.3 Contribution

We present a novel interface modeling framework supporting the component-based development with the idea correct-by-construction and facilitate component reuse for software developers. The contributions of the thesis are summarized as follows.

– Based on the work of rCOS method, we define a full semantics model of rCOS components which are modeled as labeled state transition system, which is also a basis for the automata-based models.

– As to our knowledge, the automata-based model we develop is the first to explicitly describe the dependence relation between provided and required interfaces of software components. In this model, given sequence of provided services, it is easy to produce the requirement the component needs. This explicit structure of provided/required relation helps developing components according to the interface model.

– We contribute to the concept of *interface* by defining what we call *interface property*. We believe that as an interface model of components for third party composition, it should be able to guarantees non-blockableness during run time. That is, the sequences of provided services specified in the interface model can not be blocked as long as the requirement is satisfied. We also develop an algorithm to check whether an abstract model of components are suitable as an interface model for such component. Furthermore, we present an interface generating algorithm, which transforms any given abstract model of components into an interface model by disallowing the services that may be blocked. The composition of components is defined.

– We present a trace-based model describing behaviors and interface behaviors of components from the denotational perspective. This helps compatibility checking directly by trace inclusion and coincides with the automata-based model.

– We present the refinement relation for our interface models to support the independent implementation and substitutability of components by simulation and trace-inclusion techniques. The refinement focuses on non-blockable behaviors and guarantees less failure behaviors. This is quite contrary to intuitive idea that an abstract model should implement and refine the interface model, because component reuse is mainly based on refinement relation between interface models instead of abstract models.

– Process automaton is introduced to coordinate component automaton by actively internalizing provided events. A simple and easy-understanding coordinator component is introduced to coordinator the order of service invocation among components and also filter out certain services which may cause deadlock or violate the security policy. Components can be reused in various context with specific coordinators.

1.4 Organization of the Thesis

The rest of the thesis is organized in the following way.

In Chapter 2, we made a survey of the work on rCOS (the Refinement Calculus of Component and Object Systems) and have a systemic formal techniques in the use of model-driven development of component-based software system. A primary introduction of UTP, the basis of rCOS language, is introduced. Then we introduce the design of object-oriented and component-based programming with the semantics. Closed components are a package of codes with an interface, called *provided interface*, which specifies the functionality of the provided methods and also the protocol for using these methods. Closed components are self-contained and the implementation can directly provide services to the environment. On the contrary, open components extend closed components by required interfaces which contain services that services in the provided interfaces need in order to perform their job. The dependency relation between services in the provided and required interfaces are described.

Chapter 3 presents an automata-based model of components, in which invocation to services are modeled as events. Symbolic states and sets of available transitions are used to model the guards which control the accessibility to the provided services. We study the conditions under which a sequence of invocation to provided services can be blocked or not, and characterize the components which provide only non-blockable services as *interface models* or *component interface automata*. And the non-blockableness property is defined as the *interface property*, a criteria of interface models.

An algorithm is developed to produce, for any component automaton, a component interface automaton which preserves all the non-lockable behaviors of the original ones. Components are composed in such a way that they synchronize on the services that are provided by one and required by the other, and behave independently otherwise. We explicitly define a *compatibility property* as to whether two components are *compatible* or not. Refinement relation between component automata is defined by the state simulation technique in such a way that a refined component automaton can provide more and require less services, and therefore less likely cause deadlock or livelock.

A practical example of components Alarm and Timer is used through this chapter and shows the use of our model.

Chapter 4 gives a denotational description of components, called failure models, where components are specified by all the possible behaviors and the behaviors which can be blocked. We show that the failure models can be directly derived from the automata-based models. Plugging operation is defined for failure models and proved to be consistent with the plugging operation in component automata. Refinement of failure models is defined as trace inclusion.

Chapter 5 presents a coordinator component to coordinate service invocation among components by constraining the order of invocation to provided services which are independent with each other. A component may be reused by being coordinated by a coordinator according to the requirement of a new context. We also show that the component interface automata can be obtained by the coordination between the given component automaton and a synthesized coordinator. An algorithm is developed to synthesize such interface coordinator and correctness is proved.

Chapter 6, we summarize the main results of the thesis and discuss the possible research topics for future work.

1.5 Origins of the Chapters

Part of the material presented in this thesis has been published in some publications or has been submitted for publication, in details:

The summary and survey work of rCOS in Chapter 2 is published as a book chapter in [DFKL13]. The work of component automata, component interface automata, and the composition operations are published as conference papers [DFL$^+$12, DZZ13].

The denotational description of component-based software in Chapter 4 and coordination in Chapter 5 are published as conference paper [DZ14].

Chapter 2

rCOS

In this chapter, we will introduce the Refinement Calculus of Component and Object Systems(rCOS) based on a literature survey of the publications of last decade [CHL06, CZ06, CLM07, CLL$^+$07, CLS$^+$07, CLS08, CHH$^+$08, CMS09, CLR$^+$09, HLL05a, HLL05b, HLL06a, HLL06b, LH06, DFKL13] by the rCOS group lead by Zhiming Liu et al. rCOS is a formal framework which supports design and model-driven development in component-based software engineering. And I define the full semantics model of rCOS components and then derive a general labeled state transition system model of components, which is also a basis for the automata-based models in the next chapter. In the following part, we will first give a general background of rCOS and the theoretic basis, UTP. Then *closed* and *open* components are discussed respectively with the semantic model which is described as labeled state transitions system.

The Aim and Theme of rCOS The aim of the rCOS method is to bridge the gap between formal techniques, together with their tools, and their potential support to practical software development by defining the unified meanings of component-based architectures at different levels of abstraction in order to support seamless integration of formal methods in modeling software development processes.It thus provides support to MDA with formal techniques and tools for predictable development of reliable software. Its scope covers theories, techniques, and tools for modeling, analysis, design, verification and validation. A distinguishing feature of rCOS is the formal model of system architecture that is essential for model compositions, transformations, and integrations in a development process. This is particularly the case when dealing with safety critical systems (and so must be shown to satisfy certain properties before being commissioned), but beyond that, we promote with rCOS the idea that formal methods are not only or even mainly for producing software that is safety critical: they are just as much needed when producing a software system that is too complex to be produced without tool assistance. As it will be shown in this chapter, rCOS systematically addresses these complexity problems by dealing with architecture at a large granularity, compositionality, and separation of concerns.

2.1 Unified Semantics of Sequential Programming

The rCOS method supports programming software components that exhibit interacting behavior with the environment as well as local data functionality through the executions of operations triggered by interactions. The method supports interoperable compositions of components for that the local data functionality are implemented in different programming paradigms, including modular, procedural and object-oriented programming. This requires a unified semantic theory of models of programs. To this end, rCOS provides a theory of relational semantics for object-oriented programming, in which the semantic theories of modular and procedural programming are embedded as sub-theories. This section first introduces a theory of sequential programs, which is then extended by concepts for object-oriented and reactive systems.

To support model-driven development, models of components built at different development stages are related so that properties established for a model at a higher level of abstraction are preserved by its lower level refined models.

2.1.1 Designs of Sequential Programs

We first introduce a unified theory of imperative sequential programming. In this programming paradigm, a program P is defined by a set of *program variables*, called the *alphabet* of P, denoted by αP, and a program command c written in the following syntax, given as a BNF grammar,

$$c ::= x := e \mid c; c \mid c \lhd b \rhd c \mid c \sqcap c \mid b * c \tag{2.1}$$

where e is an expression and b a boolean expression; $c_1 \lhd b \rhd c_2$ is the conditional choice equivalent to "if b then c_1 else c_2" in other programming languages; $c \sqcap c$ is the *non-deterministic choice* that is used as an abstraction mechanism; $b * c$ is iteration equivalent to "while b do c".

A sequential program P is regarded as a closed program such that for given initial values of its variables (that form an *initial state*), the execution of its command c will change them into some possible final values, called the *final state* of the program, if the execution terminates. The semantics of programs in the above simple syntax is defined based on UTP [HH98] as relations between the initial and final states.

States It is assumed that an infinite set of names $\mathcal{X}$ representing *state variables* with an associated value space $\mathcal{V}$. A *state* of $\mathcal{X}$ is a function $s : \mathcal{X} \to \mathcal{V}$ and Σ is used to denote the set of all states of $\mathcal{X}$.

This allows us to study all the programs written in our language. For a subset X of $\mathcal{X}$, we call Σ_X the restrictions of Σ on X *the states of* X; an element of this set is called *state over* X. Note that state variables include both variables used in programs and auxiliary variables needed for defining semantics and specifying properties of programs. In particular, for a program, we call $\Sigma_{\alpha P}$ the *states of program* P.

For two sets X and Y of variables, a state s_1 over X and a state s_2 over Y, we define $s_1 \oplus s_2$ as the state s for which $s(x) = s_1(x)$ for $x \in X$ but $x \notin Y$ and $s(y) = s_2(y)$ for $y \in Y$. Thus, s_2 overwrites s_1 in $s_1 \oplus s_2$.

State Properties and State Relations A state property is a subset of the states Σ and can be specified by a predicate over $\mathcal{X}$, called a *state predicate.* For example, $x > y + 1$ defines the set of states s for that $s(x) > s(y) + 1$ holds. We say that a state s satisfies a predicate F, denoted by $s \models F$, if it is in the set defined by F.

A *state relation* R is a relation over the states Σ, i.e., a subset of the Cartesian product $\Sigma \times \Sigma$, and can be specified by a predicate over the state variables $\mathcal{X}$ and their primed version $\mathcal{X}' = \{x' \mid x \in \mathcal{X}\}$, where $\mathcal{X}'$ and $\mathcal{X}$ are disjoint sets of names. We say that a pair of states (s, s') satisfies a relation predicate $R(x_1, \ldots, x_k, y'_1, \ldots, y'_n)$ if

$$R(s(x_1)/x_1, \ldots, s(x_k)/x_k, s'(y_1)/y'_1, \ldots, s'(y_n)/y'_n)$$

holds, denoted by $(s, s') \models R$. Therefore, a relational predicate specifies a set of possible state changes. For example, $x' = x + 1$ specifies the possible state changes from any initial state to a final state in which the value of x is the value of x in the initial state plus 1. However, $x' \geq x + 1$ defines the possible changes from an initial state to a state in which x has a value not less than the initial value plus 1. A state predicate and a relational predicate only constrain the values of variables that occur in the predicates, leaving the other variables to take values freely. Thus, a state predicate F can also be interpreted as a relational predicate such that F holds for (s, s') if s satisfies F. In addition to the conventional propositional connectors $\vee$, $\wedge$ and $\neg$, we also define the sequential composition of relational predicates as the composition of relations

$$R_1; R_2 \mathrel{\widehat=} \exists x_0 \bullet R_1(x_0/x') \wedge R_2(x_0/x), \tag{2.2}$$

where x_0 is a vector of state variables; x and x' represent the vectors of all state variables and their primed versions in R_1 and R_2; and the substitutions are element-wise substitutions. Therefore, a pair of states (s, s') satisfies $R_1; R_2$ iff there exists a state s_0 such that (s, s_0) satisfies R_1 and (s_0, s') satisfies R_2.

Designs A semantic model of programs is defined based on the way we observe the execution of programs. For a sequential program, we observe which possible final states a program execution reaches from an initial state, i.e., the relation between the starting states and the final states of the program execution.

Definition 2.1.1 (Design)**.** *Given a finite set α of program variables (as the alphabet of a program in our interest), a state predicate p and a relational predicate R over α, we use the pair $(\alpha, p \vdash R)$ to represent a program* **design***. The relational predicate $p \vdash R$ is defined by $p \Rightarrow R$ that specifies a program that starts from an initial state s satisfying p and if its execution terminates, it terminates in a state s' such that $(s, s') \models R$.*

Such a design does not observe the *termination* of program executions and it is a model for reasoning about *partial correctness*. When the alphabet is known, we simply denote the design by $p \vdash R$. We call p the *precondition* and R the *postcondition* of the design.

To define the semantics of programs written in Syntax (2.1), we define the operations on designs over the same alphabet. In the following inductive definition, we use a simplified notation to assign design operations to program constructs. Note that on the left side of the definition, we mean the program symbols while the right side uses relational operations over the corresponding designs of a program, i.e., we identify programs with a corresponding design.

$$\begin{aligned}
x := e &\mathrel{\widehat=} \mathit{true} \vdash x' = e \wedge \textstyle\bigwedge_{y \in \alpha, y \not\equiv x} y' = y, \\
c_1; c_2 &\mathrel{\widehat=} c_1; c_2 \\
c_1 \lhd b \rhd c_2 &\mathrel{\widehat=} b \wedge c_1 \vee \neg b \wedge c_2, \\
c_1 \sqcap c_2 &\mathrel{\widehat=} c_1 \vee c_2, \\
b * c &\mathrel{\widehat=} (c; b * c) \lhd b \rhd \mathbf{skip},
\end{aligned} \tag{2.3}$$

where we have **skip** $\mathrel{\widehat=}$ $\mathit{true} \vdash \bigwedge_{x \in \alpha}(x' = x)$. We also define **chaos** $\mathrel{\widehat=}$ $\mathit{false} \vdash \mathit{true}$. In the rest of the paper, we also use *farmed designs* of the form $X : p \vdash R$ to denote that only variables in X can be changed by the design $p \vdash R$. So $x := e = \{x\} : \mathit{true} \vdash x' = e$.

However, for the semantics definition to be sound, we need to show that the set $\mathcal{D}$ of designs is closed under the operations defined in (2.3), i.e., the predicates on the right-hand-side of the equations are equivalent to designs of the form $p \vdash R$. Notice that the iterative command is inductively defined. Closure requires the establishment of a partial order $\sqsubseteq$ that forms a *complete partial order* (CPO) for the set of designs $\mathcal{D}$.

Definition 2.1.2 (Refinement of designs). *A design $D_l = (\alpha, p_l \vdash R_l)$ is a* refinement *of a design $D_h = (\alpha, p_h \vdash R_h)$, if*

$$\forall x, x' \bullet (p_l \Rightarrow R_l) \Rightarrow (p_h \Rightarrow R_h)$$

is valid, where x and x' represent all the state variables and their primed versions in D_l and D_h.

We denote the refinement relation by $D_h \sqsubseteq D_l$. The refinement relation says that any property satisfied by the "higher level" design D_h is preserved (or satisfied) by the "lower level" design D_l. The refinement relation can be proved using the following theorem.

Theorem 2.1.1. *$D_h \sqsubseteq D_l$ when*

1. *the pre-condition of the lower level is weaker: $p_h \Rightarrow p_l$, and*
2. *the post-condition of the lower level is stronger: $p_l \wedge R_l \Rightarrow R_h$.*

The following theorem shows that $\sqsubseteq$ is indeed a "refinement relation between programs" and forms a CPO.

Theorem 2.1.2. *The set $\mathcal{D}$ of designs and the refinement relation $\sqsubseteq$ satisfy the following properties:*

1. *$\mathcal{D}$ is closed under the sequential composition "$;$", conditional choice "$\lhd b \rhd$" and non-deterministic choice "$\sqcap$" defined in (2.3),*
2. *$\sqsubseteq$ is a partial order on the domain of designs $\mathcal{D}$,*
3. *$\sqsubseteq$ is preserved by sequential composition, conditional choice and non-deterministic choice, i.e., if $D_h \sqsubseteq D_l$ then for any D*

$$\begin{array}{rr} D; D_h \sqsubseteq D; D_l, & D_h; D \sqsubseteq D_l; D, \\ D_h \lhd b \rhd D \sqsubseteq D_l \lhd b \rhd D, & D_h \sqcap D \sqsubseteq D_l \sqcap D, \end{array}$$

4. *$(\mathcal{D}, \sqsubseteq)$ forms a CPO and the recursive equation $X = (D; X) \lhd b \rhd$* **skip** *has a smallest fixed-point, denoted by $b * D$, which may be calculated from the bottom element* **chaos** *in $(\mathcal{D}, \sqsubseteq)$.*

For the proof of the theorems, we refer to the book on UTP [HH98]. D_1 and D_2 are *equivalent*, denoted as $D_1 = D_2$ if they refine each other, e.g., $D_1 \sqcap D_2 = D_2 \sqcap D_1$, $D_1 \lhd b \rhd D_2 = D_2 \lhd \neg b \rhd D_1$, and $D_1 \sqcap D_2 = D_1$ iff $D_1 \sqsubseteq D_2$. Therefore, the relation $\sqsubseteq$ is fundamental for the development of the refinement calculus to support correct by design in program development, as well as for defining the semantics of programs.

When refining a higher level design to a lower level design, more program variables are introduced, or types of program variables are changed, e.g., a set variable implemented by a list. We may also compare designs given by different programmers. Thus, we must relate programs with different alphabets.

Definition 2.1.3 (Data refinement)**.** *Let $D_h = (\alpha_h, p_h \vdash R_h)$ and $D_l = (\alpha_l, p_l \vdash R_l)$ be two designs. $D_h \sqsubseteq D_l$ if there is a design $(\alpha_h \cup \alpha_l, \rho(\alpha_l, \alpha'_h))$ such that $\rho; D_h \sqsubseteq D_l; \rho$. We call ρ a* data refinement mapping.

Designs of Total Correctness The designs defined above do not support reasoning about termination of program execution. To observe execution initiation and termination, we introduce a boolean state variable ok and its primed counterpart ok', and lift a design $p \vdash R$ to $\mathcal{L}(p \vdash R)$ defined below:

$$\mathcal{L}(p \vdash R) \mathrel{\widehat{=}} ok \wedge p \Rightarrow ok' \wedge R.$$

This predicate describes the execution of a program in the following way: if the execution starts successfully ($ok = true$) in a state s such that precondition p holds, the execution will terminate ($ok' = true$) in a state s' for which $R(s, s')$ holds. A design D is called a *complete correctness design* if $\mathcal{L}(D) = D$. Notice that $\mathcal{L}$ is a *healthy lifting function* from the domain $\mathcal{D}$ of partially correct designs to the domain of complete correct designs $\mathcal{L}(\mathcal{D})$ in that $\mathcal{L}(\mathcal{L}(D)) = \mathcal{L}(D)$. The refinement relation can be lifted to the domain $\mathcal{L}(\mathcal{D})$, and Theorem 2.1.1 and 2.1.2 both hold. For details of UTP, we refer to the book [HH98]. In the rest of the paper, we assume the complete correctness semantic model, and omit the lifting function $\mathcal{L}$ in the discussion.

Linking Theories We can unify the theories of Hoare-logic [Hoa69] and the predicate transformer semantics of Dijkstra [DS90]. The Hoare-triple $\{p\}D\{r\}$ of a program D, which can be represented as a design according to the semantics given above, is defined to be $p \wedge D \Rightarrow r'$, where p and r are state predicates and r' is obtained from r by replacing all the state variables in r with their primed versions.

Given a state predicate r, the *weakest precondition* of the postcondition r for a design D, $wp(p \vdash R, r)$, is defined to be $p \wedge \neg(R; \neg r)$. Notice that this is a state predicate.

This unification allows the use of the laws in both theories to reason about program designs within UTP. The unifying theory is extended to object-oriented specification and design [KLWZ09, ZLLQ09].

2.1.2 Reactive Systems and Reactive Designs

The programs that have been considered so far in this section are sequential and object-oriented programs. For these programs, our semantic definition is only concerned with the relation between the initial and final states and the termination of execution. In general, in a *concurrent* or *reactive* program, a number of components (usually called processes) are running in parallel, each following its own thread of control. However, these processes interact with each other and/or with the environment (in the case of a reactive program) to exchange data and to synchronize their behavior. The termination of the processes and the program as whole is usually not a required property, though the *enabling condition* and *termination* of execution of individual actions are essential for the progress of all processes, i.e., they do not show *livelock* or *deadlock* behavior.

There are mainly two different paradigms of programming interaction and synchronization, shared memory-based programming and message-passing programming. However, there can be programs using both synchronization mechanisms, in distributed systems in which processes on the same node interact through shared variables, and processes on different nodes interact through message passing. We define a general model of *labeled transition systems* for describing the behavior of reactive systems.

Reactive Designs In general a reactive program can be considered as a set of *atomic actions* programmed in a concurrent programming language. The execution of such an atomic action carries out interactions with the environment and changes of the state of the variables. We give a symbolic name for each atomic action, which will be used to label the state transitions when defining the execution of a reactive program.

The execution of an atomic action changes the current state of the program to another state, just in the way a piece of sequential code does, thus it can be specified as a design $p \vdash R$. However, the execution requires resources that might be occupied by another process or a synchronization condition. The execution is then suspended in a *waiting* state. For allowing the observation of the waiting state, we introduce the new boolean state variables *wait* and *wait'* and define the following lifting function on designs

$$\mathcal{H}(D) \mathrel{\widehat{=}} \mathit{wait}' \lhd \mathit{wait} \rhd D,$$

specifying that the execution cannot proceed in a waiting state. Note that *wait* is not a program variable, and thus cannot be directly changed by a program command. Instead, *wait* allows us to observe waiting states when talking about the semantics of reactive programs. We call a design D a *reactive design* if $\mathcal{H}(D) = D$. Notice that $\mathcal{H}(\mathcal{H}(D)) = \mathcal{H}(D)$. The proofs of the following theorems are referred to the book on UTP [HH98].

Theorem 2.1.3 (Reactive design)**.** *The domain of reactive designs has the following closure properties:*

$$\begin{aligned} \mathcal{H}(D_1 \vee D_2) &= \mathcal{H}(D_1) \vee \mathcal{H}(D_2), \\ \mathcal{H}(D_1; D_2) &= \mathcal{H}(D_1); \mathcal{H}(D_2), \\ \mathcal{H}(D_1 \lhd b \rhd D_2) &= \mathcal{H}(D_1) \lhd b \rhd \mathcal{H}(D_2). \end{aligned}$$

Given a reactive design D and a state predicate g, we call $g \mathbin{\&} D$ a *guarded design* and its meaning is defined by

$$g \mathbin{\&} D \mathrel{\widehat{=}} D \lhd g \rhd (\mathit{true} \vdash \mathit{wait}').$$

Theorem 2.1.4. *If D is a reactive design, so is $g \mathbin{\&} D$.*

For non-reactive designs $p \vdash R$, we use the notation $g \mathbin{\&} (p \vdash R)$ to denote the guarded design $g \mathbin{\&} \mathcal{H}(p \vdash R)$, where it can be proved $\mathcal{H}(p \vdash R) = (\mathit{wait} \vee p) \vdash (\mathit{wait}' \lhd \mathit{wait} \rhd R)$. This guarded design specifies that if the guard condition g holds, the execution of design proceeds from non-waiting state, otherwise the execution is suspended. It is easy to prove that a guarded design is a reactive design.

Theorem 2.1.5 (Guarded design)**.** *For guarded designs, we have*

$$\begin{aligned} g_1 \mathbin{\&} D_1 \lhd b \rhd g_2 \mathbin{\&} D_2 &= (g_1 \lhd b \rhd g_2) \mathbin{\&} (D_1 \lhd b \rhd D_2), \\ g_1 \mathbin{\&} D_1; g_2 \mathbin{\&} D_2 &= g_1 \mathbin{\&} (D_1; g_2 \mathbin{\&} D_2), \\ g \mathbin{\&} D_1 \vee g \mathbin{\&} D_2 &= g \mathbin{\&} (D_1 \vee D_2), \\ g \mathbin{\&} D_1; D_2 &= g \mathbin{\&} (D_1; D_2). \end{aligned}$$

A concurrent program P is a set of atomic actions, and each action is a *guarded command* in the following syntax:

$$c ::= x := e \mid c; c \mid c \lhd b \rhd c \mid c \,\Box\, c \mid g \,\&\, c \mid b * c \tag{2.4}$$

Note that $x := e$ is interpreted as command guarded by *true*. The semantics of the commands is defined inductively by

$$\begin{aligned} x := e &\mathrel{\widehat{=}} \mathcal{H}(\mathit{true} \vdash x' = e \textstyle\bigwedge_{y \in \alpha, y \not\equiv x} y' = y) \\ g \,\&\, c &\mathrel{\widehat{=}} c \lhd g \rhd (\mathit{true} \vdash \mathit{wait}') \\ g_1 \,\&\, c_1 \,\Box \cdots \Box\, g_n \,\&\, c_n &\mathrel{\widehat{=}} (g_1 \vee \cdots \vee g_n) \,\&\, (g_1 \wedge c_1 \vee \cdots \vee g_n \wedge c_n) \end{aligned}$$

and for all other cases as defined in equation (2.3) for the sequential case. The semantics and reasoning of concurrent programs written in such a powerful language are quite complicated. The semantics of an atomic action does not generally equal to a guarded design of the form $g \,\&\, p \vdash R$. This imposes difficulty to separate the design of the synchronization conditions, i.e., the guards, from the design of the data functionality. Therefore, most concurrent programming languages only allow guarded commands of the form $g \,\&\, c$ such that no guards are in c anymore. A set of such atomic actions can also be represented as a Back's *action system* [BvW94], a UNITY program [CM88] and a TLA specification [Lam94].

Labeled State Transition Systems Labeled transition systems are often used to describe the behavior of reactive systems, and we will use them in the following sections when defining the semantics of components. Hence, the remaining part of this section deals with basic definitions and theorems about labeled transition systems. Intuitively, states are defined by the values of a set of variables including both data variables and variables for the flow of control, which we do not distinguish here. Labels represent events of execution of actions that can be *internal* events or events *observable* by the environments, i.e., interaction events.

Definition 2.1.4 (Labeled transition system)**.** *A* **labeled transition system** *is a tuple*

$$S = \langle \mathit{var}, \mathit{init}, \Omega, \Lambda \rangle,$$

where

– *var is the set of typed variables (not including ok and wait), denoted S.var, we define Σ_{var} to be the set of states over $\mathit{var} \cup \{\mathit{ok}, \mathit{wait}\}$,*

– *init is the initial condition defining the allowable initial states, denoted by S.init, and*

– *Ω and Λ are two disjoint sets of named atomic actions, called observable and internal actions, respectively; actions are of the form $a\{c\}$ consisting of a name a and a guarded command c as defined in Syntax (2.4). Observable actions are also called* interface actions.

In an action $a\{c\}$, we call c the *body* of a. For $\Gamma = \Omega \cup \Lambda$ and for two states s and s' in Σ_{var},

– an action $a \in \Gamma$ is said to be **enabled** at s if for the body c of a the implication $c[s(x)/x] \Rightarrow \neg wait'$ holds, and **disabled** otherwise.

– a state s is a **divergence state** if ok is $false$ and a **deadlock state** if $wait = true$.

– we define $\rightarrow\ \subseteq \Sigma_{var} \times \{a | a\{c\} \in \Gamma\} \times \Sigma_{var}$ as the state transition relation such that $s \xrightarrow{a} s'$ is a transition of S, if a is enabled at s and s' is a post-state of the body c of action a.

Notice that this is a general definition of labeled transition systems that includes both finite and infinite transition systems, closed concurrent systems in which processes share variables (when all actions are internal), and I/O automata. Further, it models both data rich models in which a state contains values of data variables, and symbolic state machines in which a state is a symbol represents an abstract state of a class of programs. In later sections, we will see the symbols for labeling the actions can also be interpreted as a combination of input events triggering a set of possible sequences of output events.

Definition 2.1.5 (Execution, observable execution and stable state)**.** *Given a labeled transition system S,*

1. *an* **execution** *of S is a sequence of transitions $s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} \cdots \xrightarrow{a_n} s_n$ of S, where $n \geq 0$ and s_i $(0 \leq i \leq n)$ are states over $var \cup \{ok, wait\}$ such that s_0 is an initial state of S.*

2. *a state s is said to be* **unstable** *if there exists an internal action enabled in s. A state that is not unstable is called a* **stable state***.*

3. *an* **observable execution** *of S is a sequence of external transitions*

$$s_0 \xRightarrow{a_1} s_1 \xRightarrow{a_2} \cdots \xRightarrow{a_n} s_n$$

where all $a_i \in \Omega$ for $i = 1, \ldots, n$, and $s \xRightarrow{a} s'$ if s and s' there exist internal actions $\tau_1, \ldots, \tau_{k+\ell}$ as well as states t_j for $k, \ell \geq 0$ such that

$$s \xrightarrow{\tau_1} \cdots \xrightarrow{\tau_k} t_k \xrightarrow{a} \cdots \xrightarrow{\tau_{k+\ell}} s'.$$

Notice that the executions (and observable executions) defined above include chaotic executions in which divergence states may occur. Therefore, we give the semantic definitions for transitions systems below following the ideas of *failure-divergence semantics* of CSP.

Definition 2.1.6 (Execution semantics)**.** *Let* $S = \langle var, init, \Omega, \Lambda\rangle$ *be a transition system. The* execution semantics *of* S *is defined by a pair* $(\mathcal{ED}(S), \mathcal{EF}(S))$ *of* execution divergences *and* execution failures*, where*

1. *A divergence execution in* $\mathcal{ED}(S)$ *is a finite observable execution sequence of* S
$$s_0 \overset{a_1}{\Longrightarrow} s_1 \overset{a_2}{\Longrightarrow} \cdots \overset{a_n}{\Longrightarrow} s_n$$
where there exists an divergence state s_k*,* $k \leq n$*. Notice that if* s_k *is a divergence state, each* s_j *with* $k \leq j \leq n$ *is also a divergence state.*

2. *The set* $\mathcal{EF}(S)$ *contains all the pairs* (σ, X) *where* σ *is a finite observable execution sequence of* S *and* $X \subseteq \Omega$ *such that one of the following conditions hold*

   (a) σ *is empty, denoted by* ε*, and there exists an allowable initial state* s_0 *such that* a *is disabled at* s_0 *for any* $a \in X$ *or* s_0 *is unstable and* X *can be any set,*

   (b) $\sigma \in \mathcal{ED}(S)$ *and* X *can be any subset of* Ω*, i.e., any interaction with the environment can be refused,*

   (c) $\sigma = s_0 \overset{a_1}{\Longrightarrow} \cdots \overset{a_k}{\Longrightarrow} s_k$ *and for any* s *in the sequence,* $s(ok) = true$ *and* $s_k(wait) = false$*, and each* $a \in X$ *is disabled at* s_k*, or* s_k *is unstable and* X *can be any set.*

The semantics takes both traces and data into account. The component X of $(\sigma, X) \in \mathcal{EF}(S)$ is called a set of *refusals* after the execution sequence tr. We call the subset $ExTrace(S) = \{\sigma \mid (\sigma, \emptyset) \in \mathcal{EF}(S)\}$ the *normal execution traces*, or simply *execution traces*.

When interaction behavior and properties are the main interest, we can omit the states from the sequences and define the *interaction divergences* $\mathcal{ID}(S)$ and *interaction failures* $\mathcal{IF}(S)$ as

$$\mathcal{ID}(S) = \{a_1 \ldots a_n \mid s_0 \overset{a_1}{\Longrightarrow} s_1 \overset{a_2}{\Longrightarrow} \cdots \overset{a_n}{\Longrightarrow} s_n \in \mathcal{ED}(S)\}$$
$$\mathcal{IF}(S) = \{(a_1 \ldots a_n, X) \mid (s_0 \overset{a_1}{\Longrightarrow} s_1 \overset{a_2}{\Longrightarrow} \cdots \overset{a_n}{\Longrightarrow} s_n, X) \in \mathcal{EF}(S)\}$$

We call the set $\mathcal{T}(S) = \{\sigma \mid (tr, \emptyset) \in \mathcal{IF}(S)\}$ the *normal interaction traces*, or simply *traces*. Also, when interaction is the sole interest, abstraction would be applied to the states so as to generate transition systems with *symbolic states* for the flow of control. Most existing modeling theories and verification techniques work effectively on transition systems with finite number of states, i.e., finite state systems.

Definition 2.1.7 (Refinement of reactive programs)**.** *Let*

$$S_l = \langle var, init_l, \Omega_l, \Lambda_l\rangle \quad and \quad S_h = \langle var, init_h, \Omega_h, \Lambda_h\rangle$$

be transition systems. S_l is a **refinement** *of S_h, denoted by $S_h \sqsubseteq S_l$, if $\mathcal{ED}(S_l) \subseteq \mathcal{ED}(S_h)$ and $\mathcal{EF}(S_l) \subseteq \mathcal{EF}(S_h)$, meaning that S_l is not more likely to diverge or deadlock when interacting with the environment through the interface actions Ω.*

Notice that we, for simplicity, assume that S_l and S_h have the same set of variables. When they have different variables, the refinement relation can be defined through a state mapping (called refinement mapping in TLA [Lam94]).

A labeled transition system is a general computational model for reactive programs developed in different technologies. Thus, the definition of refinement will lead to a refinement calculus when a modeling notation of reactive programs is defined that includes models of primitive components and their compositions. We will discuss this later when the rCOS notation is introduced, but the discussion will not be in great depth as we focus on defining the meaning of component-based software architecture. On the other hand, the following theorem provides a verification technique for checking refinement of transition systems that is similar to the relation of simulation of transition systems, but extended with data states.

Definition 2.1.8 (Refinement by simulation)**.** *For two transition systems S_h and S_l such that they have the same set of variables,*

- *let $guard_h(a)$ and $guard_l(a)$ be the enabling conditions, i.e., the guards g for an action a with body $g \,\&\, c$ in S_h and S_l, respectively,*
- *$next_h(a)$ and $next_l(a)$ are the designs, i.e., predicates in the form of $p \vdash R$, specifying the state transition relations defined by the body of an action $a\{g \,\&\, (p \vdash R)\}$ in S_h and S_l, respectively,*
- *$g(\Omega_h)$, $g(\Lambda_h)$, $g(\Omega_l)$ and $g(\Lambda_l)$ are the disjunctions of the guards of the interface actions and invisible actions of the programs S_h and S_l, respectively,*
- *$inext(S_h) = \bigvee_{a\in\Lambda_h} guard_h(a) \wedge next_h(a)$ the state transitions defined by the invisible actions of S_h, and*
- *$inext(S_l)$ analogously defined as $inext(S_h)$ above.*

We have $S_h \sqsubseteq S_l$ if the following conditions holds

1. *$S_l.init \Rightarrow S_h.init$, i.e., the initial condition of S_h is preserved by S_l,*
2. *for each $a \in \Omega_l$, $a \in \Omega_h$ and $guard_l(a) \Leftrightarrow guard_h(a)$,*
3. *for each $a \in \Omega_l$, $a \in \Omega_h$ and $next_h(a) \sqsubseteq next_l(a)$, and*
4. *$\neg g(\Omega_h) \Rightarrow (g(\Lambda_h) \Leftrightarrow g(\Lambda_l)) \wedge (inext(S_l) \Rightarrow inext(S_h))$, i.e., any possible internal action of S_l in an unstable state would be a transition allowable by an internal action of S_h.*

When $inext(S_h) \sqsubseteq inext(S_l)$, the fourth condition can be weakened to

$$\neg g(\Omega_h) \Rightarrow (g(\Lambda_h) \Leftrightarrow g(\Lambda_l))$$

In summary, the first condition ensures the allowable initial states of S_l are allowable for S_h; the second ensures S_l is not more likely to deadlock; the third guarantees that S_l is not more non-deterministic, thus not more likely to diverge, than S_h, and the fourth condition ensures any refining of the internal action in S_l should not introduce more deadlock because of removing internal transitions from unstable states. Notice that we cannot weaken the guards of the actions in a refinement as otherwise some safety properties can be violated.

This semantics extends and unifies the theories of refinement of closed concurrent programs with shared variables in [CM88, Lam94, BvW94, LJ99] and failure-divergence refinement of CSP [Ros98]. However, the properties of this unified semantics still have to be formally worked out in more detail.

Design and verification of reactive programs are challenging and the scalability of the techniques and tools is fundamental. The key to scalability is compositionality and reuse of design, proofs and verification algorithms. Decomposition of a concurrent program leads to the notion of reactive programs, that we model as components in rCOS. The rCOS component model is presented in the following sections.

2.2 Closed Components

The aim of this section is to develop a unified model of architecture of components, that are *passive service components* (simply called *components*) and *active coordinating components* (simply referred to as *processes*). This is the first decision that we make for *separation of concerns*. The reason is that components and processes are different in nature, and they play different roles in composing and coordinating services to form larger components. Components maintain and manage data to provide services, whereas processes coordinate and orchestrate services in business processes and workflows. Thus, they exhibit simpler semantic behaviors than "hybrid" components that can have both passive and active behaviors when interacting with the environment. However, as a semantic theory, we develop a unified semantic model for all kinds of architectural components - the passive, active and the general hybrid components. We do this step by step, starting with the passive components, then the active process and finally we will define compositions of components that produces general components with both passive and active behavior. We start in this section with the simplest kind of components - primitive closed components. They are passive.

A closed and passive component on one hand interacts with the environment (users/actors) to provide services and on the other hand carries out data processing and computation in response to those services. Thus, the model of a component consists of the *types* of the data, i.e., the program variables, of the component, the *functionality* of the operations on the data when providing services, and the *protocol* of the interactions in which the component interacts with the environment. The design of a component evolves from the techniques applied during the design process, i.e., decomposing, analyzing, and integrating different *viewpoints* to form a correctly functioning whole component, providing the services required by the environment. The model of a component is separated into a number of related models of different viewpoints, including static structure, static data functionality, interaction protocol, and dynamic control behavior. This separation of design concerns of these viewpoints is crucial to a) control the complexity of the models, and b) allow the appropriate use of different techniques and tools for modeling, analysis, design, and verification.

It is important to note that the types of program data are not regarded as a difficult design issue anymore. However, when *object-oriented programming* is used in the design and implementation of a component-based software system, the types, i.e., the classes of objects become complicated and their design is much more tightly coupled with the design of the functionality of a component. The rCOS method presents a combination of OO technology and component-based technology in which local data functionality is modeled with the unified theory of sequential programming, as discussed in the previous section.

2.2.1 Specification Notation for Primitive Closed Components

To develop tool support for a formal method, there is a need for a specification notation. In rCOS, the specification notation is actually a graphical input notation implemented in a tool, called *the rCOS modeler*.[1] However, in this chapter the specification notation is introduced incrementally so as to show how architectural components, their operations and semantics can be defined and used in examples. We first start with the simplest building blocks[2] in component software, which we call *primitive closed components*. Closed components *provide services* to the environment but they do not *require services* from other components to deliver the services. They are passive as they wait for the environment to call their provided services, having no autonomous actions to interact with the environment. Furthermore, being primitive components, they do not have internal autonomous actions that result from interaction among sub-components. We use the notation illustrated in Fig. 2.1 to specify primitive closed components, which is explained as follows.

Interfaces of Components The *provided interface* declares a list of methods or services that can be invoked or requested by clients. The interface also allows declarations of state variables. A closed component only provides services, and thus, it has only a provided interface and optionally an *internal interface*, which declares private methods. Private methods can only be called by provided or private methods of the same component.

Access Control and Data Functionality The control to the access and the data functionality of a method m, in a provided or internal interface, is defined by a combination of a guard g and a command c in the form of a guarded command $g \,\&\, c$.

The components that we will discuss in the rest of this section are all primitive closed components. This definition emphasizes on the *interface* of the provided services. The interface supports input and output identifications, data variables, and the functional description defined by the bodies of the interface methods. On the other hand, the guards of the methods are used to ensure that services are provided in the right order.

[1] `http://rcos.iist.unu.edu`

[2] In the sense of concepts and properties rather than size of software, e.g., measured by number of lines of code.

```
component K {
   T x = c; // initial state of component
   provided interface I { // provided methods
      m1(parameters) { g1 & c1 /* functionality definition */ };
      m2(parameters) { g2 & c2 /* functionality definition */ };
      ...
      m(parameters) { g & c /* functionality definition */ };
   };
   internal interface { // locally defined methods
      n1(parameters) { h1 & d1 /* functionality definition */ };
      n2(parameters) { h2 & d2 /* functionality definition */ };
      ...
      n(parameters) { h & d /* functionality definition */ };
   }
   class C1{...}; class C2{...}; ... // used in the above specification
}
```

Figure 2.1: Format of rCOS closed components

Based on the theory of guarded designs presented previously, we assume that in a closed component the access control and data functionality of each provided interface method m is defined by a guarded design $g\ \&\ D$. For a component K, we use $K.pIF$ to denote the provided interface of K, $K.iIF$ the internal interface of K, $K.var$ the variables of K, $K.init$ the set of initial states of K. Furthermore, we use $guard(m)$ and $body(m)$ to denote the guard g and the body D of m, respectively. For the sake of simplicity but without loosing theoretical generality, we only consider methods with at most one input parameter and at most one return parameter.

We define the behavior of component K by the transition relation of $\underline{K}$ defined in the next subsection.

2.2.2 Labeled Transition Systems of Primitive Closed Components

We now show that each primitive closed component specified using the rCOS notation can be defined by a labeled transition system. To this end, for each method definition $m(T_1\,x; T_2\,y)\{c\}$, we define the following set of events

$$\omega(m) = \{m(u)\{c[u/x, v/y]\} \mid u \in T_1\}.$$

We further define $\Omega(K) = \bigcup_{m\in K.pIF} \omega(m)$. Here, there is a quite subtle reason why the return parameter is not included in the events. It is because that

– returning a value is an "output" event to the environment and the choice of a return value is decided by the component itself, instead of the environment,

– we assume that the guards of provided methods do not depend on their return values,

– we assume a run to complete semantics, thus the termination of a method invocation does not depend on the output values of the methods, and

– most significantly, it is the data functionality design, i.e. design $p \vdash R$, of a method, that determines the range of non-deterministic choices of the return values of an invocation for a given input parameter, thus refining the design will reduce the range of non-determinism.

Definition 2.2.1 (Transition system of primitive closed component)**.** *For a primitive closed component K, we define the transition system*

$$\underline{K} = \langle K.var, K.init, \Omega(K), \emptyset\rangle$$

A transition $s \xrightarrow{m(u)} s'$ *of* $\underline{K}$ *is an execution of the invocation* $m(u)$ *if the following conditions hold,*

1. *the state space of* $\underline{K}$ *is the states over* $K.var$, $\Sigma_{K.var}$,
2. *the initial states of* $\underline{K}$ *are the same initial states of* K,
3. s *and* s' *are states of* K,
4. $m(u) \in \Omega(K)$ *and it is an invocation of a provided method* m *with in input value* u,
5. $s \oplus u$ *satisfies* $guard(m)$, *i.e.,* m *is enabled in state* s *for the input* u *(note that we identify the value* u *with the corresponding state assigning values to inputs* $u = u(in)$*), and there exists a state* v *of the output parameter* y *of* m
6. $(s \oplus u, s' \oplus v) \in body(m)$.

We omit the empty set of internal actions and denote the transition system of K by $= \langle K.var, K.init, \Omega(K)\rangle$. A step of state transition is defined by the design of the method body when the guard holds in the starting state s. For transition $t = s \xrightarrow{m(u)} s'$, we use *pre.t*, *post.t* and *event.t* to denote the pre-state s, the post-state s' and the event $m(u)$, respectively.

Definition 2.2.2 (Failure-divergence semantics of components)**.** *The* **execution failure divergence semantics** $\langle\mathcal{ED}(K), \mathcal{EF}(S)\rangle$ *(or the* **interaction failure divergence semantics** $\langle\mathcal{ID}(K), \mathcal{IF}(S)\rangle$*) of a component K is defined by the semantics of the corresponding labeled transition system, i.e., by the execution failure-divergence semantics $\langle\mathcal{ED}(\underline{K}), \mathcal{EF}(\underline{K})\rangle$ (or the interaction failure-divergence semantics $\langle\mathcal{ID}(\underline{K}), \mathcal{IF}(\underline{K})\rangle$).*

The traces *traces*(K) of K are also defined by the traces of the corresponding transition system: *traces*$(K) \mathrel{\widehat{=}}$ *traces*$(\underline{K})$.

Example 2.2.1. *To illustrate a reactive component using guarded commands, we give an example of a component model below describing the behavior of a memory that a processor can interact with to write and read the value of the memory. It provides two methods for writing a value to and reading the content out of the memory cell of type Z, requiring that the first operation has to be a write operation.*

```
component M {
   provided interface MIF {
      Z d;
      bool start = false;
      W(Z v) { true & (d := v; start := true) }
      R(; Z v) { start & (v := d) }
   }
}
```

Remarks page We would like to make the following important notes on the expressiveness of this model by relating it to traditional theories.

1. This model is very much similar to the model of Temporal Logic of Actions (TLA) for concurrent programs [Lam02]. However, "actions" in TLA are autonomous and models interact through shared variables. Here, a component is a passive entity and it interacts with the environment through method invocations. Another significant difference between rCOS and TLA is that rCOS combines state-based modeling of data changes and event-based description of interaction protocols or behavior.

2. In the same way as to TLA, the model of components in rCOS is related to Back's *action systems* [BvW94] that extends Dijkstra's *guarded commands language* [DS90] to concurrent programming.

3. Similar to the relation with I/O automata, the rCOS model of components combines data state changes with event-based interaction behavior. The latter

can be specified in CSP [Hoa85, Ros98]. Indeed, failure-divergence semantics and the traces of a component K are directly influenced by the concepts and definitions in CSP. However, an event $m(u)$ in rCOS is an abstraction of the *extended rendezvous* for the synchronizations of receiving an invocation to m and returning the completion of the execution of m. This assumes a *run to complete semantics for method invocations.* For the relation between I/O automata and process algebras, we refer to the paper by Vaandrager [Vaa91].

4. Other formalisms like, e.g. CSP-OZ [Fis00, HO02], also combine state and event-based interaction models in a similar way. These approaches and also similar combinations like Circus [WC02] share the idea of rCOS that different formal techniques are necessary to cope with the complexity of most non-trivial applications. Contrary to rCOS, they promote the combination of fixed existing formal languages, whereas the spirit of rCOS is to provide a general semantic framework and leaving the choice of the concrete applied formalisms to the engineers.

The above relations show that the rCOS model of components unifies the semantics models of data, data functionality of each step of interaction, and event-based interaction behavior. However, the purpose of the unification is not to "mix them together" for the expressive power. Instead, the unification is for their consistent integration and the separation of the treatments of the different concerns. Therefore, rCOS promotes the ideas of *Unifying Theories of Programming* [HH98, BG77] for *Separation of Concerns*, instead of extending a notation to increase expressive power.

2.2.3 Component Contracts

We continue with defining necessary constructs for component-based design, i.e., contracts.

Definition 2.2.3 (Contract). *A* **component contract** *C is just like a primitive component, but the body of each method $m \in C.pIF$ is a guarded design g_m & $(p_m \vdash R_m)$.*

So each closed component K is semantically equivalent to a contract. Contracts are thus an important notion for the requirement specification and verification of the correct design and implementation through refinements. They can be easily modeled by a state machine, which is the vehicle of model checking. The contract of component M of Example 2.2.1 on page 40 is given as follows.

```
component M {
   provided interface MIF {
      Z d; bool start = false;
      W(Z v) { true & ({d,start}:true ⊢ d' = v  ∧ start' = true) }
      R(; Z v) { start & ({v}: true  ⊢ v' = d) }
   }
}
```

Notice that in both the component M of Example 2.2.1 and its contract, the state variable *start* is a protocol control variable.

Clearly, for each contract C, the labeled actions in the corresponding transition system $\underline{C}$ are all of the form $m(T_1\, x; T_2\, y)\{g \;\&\; (p \vdash R)\}$. Notice that in general a method of the provided interface can be non-deterministic, especially at a high level abstraction. Some of the traces are non-deterministic in a way that a client can still get blocked, even if it interacts with K following such a trace from the provided interface. Therefore, $\mathit{traces}(K)$ cannot be used as a description of the provided protocol of the component, for third party composition, because a protocol is commonly assumed to ensure non-blocking behavior.

Definition 2.2.4 (Input-deterministic trace and protocol)**.** *We call a trace $tr = a_1 \cdots a_n$ of a component transition system $\underline{K}$* **input-deterministic** *or* **non-blockable** *if for any of its prefixes $\mathit{pref} = a_1 \cdots a_k$, there does not exist a set X of provided events of K such that $a_{k+1} \in X$ and $(\mathit{pref}, X) \in \mathcal{IF}(K)$. And for a closed component K, we call the set of its input deterministic traces the* **provided protocol** *of K, and we denote it by $\mathcal{PP}(K)$ (and also $\mathcal{PP}(\underline{K})$).*

For the rest of the chapter, We use the notion “component” also for a “contract”, as they are specifications of components at different levels of abstractions and for different purposes.

2.2.4 Refinement between Closed Components

Refinement between two components K_h and K_l, denoted by $K_h \sqsubseteq K_l$, compares the services that they provide to the clients. However, this relation is naturally defined by the refinement relation $\underline{K_h} \sqsubseteq \underline{K_l}$ of their labeled transitions systems. Also, as a specialized form of Theorem 2.1.8, we have the following theorem for the refinement relation between two primitive closed components.

Theorem 2.2.1. *If $K_h \sqsubseteq K_l$, $\mathcal{PP}(K_h) \subseteq \mathcal{PP}(K_l)$.*

Proof. The proof is given by induction on the length of traces. From an initial state s_0, e is non-blockable in K_h only if e is enabled in all the possible initial states of K_h. Hence, if e is non-blockable in K_h, so is it in K_l. Assume the theorem holds for all traces of length no longer than $k \geq 1$. If a trace $tr = e_1 \ldots e_k e_{k+1}$ is not blockable in K_h, all its prefixes are non-blockable in K_h, thus so are they in K_l. If tr is blockable in K_l, then there is an X such that $e_{k+1} \in X$ and $(e_1 \ldots e_k, X) \in \mathcal{IF}(K_l)$. Because $K_h \sqsubseteq K_l$, $\mathcal{IF}(K_l) \subseteq \mathcal{IF}(K_h)$, thus $(e_1 \ldots e_k, X) \in \mathcal{IF}(K_h)$. This is impossible because tr is not blockable in S_h. Hence, we have $tr \in \mathcal{PP}(K_l)$. □

Thus, a refined component provides more deterministic services to the environment, because protocols represent executions for which there is no internal non-determinism leading to deadlocks.

The result of Theorem 2.2.1 is noteworthy, because the subset relation is reversed compared to the usual subset relation defining refinement; for instance, we have $\mathcal{IF}(K_l) \subseteq \mathcal{IF}(K_h)$ and $\mathcal{T}(K_l) \subseteq \mathcal{T}(K_h)$, but $\mathcal{PP}(K_h) \subseteq \mathcal{PP}(K_l)$. However, a bit of thought reveals that this actually makes sense, because removal of failures leads to potentially more protocols. For traces this is a bit more surprising, but in failure-divergence semantics the traces are derived from failures, so they are not independent. This also leads to the fact that the correctness of the theorem actually depends on the divergences: the theorem cannot hold in the stable-failures model and the traces model, because both have a top element regarding the refinement order. For both of these top elements (the terminating process for the trace model and the divergent process for the stable-failures model) the set of protocols is empty.

The semantic definition of refinement of components (or contracts) by Definition 2.1.2 does not directly support to verify that one component M_l refines another M_h. To solve this problem, we have the following theorem.

Theorem 2.2.2. *Let C_l and C_h be two contracts such that $C_l.pIF = C_h.pIF$, (simply denoted as pIF). $M_h \sqsubseteq M_l$ if there is a total mapping from the states over $C_l.var$ to the states over $C_h.var$, $\rho : C_l.var \longmapsto C_h.var$, that can be written as a design with variables in $C_l.var$ and $C_h.var'$ such that the following conditions hold.*

1. *Mapping ρ preserves initial states, i.e., $\rho(C_l.init) \subseteq C_h.init$.*
2. *No guards of the methods of C_h are weakened — undermines safety, or strengthened — introduces likelihood of deadlock, i.e., $\rho \Rightarrow (guard_l(m) \Leftrightarrow guard_h(m)')$ for all $m \in pIF$, where $guard_h(m)'$ is the predicate obtained from $guard_h(m)$ with all its variables replaced by their primed versions,*
3. *The data functionality of each method in C_l refines the data functionality of*

the corresponding method in C_h, i.e., for all $m \in pIF$,

$$\rho; body_h(m) \sqsubseteq body_l(m); \rho.$$

The need for the mapping to be total is to ensure that any state in the refined component C_l implements a state in the "abstract contract" C_h. With the upward refinement mapping ρ from the states of C_l at the lower level of abstraction to the states of C_h at a higher level of abstraction, the refinement relation is also called an *upward simulation* and it is denoted by $C_l \preceq_{up} C_h$. Similarly, we have a theorem about *downward simulations*, which are denoted by $C_l \preceq_{down} C_h$.

Theorem 2.2.3. *Let C_l and C_h be two contracts. $C_h \sqsubseteq C_l$ if there is a total mapping from the states over $C_h.var$ to the states over $C_l.var$, $\rho : C_h.var \longmapsto C_l.var$, that can be written as a design with variables in $C_l.var'$ and $C_h.var$ such that the following conditions hold.*

1. *Mapping ρ preserves initial states, i.e., $C_l.init \subseteq \rho(C_h.init)$.*
2. *No guards of the methods of C_h are weakened — undermines safety, or strengthened — introduces likelihood of deadlock, i.e., $\rho \Rightarrow (guard_l(m)' \Leftrightarrow guard_h(m))$ for all $m \in pIF$, and*
3. *The data functionality of each method in C_l refines the data functionality of the corresponding method in C_h, i.e., for all $m \in pIF$,*

$$body_h(m); \rho \sqsubseteq \rho; body_l(m).$$

The following theorem shows the completeness of the simulation techniques for proving refinement between components.

Theorem 2.2.4. *$C_h \sqsubseteq C_l$ if and only if there exists a contract C such that*

$$C_l \preceq_{up} C \preceq_{down} C_h.$$

The proofs and details of the discussion about the importance of the above theorems can be found in [CZ06].

2.3 Open Components

The components defined in the previous section are self-contained and they implement the functionality of the services, which they provide to the clients. However, component-based software engineering is about to build new software through reuse of exiting components that are *adapted* and *connected* together. These adapters and connectors are typical *open components*. They provide methods to be called by clients on one hand, and on the other, they *require* methods of other components.

2.3.1 Specification of Open Components

Open components extend closed components with *required interfaces*. The body of a provided method may contain undefined methods that are to be provided when composed with other components. We therefore extend the rCOS specification notation for closed components with a declaration of a *required interface* as given in Fig. 2.2.

```
component K {
   T x = c; // state of component
   provided interface I { // provided methods
      m1(parameters) { g1 & c1 /* functionality definition */ };
      m2(parameters) { g2 & c2 /* functionality definition */ };
      ...
      m(parameters) { g & c /* functionality definition */ };
   };
   internal interface { // locally defined methods
      n1(parameters) { h1 & d1 /* functionality definition */ };
      n2(parameters) { h2 & d2 /* functionality definition */ };
      ...
      n(parameters) { h & d /* functionality definition */ };
   };
   required interface J { // required services
      T y = d;
      n1(parameters), n2(parameters), n3(parameters)
   };
   class C1{...}; class C2{...}; ... // used in the above specification
}
```

Figure 2.2: Format of rCOS primitive open components

Notice that the required interface declares method signatures that do not occur in either the provided or the internal interfaces. It declares method signatures without bodies, but for generality we allow a required interface to declare its state variables too.

Example 2.3.1. *If we "plug" the provided interface of the memory component M of Example 2.2.1 into the required interface of the following open component, we obtain an one-place buffer.*

```
component Buff {
   provided interface BuffIF {
      bool r = false, w = true;
      put(Z v) { w& (W(v); r := true; w := false) }
      get(; Z v) { r& (R(; v); r := false; w := true) }
   }
   required interface BuffrIF {
      W(Z v), R(; Z v)
   }
}
```

2.3.2 Semantics and Refinement of Open Components

With the specification of open components using guarded commands, the denotational semantics of an open component K is defined as a functional as follows.

Definition 2.3.1 (Semantics of commands with calls to undefined methods)**.** *Let K be a specification of an open component with provided interface $K.pIF$, state variables $K.var$, internal interface $K.iIF$ and required interface $K.rIF$, the semantics of K is the functional $[\![K]\!] : \mathcal{C}(K.rIF) \longmapsto \mathcal{C}(K.pIF)$ such that for each contract C in the set $\mathcal{C}(K.rIF)$ of all the possible contracts for the interface $K.rIF$, $[\![K]\!](C)$ is a contract in the set $\mathcal{C}(K.pIF)$ of all contracts for the interface $K.pIF$ defined by the specification of the closed component $K(C)$ in which*

1. *the provided interface $K(C).pIF = K.pIF$,*
2. *the state variables $K(C).var = K.var$, and*
3. *the internal interface $K(C).iIF = K.iIF \cup K.rIF$, where the bodies of the methods in $K.rIF$ are now defined to be their guarded designs given in C.*

Definition 2.3.2. *Let K_1 and K_2 be specifications of open components with the same provided and required interfaces, respectively. K_2 is a* **refinement** *of K_1, $K_1 \sqsubseteq K_2$, if $K_1(C) \sqsubseteq K_2(C)$ holds for any contract C of the required interface of K_1 and K_2.*

The following theorem is used to establish the refinement relation of instantiated open components.

Theorem 2.3.1. *Let K be a specification of open components. For two contracts C_1 and C_2 of the required interface $K.rIF$, if $C_1 \sqsubseteq C_2$ then $K(C_1) \sqsubseteq K(C_2)$.*

To establish a refinement relation between two concretely given open components $C_1 \sqsubseteq C_2$, a refinement calculus with algebraic laws of programs are useful, e.g.

$c; n(a, y) = n(a, y); c$ for any command if a and y do not occur in command c. However, the above denotational semantic semantics is in general difficult to use for checking of one component refines another or for verification of properties.

We define the notion of contracts for open components by extending the semantics of sequential programs and reactive programs to those programs in which commands contain invocations to undefined methods declared in the required interface of an open component.

Definition 2.3.3 (Design with history of method invocations). *We introduce an auxiliary state variable sqr, which has the type of sequences of method invocation symbols and define the design of a command that contains invocations to undefined methods as follows,*

- $x := e = \{x\} : true \vdash x' = e$, *implying an assignment does not change sqr,*
- *each method invocation to an undefined method* $n(T_1\, x; T_2\, y)$ *is replaced by a design*
$$\{sqr, y\} : true \vdash y' \in T_2 \wedge sqr' = sqr \cdot \{n(x)\},$$
where $\cdot$ *denotes concatenation of sequences, and*
- *the semantics for all sequential composition operations, i.e., sequencing, conditional choice, non-deterministic choice, and recursion, are not changed.*

A sequential design that has been enriched with the history variable sqr introduced above can then be lifted to a reactive design using the lifting function.

With the semantics of reactive commands, we can define the semantics of a provided method $m()\{c\}$ in an open component. Also, given a state s of the component, the execution of an invocation to $m()$ from s will result in a set of sequences of possible (because of non-determinism) invocations to the required methods, recorded as the value of sqr in the post-state, denoted by $sqr(m(), s)$.

Definition 2.3.4 (Contract of open component). *The* **contract** $\overline{K}$ *of an open component* K *is defined analogously to that of a closed component except that the semantics of the bodies of provided methods are enriched with sequence observables as defined in Definition 2.3.3.*

For further understanding of this definition, let us give the weakest assumption on behavior of the methods required by an open component. To this end, we define the *weakest terminating contract*, which is a contract without side-effects, thus leaving all input variables of a method unchanged, and setting its output to an arbitrary value. The weakest terminating contract $wtc(rIF)$ of the required interface rIF is defined such that each method $m(x; y) \in rIF$ is instantiated with

$$m(x; y)\{true \ \& \ (true \vdash x' = x)\}.$$

Thus, $wtc(rIF)$ accepts all invocations to its methods and the execution of a method invocation always terminates. However, the data functionally is unspecified.

Proposition 2.3.1. *We have the following conjectures, but their proofs have not been established yet.*

1. *Given two open components K_1 and K_2, $K_1 \sqsubseteq K_2$ if $\overline{K_1} \sqsubseteq \overline{K_2}$.*
2. *$\overline{K}$ is equivalent to $\overline{K(wtc(K.rIF))}$.*

2.3.3 Transition Systems

Given an open component K, let

– $pE(K) = \{m(u) \mid m(T_1\, x; T_2\, y) \in K.pIF \wedge u \in T_1\}$, and

– $rE(K) = \{n(u) \mid n(T_1\, x; T_2\, y) \in K.rIF \wedge u \in T_1\}$

be the possible incoming method invocations and outgoing invocations to the required methods, respectively. Further, let $\Omega(K) = pE(K) \times 2^{rE(K)^*}$. With this preparation, we can define the transition systems of open components:

Definition 2.3.5 (Transition system of open component)**.** *Let K be an open component, we define the labeled state transition system*

$$\underline{K} = \langle K.var, K.init, \Omega(K), \emptyset \rangle,$$

such that $s \xrightarrow{m(u)/E} s'$ is a transition from state s to state s' if

– *$(s, s') \models c[u/x, v/y']$, where c is the semantic body of the method $m()$ in $\overline{K}$, and*

– *E is the set of sequences of invocations to methods in $K.rIF$, recorded in sqr in the execution from state s that leads to state s'. Here the states of $\underline{K}$ do not record the value of sqr as it is recorded in the events of the transition.*

Notice E in $s \xrightarrow{m(u)/E} s'$ is only the set of possible traces of required method invocations from s to s', not from the initial state of the transition system $\underline{K}$. The definition takes non-determinism of the provided methods into account. It shows that each state transition is triggered by an invocation of a method in the provided interface. The execution of the method may require a set of possible sequences of invocations to the methods in the required interface. Therefore, we define the following notions for open component K.

– For each trace $tr = a_1/E_1 \ldots a_k/E_k$, we have a provided trace $tr^{>} = a_1 \ldots a_k$ and sets of required traces $tr^{<} = E_1 \cdots E_k$, where $\cdot$ is the concatenation operation on set of sequences.

– For each provided trace pt, $\mathcal{Q}(pt) = \bigcup\{tr^{<} \mid tr \in \mathcal{T}(\overline{K}), tr^{>} = pt\}$ is the set of all corresponding required traces of pt.

– A provided trace pt is a **non-blocking provided trace** if for any trace tr such that $tr^{>} = pt$, tr is a non-blocking trace of $\overline{K}$.

– The provided protocol of K, denoted by $\mathcal{PP}(K)$ is the set of all non-blocking provided traces.

– The required protocol of K is a union of the sets of required traces of non-blocking provided traces $\mathcal{RP}(K) = \bigcup_{pt \in \mathcal{PP}(K)} \mathcal{Q}(pt)$.

The model of an open component is a natural extension to that of a closed component, and a closed component is a special case when the required interface is empty. Consequently, the set of required traces of a closed component is empty.

2.4 Processes

All components that we have defined so far are passive in the sense that a component starts to execute only when a provided method is invoked from the environment (say, by a client). Once a provided method is invoked, the component starts to execute the body of the method, provided it is enabled. The execution of the method is atomic and follows the *run to complete semantics*. However, it is often the case that *active software entities* are used to coordinate the components when the components are being executed. For example, assume we have two copies of component *Buff* in Example 2.3.1, say B_1 and B_2 whose provided interfaces are the same as *Buff*, except for *put* and *get* being renamed to put_i and get_i for B_i, respectively, where $i = 1, 2$. We can then write a program P that repeatedly calls $get_1(; a); put_2(a)$

```
process P {
   T x = c; // initial state of process
   actions { // guarded commands
      a1 { g1 & c1 };
      ...
      ak { gk & ck }
   };
   required interface J { // required services
      T y = d;
      n1(parameters), n2(parameters), n3(parameters)
   };
   internal interface { // locally defined methods
      n1(parameters) { h1 & d1 /* functionality definition */ };
      n2(parameters) { h2 & d2 /* functionality definition */ };
      ...
      n(parameters) { h & d /* functionality definition */ };
   };
   class C1{...}; class C2{...}; ... // used in the above specification
}
```

Figure 2.3: Format of rCOS process specifications

when both get_1 and put_2 are enabled. Then, P *glues* B_1 and B_2 to form a two-place buffer. We call such an active software entity a *process*.

2.4.1 Specification of Processes

In this section, we define a class of processes that do not provide services to clients but only actively calls provided services of other components. In the rCOS specification notation, such a process is specified in the format shown in Fig. 2.3. In the body of an action (which does not contain parameters), there are calls to methods in both the internal interface section and the required interface section, but not to other methods.

2.4.2 Contracts of Processes

Notice that the actions, denoted by $P.ifa$, are autonomous in the sense that when being enabled they can be non-deterministically selected to execute. The execution of an action is atomic and may involve invocations to methods in the required interface $P.rIF$, as well as program statements and invocations to methods defined in the internal interface $P.iIF$. We will see later when we define the composition of

```
process P {
   T x = c; // initial state of process
   actions { // reactive designs
      a1 { /* g1 & c1 design enriched by history variables sqr */ };
      ...
      ak { /* gk & ck design enriched by history variables sqr */ }
   };
   required interface J { // required services
      T y = d;
      n1(parameters), n2(parameters), n3(parameters)
   };
   class C1{...}; class C2{...}; ... // used in the above specification
}
```

Figure 2.4: Format of rCOS process contracts

a component and a process that execution of an atomic action a in P *synchronizes* all the executions of required methods contained in a, i.e., the execution of a locks all these methods until a terminates. For instance, in the two place buffer example at the beginning of this section, $get_1(;a); put_2(a)$ is the only action of the process P. When this action is being executed, B_1 cannot execute another *get* until this action finishes.

The denotational semantics of a process P is similar to that of an open component in the sense that it is a functional over the set $\mathcal{C}(P.rIF)$ of the contracts of interface $P.rIF$ such that for each contract C in $\mathcal{C}(P.rIF)$, $[\![P]\!](C)$ is a fully defined process, called a *self-contained process*, containing the autonomous actions $P.ifa$. In this way, a failure-divergence semantics in terms of actions in $P.ifa$ and a refinement relation can be defined following the definitions of Sect. 2.1.

However, we apply the same trick as we did when defining the semantics in Definition 2.3.3 for the body of a provided method in an open component, which contains calls to undefined methods. Therefore, the execution of an atomic action a in a process from a state s records the set *sqr* of possible sequences of invocations to methods declared in the required interface.

Definition 2.4.1 (Contract of process)**.** *Given a specification of a process P in the form shown in Fig. 2.3, its* **contract** $\overline{P}$ *is defined analogously to Definition 2.3.4 by enrichment with history variables, i.e., it is specified as shown in Fig. 2.4.*

Example 2.4.1. *Consider two instances of the Buff component, B_1 and B_2, obtained from Buff by respectively renaming put to put_1 and put_2 as well as get to get_1 and get_2. We design a process that keeps getting an item from B_1 and putting it into B_2 when get_1 and put_2 are enabled. The contract of the process is specified as follows.*

```
process Shift {
   T x = c; // state of process
   actions { // reactive designs
      move { {sqr}: (get1(; x); put2(x) };
           // equals to true ⊢ sqr' = {get_1(;a) · put_2(a) | a ∈ Z}
   }
   required interface J { // required services
      get1(; Z x), put2(Z x)
   };
}
```

Notice that there is no guard for the process in the above example, it will be enabled whenever its environment are ready to synchronize on the required methods, i.e., they are enabled in their own flows of execution. Now we are ready to define the transition system for a process.

2.4.3 Transition Systems

Given a process P, we define the set $\omega P = 2^{P.rIF^*}$ to be the set of all sets of invocations sequences to methods in the required interface of P. Following the way in which we defined the transition system of an open component, we define the transition system of a process.

Definition 2.4.2 (Transition system of processes). *The transition system $\underline{P}$ of a process P is the quadruple $\langle P.var, P.init, \omega P, \emptyset\rangle$, where for $E \in \omega P$, states s, s' of P, and an action a of P with body c,*

$$s \xrightarrow{a/E} s' \text{ if } (s \oplus \{sqr \mapsto \emptyset\}, s' \oplus \{sqr \mapsto E\}) \vdash c$$

holds.

We can define the execution failure-divergence semantics $(\mathcal{ED}(P), \mathcal{EF}(P))$ and interaction failure-divergence semantics $(\mathcal{ID}(P), \mathcal{IF}(P))$ for process P in terms of the transition system $\underline{P}$. The interaction traces and the *failure-divergence refinement* of processes follow straightforward. However, a process can non-deterministically invoke methods of components, and its whole trace set is taken as the **required protocol**

$$\mathcal{RP}(P) = \bigcup\{E_1 \cdots E_k \mid /E_1 \cdots /E_k \in \mathcal{T}(P)\}.$$

2.5 Summary and Discussion

The chapter gives a brief introduction to rCOS, the formal methodology of component-based design and development. It defines and gives sound formal foundation of closed and open components, processes, and how components are composed and refined. More details about rCOS can be found in the list of publications by Zhiming Liu et al [CHL06, CZ06, CLM07, CLL$^+$07, CLS$^+$07, CLS08, CHH$^+$08, CMS09, CLR$^+$09, HLL05a, HLL05b, HLL06a, HLL06b, LH06, LQL$^+$05, DFKL13]. The semantics of the component architecture is based on unified labeled transition systems with a failure-divergence semantics and refinement for sequential, object-oriented, and reactive designs. Our semantics particularly integrates a data-based as well as an interaction-based view. This allowed us to introduce a general and unified model of components, which are the building blocks of a model-driven software architecture: primitive closed components, open components, as well as active and passive generalized components. Construction of models and model refinements are supported by the rCOS Modeler tool. The method has been tested on enterprise systems [CHH$^+$08, CLR$^+$09], remote medical systems [XLD10] and service oriented systems [LH06].

Chapter 3

Automata-based Model of Components

In this chapter, we motivate and present an automata-based model describing the interaction behaviors of components. This automata-based model of components can be used in rCOS for easy checking of correct composition. We focus on the communication protocols of components and are concerned with the provided-required relation between a component and its environment.

Services provided by a component can be used required in any order, for example, *readFile()* cannot be called before *openFile()* in a file system. And to provide a service, the component may need to require services from outside, for example, a *cardPay* service of the storeseller component may need services *verifyCard* and *transaction* from the bank component. The requirement for given provided service of a component can be captured a regular set of sequences of invocations to services that should be provided by the environment. Before giving the formal definition of the models, we first introduce the needed techniques of automata theory and regular set.

Th interaction behavior of components may contain services that cannot be called during run-time due to non-determinism. We consider the *input-determinism* as the interface property for non-blockable composition. We develop an algorithm for checking whether the automata-based models satisfy the interface property. And another algorithm is presented for generating an interface model for any abstract model of components, and the generated interface model should satisfy the interface property and all the behaviors of the interface should be accepted by the abstract model. Components interact by service invocation or sending/receiving messages. Interaction style can be *synchronous* or *asynchronous*. The synchronous interaction means that the component will wait until receiving the reply after sending the request, and then continue next instruction. In the asynchronous style, a component continue to execute before receiving the reply of a request that is just sent. According to the number of components involved, interaction can be classified as one-to-one, or one-to-many, called broadcast. Composition operations can be different based on different interaction styles.

Substitutability is another important issue in component-based development, especially during the maintenance stage of life cycle. A component can be replaced by a new component as long as it conforms to the interface model. Further, a component with a new and better interface model should surely fit this replacement. Such criteria for judging whether an interface model 'better than' another is defined as *refinement* relation between interface models. Intuitively, a better one should provide more services while requiring less and less likely cause deadlock. The refinement relation will be discussed and studied based on traces and stepwise simulation on the provided/required events.

3.1 Background

In this chapter, we first briefly introduce finite state machine and regular set

3.1.1 Finite State Machine

A finite sate machine (FSM) or finite state automaton, or simply a state machine is widely used in mathematics for computation and in computer science as a formal language. It contains a finite set of states and transitions from one state to another by performing certain actions. The sequences of actions that record transitions from the initial state to the accepting state. For details, we refer the readers to [HMU79].

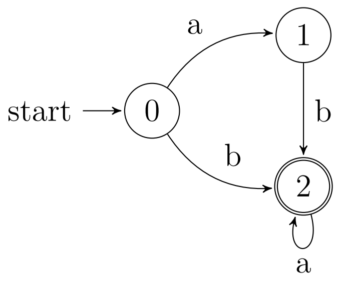
Definition 3.1.1 (Finite state machine). *An automaton $\mathcal{A}$ over Act is a tuple $\mathcal{A} = (Q, q_0, \mathcal{F}, Act, \mathcal{T})$*

- *a finite set $\mathcal{Q} = \{q_0, \ldots, q_n\}$ of states;*
- *a state $q_0 \in Q$ called the initial, or start, state;*
- *a subset $\mathcal{F}$ of $\mathcal{Q}$ called the accepting states;*
- *a finite set Act of actions, sometimes called an alphabet;*

– *a subset $\mathcal{T}$ of $\mathcal{Q} \times Act \times \mathcal{Q}$ called the transitions.*

A transition $(q, a, q') \in \mathcal{T}$ is usually written $q \xrightarrow{a} q'$. A FSM is called deterministic if for any transitions $q \xrightarrow{a} q_1$ and $q \xrightarrow{a} q_2$, it implies that $q_1 = q_2$.

An finite state machine is usually represented by a transition graph, whose nodes are states and whose arcs are the transitions. As an example consider the following finite state machine $\mathcal{A}$ over the alphabet $\{a, b\}$: The set of states is $\{0, 1, 2\}$ where



0 is the initial state and 2 is the accepting state. And it is obviously deterministic.

In the classical automaton theory, the behavior of an FSM $\mathcal{A}_0$ is usually taken to be the set of strings over *Act* which the it accepts and this set is called the language of $\mathcal{A}_0$.

Definition 3.1.2 (Language of a FSM.). *Let $\mathcal{A}$ be an automaton over Act, and $s = a_0, \ldots, a_n$ is a string over Act. Then $\mathcal{A}$ is said to accept s if there is a sequence of transitions in $\mathcal{A}$, from q_0 to an accepting state, whose arcs are labeled successively by $a_0, \ldots, a_n$.*

The Language of $\mathcal{A}$, denoted by $\mathcal{L}(\mathcal{A})$, is the set of strings accepted by $\mathcal{A}$.

3.1.2 Regular sets

We consider a set of strings over a given set *Act*, and would build by operations from other sets. Three important operations are:

$$\begin{aligned} \textit{Union}: &\quad S_1 \cup S_2 \\ \textit{Concatenation}: &\quad S_1 \cdot S_2 \quad \widehat{=} \{s_1 s_2 \mid s_1 \in S_1, s_2 \in S_2\} \\ \textit{Iteration}: &\quad S^* \quad \widehat{=} \{\epsilon\} \cup S \cup S \cdot S \cup \cdots \end{aligned}$$

The iteration S^* consists of all strings $s_1 s_2 \cdots S_n$, for $n \geq 0$, such that $s_i \in S$ for each i. The notion ϵ is for the empty string.

Definition 3.1.3 (Regular sets). *A set of strings over Act is said to be regular if it can be built from the empty set $\emptyset$ and the singleton sets $\{a\}$, for each $a \in Act$, using operations of union, concatenation, and iteration.*

The regular set can also be written in expression way, called *regular expression.* In the expression, a is for singleton set $\{a\}$, and ϵ is for the set $\{\epsilon\}$. Conventionally, $+$, $\cdot$, and * are used for set union, concatenation, and iteration. For example $(b+a\cdot b)\cdot a^*$ stands for the set of strings $\{b, ab, aba, ba, \cdots\}$. It is well-known in the classical theory that the following holds.

$$\begin{aligned} S\cdot\epsilon &= S \\ S\cdot\emptyset &= \emptyset \\ (S_1\cdot S_2)\cdot S_3 &= S_1\cdot(S_2\cdot S_3) \\ (S_1+S_2)\cdot T &= S_1\cdot T + S_2\cdot T \\ T\cdot(S_1+S_2) &= T\cdot S_1 + T\cdot S_2 \end{aligned}$$

Since automata theory and algebra of regular set are classic and well-founded, the details of proofs the following propositions can be seen in [HMU79].

Proposition 3.1.1. *The relation between finite state machines and regular sets are:*

- *For any finite state machine $\mathcal{A}$, the language $\mathcal{L}(\mathcal{A})$ is a regular set.*
- *Given a regular set S over Act, there is a deterministic finite state machine whose language is S. We use $\mathcal{M}(S)$ to denote one of machines with least states.*

3.2 Component Automata

A *closed component* provide services without the need to require from other components when it is deployed. For example, the package of Java APIs can be seen as a closed component and the APIs can be used by application developers. Here, we consider to develop an *Alarm* component, illustrated in Figure 3.1, which will require methods from Java API class *Timer*, *Blink*, and *Sound*. The Alarm component is *open* in the sense that it provide services while requiring services from other components. In order to fulfill a provided service, inside the provided service body, the component will invoke the required services.Since the basic control flow is sequence, branch or iteration, it is fair enough that the set of possible sequences of

```
/** The alarm component would provide basic methods, such as
 * set, cancel, and change the alarm time.
 * once the ring is fired, the alarm will call the Blink and Sound components to make an alert.
 * when alarm is stoped, it will call the Blink and Sound to turn off the alert
 */
import Timer, Blink, and Sound;
public class Alarm{
  void set(){
    Timer.start();
    Timer.addNotification();
  }
  void cancel(){
    Timer.stop();
    Timer.removeNotification();
  }
  void change(){
    Timer.stop();
    Timer.removeNotification();
    Timer.start();
    Timer.addNotification();
  }
  void ring(){
    Blink.lightOn();
    Sound.On();
  }
  void stop(){
    Blink.lightOff()
    Sound.Off();
  }
}
```

Figure 3.1: Alarm

required service invocation can be modeled by a regular set over the required events which model the service invocation. The procedure of invocation to an provided service of a component is abstracted as *provided event*. Besides provided and required services, there are also events that are triggered inside, for example, event *fire* will be triggered internally and autonomously when the specified interval of time elapses in component *Timer*. These kinds of events are called *internal events*. And the internal events will also cause a possible sequence of invocation to required services, which can also be captured by a regular set.

Therefore, we use a pair of provided/internal event and a regular set over required events to model a step of the whole procedure of executions of a given provided service of the component during run time.

We aim to model the interaction behaviors of component as the provided-required dependence relation. Thus, the sets of provided, internal, and required events should be disjoint.

Now, we present the automata-based model for the interaction behaviors of components, called *component automata*. The states are abstract symbolic states. The guards of transitions are encoded in the symbolic states such that in some states, the transitions are enabled or disabled, representing that the provided services are available or unavailable. A specific state which may be called error or illegal is introduced and denoted as f. The details of how the error state is produced will be discussed in the composition operation part. The set of provided, internal, required events are denoted as P,A, and R. The label for transitions consists a pair of P or A, and a regular set over R. The alphabet set can be denoted as $\Sigma(P, A, R) = (P \cup A) \times \mathbb{R}$, where $\mathbb{R}$ is the set of non-empty regular sets over R.

Definition 3.2.1 (Component Automaton). *A component automaton is a tuple* $C = (S, s_0, P, R, A, \delta)$ *where*

- S *is a finite set of states;*
- $s_0 \in S$ *is the initial state;*
- $f \in S$ *is the error state;*
- P, R, *and* A *are disjoint and finite sets of provided, required, and internal events, respectively;*
- $\delta \subseteq S \setminus \{f\} \times \Sigma(P, R, A) \times S$ *is the transition relation, where* $\Sigma(P, A, R) = (P \cup A) \times \mathbb{R}$, *where* $\mathbb{R}$ *is the set of non-empty regular sets over* R.

Notes. Unless stated otherwise, we only use f for the error state and state variable s with any sub or superscript denotes the non-error state in S. Without causing misunderstanding, throughout the thesis, we assume that the default tuple for component automaton C_i is $(S_i, s_0^i, f, P_i, R_i, A_i, \delta_i)$ for any subscript i and similarly for any superscript.

Conventions. The alphabet set $\Sigma(P, R, A)$ is denoted as $\Sigma(C)$, or even Σ for simplicity when causing no confusions. A transition $(s, \ell, s') \in \delta$ is usually written $s \xrightarrow{\ell} s'$, and $s \xrightarrow{w/T} s'$ if $\ell = (w, T)$; T is called *requirement* of w at state s and $T = \{\epsilon\}$ indicates that the component can directly provide w at state s without requiring from others. Transition $s \xrightarrow{w/\{\epsilon\}} s'$ can also be simply written $s \xrightarrow{w} s'$ or $s \xrightarrow{w/} s'$. The component automaton is called *closed*, if the requirement in all transitions is $\{\epsilon\}$, and *open*, otherwise.

A sequence of transitions $s_1 \xrightarrow{\ell_1} s_2 \xrightarrow{\ell_2} \cdots \xrightarrow{\ell_n} s_{n+1}$ is denoted $s_1 \xRightarrow{\ell_1,\ldots,\ell_n} s_{n+1}$.

If $w \in P$, then $s \xrightarrow{w/T} s'$ is called an *provided transitions step*, and *internal transition step*, otherwise. $s \xrightarrow{w/T} f$ is called *failure transition*, and can be written $s \xrightarrow{w/\bullet} f$ for simplicity, because when w is triggered at state s whether externally or internally, the component will be stuck in an error state no matter what the requirement is.

The internal events are prefixed with $';'$ to differentiate them from the provided events if needed. We use τ to represent any internal event, when there is no need to differentiate the internal events. We write $s \xrightarrow{w/\bullet} s'$ for $s \xrightarrow{w/T} s'$, when T is not essential.

At state s, the set of available provided/internal events $out(s)$ is defined as

$$\{w \in P \cup A \mid \exists s', w, T \bullet s \xrightarrow{w/T} s'\}.$$

Then, $out^P(s) = out(s) \cap P$ and $out^A(s) = out(s) \cap A$ are used to denote the set of available provided and internal events at state s. Here, we do not assume $out^P(s) = P$, which is mandatory in the *input-enabled* model.

Example 3.2.1. *The Alarm component provides services, such as* set,change*, and* cancel *the alarm time,* ring *and* stop *the alert. The component also requires services from Timer, such as* start, addNotificatioin*, and* removeNotification*, from Blink component the* turnOn *and* turnOff*, and from Sound component, the* turnOn *and* turnOff *the alert sound. To model the alarm component, the provided events may be noted as* $\{set, cancel, change, ring, stop\}$ *and the set of required events is* $\{Tstart, Tstop, Tadd, Tremove, BlinkOn, BlinkOff, SoundOn, SoundOff\}$*. From the* *program of Alarm, the elements of alphabet can be* $(set, Tstart\cdot Tadd)$, $(change, Tstop\cdot Tremove\cdot Tstart\cdot Tadd)$, $(cancel, Tstop\cdot Tremove)$, $(ring, BlinkOn\cdot SoundOn)$, *and* $(stop, BlinkOff\cdot SoundOff)$. *The component automaton of Alarm is graphically shown in Figure 3.2. The states are 0, 1, and 2 where 0 is the initial state. State f is omitted, since it is not reachable.*

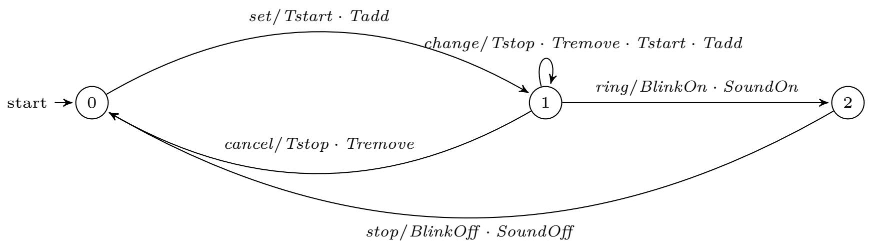


Figure 3.2: Component automaton of Alarm C_{Alarm}

Unless otherwise stated, the following definitions are given for component automaton $C = (S, s_0, f, P, R, A, \delta)$.

Definition 3.2.2 (Execution). *An alternating sequence of states and labels of the form* $e = \langle s_1, \ell_1, \ldots, s_k, \ell_k, s_{k+1}\rangle$, *with* $k > 0$ *and* $s_i \xrightarrow{\ell_i} s_{i+1}$ *for each* i *with* $0 < i \leq k$, *is called an* execution segment *(or simply* execution*) of the component automaton* C *at state* s_1. *It is called an* execution *of* C, *if* s_1 *is the initial state* s_0.

The set of all executions of component C *at state* s *is denoted* $\mathcal{E}(C, s)$, *and* $\mathcal{E}(C)$ *if* s *is the initial state.*

In component automaton C_{Alarm} of the Alarm component,

$$\langle 0, (set, Tstart\cdot Tadd), 1, (ring, BlinkOn\cdot SoundOn), 2, (stop, BlinkOff\cdot SoundOff)\rangle$$

is an execution. The $\langle 1, (change, Tstop\cdot Tremove\cdot Tstart\cdot Tadd), 1, \cdots\rangle$ is an execution segment at state 1.

Next, we define *trace* of components, a sequence of labels, representing the possible interaction behaviors of components.

Definition 3.2.3 (Trace, Provided trace, Required trace). *The traces, provided traces, and required traces are defined as follows.*

– *A sequence of elements of* Σ, $tr = \langle \ell_1, \ldots, \ell_k\rangle$ *is called a trace of component automaton* C *at state* s, *if there exists an execution segment* $e \in \mathcal{E}(C, s)$ *such* *that tr is obtained by removing states from e. That is, there exists s' such that $s \stackrel{tr}{\Longrightarrow} s'$. It is called a* trace *of the component, if s is the initial state.*

The set $traces(C, s)$ (or written $traces(s)$) of all traces of C at state s is denoted

$$traces(C, s) = \{tr \in \Sigma^* \mid \exists s' \bullet s \stackrel{tr}{\Longrightarrow} s'\}.$$

The set $traces(s_0)$ of all traces of C, can also be written $traces(C)$.

– *A sequence $pt \in P^*$ of provided events is called a* provided trace *of C at state s, if there exists $tr \in traces(s)$ such that $pt = \pi_1(tr) \upharpoonright P$. And we write $ptraces(tr)$ for $\pi_1(tr) \upharpoonright P$. It is called a provided trace of C, if s is the initial state. The set of all provided traces of C at s is denoted as $ptraces(C, s)$ (or simply $ptraces(s)$)*

$$ptraces(s) = \{ptraces(tr) \mid tr \in traces(s)\}$$

Similarly, we also write $ptraces(C)$ for $ptraces(s_0)$, which is the set of all provided traces of C.

– *Given a trace $tr \in traces(s)$ of s, $\pi_2(tr)$ is a sequence of regular set over R, and we write $rtraces(tr)$ for $conc(\pi_2(tr))$ which is used to denote a regular set that is the concatenation of these regular sets sequentially. For a provided trace $pt \in ptraces(s)$ of C at state s, the set of required traces, or called requirement, of pt is defined as*

$$rtraces(pt) = \bigcup_{\forall tr \bullet ptraces(tr) = pt} rtraces(tr).$$

Any element $rt \in rtraces(pt)$ is called a required trace *of pt. We use $rtraces(s)$ as the regular set of all required traces of provided traces of state s. That is,*

$$rtraces(s) = \bigcup_{pt \in ptraces(s)} rtraces(pt)$$

. Similarly, $rtraces(s_0)$ is also written $rtraces(C)$, representing the regular set of all required traces of component automaton C.

Given a provided trace $pt \in ptraces(s)$, we write $s \stackrel{pt}{\Longrightarrow} s'$, if there exists $tr \in traces(s)$ such that $s \stackrel{tr}{\Longrightarrow} s'$ with $ptraces(tr) = pt$.

In component automaton C_{Alarm} of the Alarm component, we see that

$$\langle (set, Tstart \cdot Tadd), (ring, BlinkOn \cdot SoundOn), (stop, BlinkOff \cdot SoundOff) \rangle$$

is a trace of C_{Alarm}, $pt = \langle set, ring, stop \rangle$ is a provided trace, and the requirement of pt is the regular set $\{\langle Tstart, Tadd, BlinkOn, SoundOn, BlinkOff, SoundOff \rangle\}$.

From the definition, we can get the following propositions directly.

Proposition 3.2.1. *The set $traces(s)$ and $ptraces(s)$ of C at state s are* regular *and* prefix closed*.*

Proof. We take the state s as the initial state and all states are accepting states. The statement is true from the classic automaton theory. □

However, the regular set *rtraces*(s) is not *prefix closed.* For example, in C_{Alarm}, the prefix $\langle Tstart\rangle$ of the required trace $\langle Tstart, Tadd, BlinkOn, SoundOn, BlinkOff, SoundOff\rangle$ is not a required trace.

Next, we consider an service component that contains non-determinism during requiring services from the environment.

Example 3.2.2. *We consider the component presented in Fig. 3.3. The set of provided events are login, print, and read. There is an internal event ; wifi. These model logging into the system, printing a document, reading emails, and automatically connecting to wifi, respectively. The required events are unu1, unu2, conmail, and conprint, which model connecting to wifi spot unu1, unu2, email server, and printer, respectively.*

Email service is available whenever the component is connected to the internet. However, Printer service is only available when the component is connected with wifi spot unu1.

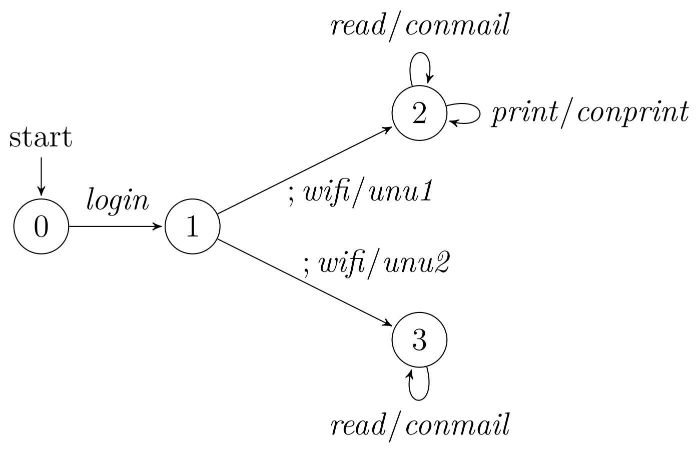


Figure 3.3: Component C_{ic}

In the component model of Fig. 3.3, one possible execution is

$e = \langle 0, (\mathit{login}, \{\epsilon\}), 1, (; \mathit{wifi}, \{\mathit{unu1}\}), 2, (\mathit{read}, \{\mathit{conmail}\}), 2, (\mathit{print}, \{\mathit{conprint}\}), 2\rangle.$

In execution e, $pt = \langle \mathit{login}, \mathit{read}, \mathit{print}\rangle$ is a provided trace and the set of required traces of pt is $\mathit{rtraces}(pt) = \{\langle \mathit{unu1}, \mathit{conmail}, \mathit{conprint}\rangle\}$.

The above examples are about automata-based models of *components*. Next we build the several closed component automata.

Example 3.2.3. *We first take a look at the codes of Blink, and Sound (seen in Figure 3.4). For simplicity, we only select the provided methods needed in this example. In Java API, exceptions are raised when the methods are not invoked properly during run time. Here, we try to build a model which presents the protocols and blocks the illegal invocations. The methods provided by Blink component are abstracted as* BlinkOn *and* BlinkOff. *Similarly,* SoundOn *and* SoundOff *are provided events in Sound component. The corresponding component automata are graphically shown in Figure 3.5.*

```
public class Blink{
  void turnOn();
  void turnOff();
}
public class Sound{
  void turnOn();
  void turnOff();
}
```

Figure 3.4: Timer, Blink, and Sound Component

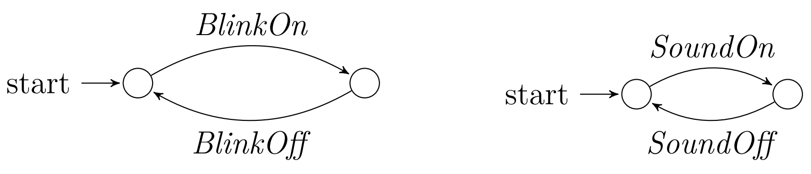


Figure 3.5: Component automaton C_{Blink} and C_{Sound}

The set of provided traces of C_{Blink} is $(\mathit{BlinkOn} \cdot \mathit{BlinkOff})^* \cdot (\epsilon + \mathit{BlinkOn})$. The set of provided traces of C_{Sound} is $(\mathit{SoundOn} \cdot \mathit{SoundOff})^* \cdot (\epsilon + \mathit{BlinkOn})$.

3.3 Component Interface Automata

3.3.1 Motivation

Component automata are abstract models of the implementation of components and specify all the possible behaviors which may be performed by the component during interaction with the environment. However, if components are not designed or implemented properly, they may contain errors which would cause problems when we use services provided by these components. For example, there is a bad-designed Blink component shown in Figure 3.6. In C_{BBlink}, the provided trace $\langle \mathit{BlinkOn}, \mathit{BlinkOff} \rangle$ may be blocked during run time, because the component may transit to state 2 at which *BlinkOff* is not available. The internal transitions can cause non-determinism which will also cause errors. For example the provided service *print* is not available, if wifi spot *unu1* is connected. A loop of the internal events, will also cause unstability or errors. Let's consider a bad-designed alarm component shown in Figure 3.7. The *change* may run internally and change the alarm time. Another cause for errors is the failure transitions that lead the component to the error state.

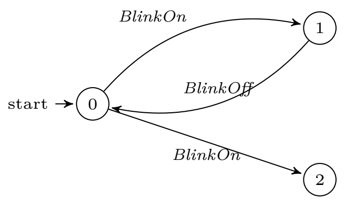


Figure 3.6: Component automaton C_{BBlink} of a bad Blink

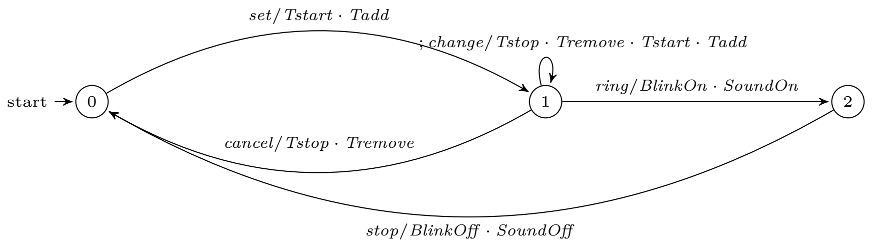


Figure 3.7: Component automaton $C_{badAlarm}$ of a bad Alarm

3.3.2 Non-blockableness

In this section, we will study the issues about what kinds of provided traces are *non-blockable*, what kinds of component automata can only provide non-blockable services, called *component interface automata*, whether there is a way to produce a component interface automaton for a given component automaton while some important properties are preserved.

Intuitively, if a provided event a is *non-blockable* at state s then any invocation to a cannot be blocked, when the automaton is at state s.

Before giving the formal definition of non-blockable events, we illustrate the idea by some more examples. For example, consider state 0 of the component automaton shown on the left of Fig. 3.8. From the viewpoint of the environment, the component automaton may be at 0 or 1, because there is an internal transition from 0 to 1. We assume that after some time, the component will eventually move to state 1, because 0 is not a stable state. So we can see that event $c \in out^P(0)$ is *blocked* at state 0, because $c \notin out^P(1)$. However, if the environment requires b, the component can react to this invocation successfully, that is, b is non-blockable at state 0. The non-blockable provided events are determined by the internally reachable stable states. In the component automaton shown on the right part of Fig. 3.8, there are no internally reachable stable states from state 0, therefore the set of its non-blockable provided events is empty. We should also assure the component to avoid the error or divergent state, so any events which may lead the automaton to such states are also blockable.

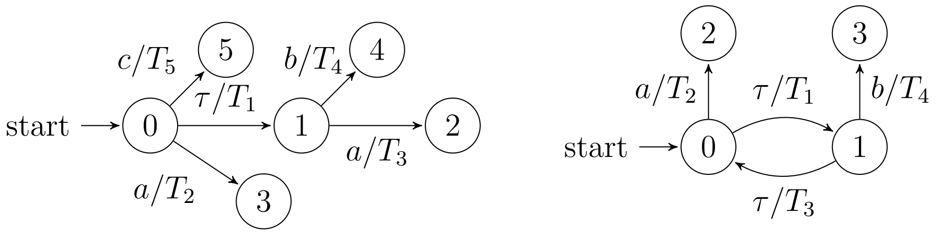


Figure 3.8: Non-Blockable Events

Unless otherwise stated, the following concepts are defined in component automaton $C = (S, s_0, f, P, R, A)$.

We say a state is stable if there is no possible internal transition from it.

Definition 3.3.1 (Stable state). *A state s is* stable*, if $out(s) \subseteq P$.*

In the component automaton C_{ic} in Figure 3.3, state 0, 2, and 3 are stable. In the automaton shown in the left part of Figure 3.8, states 1, 2, 3, 4, and 5 are stable.

Next, we define the kinds of states that can be *internally reached* from certain state, if there exists a sequence of internal transitions between these two states.

Definition 3.3.2 (Internally reachable). *State s' is* internally reachable *from state s, if there exist transitions $s \overset{tr}{\Longrightarrow} s'$ such that $\pi_1(tr) \in A^*$. We write $internal(s, s')$, if s' is internally reachable from s.*

The set $intR(s)$ of all states that are internally reachable from state s is defined as

$$intR(s) = \{s' \in S \mid internal(s, s')\} \cup \{s\}$$

Note that the internally reachable state from s contains itself, that is $s \in intR(s)$.

In component automaton shown in the left part of Figure 3.8, state 1 is internally reachable from state 0. In the component automaton on the right part, $intR(0)$ consists of states 0 and 1.

State s is *divergent*, if there exists a loop of internal transitions at s or s can transit to such kinds of states via a sequence of internal transitions.

Definition 3.3.3 (Divergent state). *State s is* divergent*, if $internal(s, s)$ or there exists state s' such that $internal(s, s')$ and $internal(s', s')$.*

We use **div** *to represent any divergent states.*

In component automaton $C_{badAlarm}$ of the bad Alarm component , state 1 is divergent. And both states 0 and 1 in component automaton shown in the right part of Figure 3.8 are divergent.

When a component is in divergent states, it may stay in such states forever without responding to any invocations from outside. Even when considering *fairness* conditions, we cannot assure how long the component can respond to other invocations. Therefore, here we should try to make component avoid divergent states in the interface models.

For state s, we use notation $intR^s(s)$ for the set of stable states that are internally reachable from state s, except that the set is empty when the states can internally reach a divergent state. That is,

$$intR^s(s) = \begin{cases} \emptyset, & \text{if } \mathbf{div} \in intR(s) \\ \{s' \text{ is stable} \mid s' \in intR(s)\}, & \text{otherwise} \end{cases}$$

We call a provided event *illegal*, if it can lead the automaton to the error or divergent states.

Definition 3.3.4 (Illegal provided event). *A provided event a at state s is called* illegal*, if it can internally lead the automaton to the error or divergent state. The set of illegal provided events of state s is defined:*

$$\text{ill}(s) = \{a \mid s \overset{a}{\Longrightarrow} t,\ t \textit{ is error or divergent}\}$$

Now we define the provided events that can not be blocked at given state. A provided event is *non-blockable* at state s, if it is available at any stable states that internally reached from state s, and a cannot lead the component automaton to any divergent state or the error state.

Definition 3.3.5 (Non-blockable provided events). *For any $s \in S$, the set of non-blockable provided events at state s is*

$$\text{non-blockable}(s) = \bigcap_{r \in intR^s(s)} (out^P(r) \setminus \text{ill}(r))$$

It is trivial that non-blockable(s) is empty, when s is divergent.

For example, in $C_{badAlarm}$, the non-blockable provided events at the initial state s is empty, the *set* will lead the component into the divergent state 1. In the component automaton shown in the left part of Figure 3.8, both a and b are *non-blockable* at state 0 or 1.

The non-blockable provided events assure that a single provided event is non-blockable at a given state. Next, we would study whether a given provided trace can be blocked or not by the component, as long as all the requirement of the provided trace is satisfied. This is especially important when the component is closed or an independent team is developing or searching the component that would provide all the services that are required here.

Let's consider the component automaton shown in Fig. 3.9. We can see that a is non-blockable at state 0, but, after the invocation of a, the component determines whether to move to state 1 or 3 by requiring services in T_1 or T_2. So from the view of the clients, both of b and c may be blocked after a. This can be taken from of view of a game between the component and its environment. The provided events are determined or chosen by the environment, while the component decides which requirement to choose. We say a provided trace is *non-blockable*, if every prefix of it is non-blockable and the sequent provided event is non-blockable.

Definition 3.3.6 (Non-blockable trace). *A provided trace $\langle a_1, \cdots, a_k\rangle$ of state s with $k \geq 0$ is* non-blockable *at state s, if for any $1 \leq i \leq k$, $a_i \in$ non-blockable(s') for any s' such that $s \xRightarrow{\langle a_1, \cdots, a_{i-1}\rangle} s'$.*

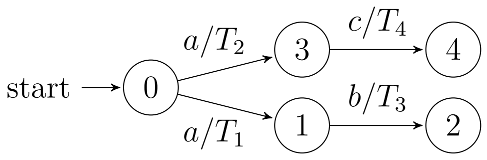


Figure 3.9: Non-blockable traces

A trace tr is non-blockable *at s, if $ptraces(tr)$ is non-blockable at s.*

We use $uptraces(s)$ and $utraces(s)$ to denote the set of all non-blockable provided traces and non-blockable traces at state s, respectively. $uptraces(s)$ and $utraces(s)$ are also written as $uptraces(C)$ and $utraces(C)$, respectively, when s is the initial state.

For example in component automaton of Figure 3.9, $\langle a\rangle$ is non-blockable, while both $\langle a, c\rangle$ and $\langle a, b\rangle$ may be blocked.

3.3.3 Component Interface Automata

The abstract model of components are not suitable as interfaces for third party composition, if it contains traces that may be blocked during run-time. In this part, we study the *interface property* Θ, which is the standard or criteria for whether a component automaton can be used as interface model or not.

Now, we introduce input-determinism and prove that it is equivalent with Θ. We then develop an algorithm to check whether a component automaton satisfies the interface property or not. Then we present an algorithm to construct an component interface automaton $\mathcal{I}(C)$ for any given component automaton C, such that $\mathcal{I}(C)$ and C have the same non-blockable traces.

Definition 3.3.7 (Component interface automaton). *The interface property Θ states that all provided traces are non-blockable. A component automaton C is called* component interface automaton *(or* interface automaton *for short), if it satisfies Θ. That is, all provided traces of C are non-blockable and f or* **div** *cannot be internally reachable from s_0.*

We usually use I with proper sub or super scripts to denote a component interface automaton.

It directly follows that all traces of a component interface automaton are non-blockable.

We can see that component automata C_{Alarm}, C_{Blink}, and C_{Sound} are also component interface automata.

However, it is not easy to check whether a component automaton C satisfies Θ. Then we introduce property *input-determinism*, which is slightly different from the traditional definition. In the traditional definition, input-determinism means that at given state, with same input/provided event, the next state is determined. We say a component automaton is *input-deterministic*, if the set of non-blockable events are always same at any state that is reached by the same provided trace from the initial state.

Definition 3.3.8 (Input-determinism). *Given a component automaton C, we say it is* input-deterministic *if neither f or* **div** *is reachable from s_0, and for any provided trace pt, states s_1 and s_2 such that*

$$s_0 \overset{pt}{\Longrightarrow} s_1 \textit{ and } s_0 \overset{pt}{\Longrightarrow} s_2,$$

then

$$\text{non-blockable}(s_1) = \text{non-blockable}(s_2).$$

That is, each state of set $\{s \in S \mid s_0 \overset{pt}{\Longrightarrow} s\}$ has the same non-blockable provided events.

The following theorem states that all the traces of an input-deterministic component automaton are non-blockable, and vice versa. That is, *input-determinism* is equivalent with Θ.

Theorem 3.3.1. *A component automaton C is input-deterministic iff C is a component interface automaton.*

Proof. Both input-determinism and interface property require that f or **div** is not internally reachable from the initial state. So we only need to prove the following.

First, we prove the direction from left to right. From the input-determinism of C, it follows that for each provided trace $pt = (a_0, \ldots, a_k)$ and for each state s with $s_0 \overset{tr}{\Longrightarrow} s$ and $\pi_1(tr) = \langle a_0, \ldots, a_i \rangle$ for $0 \leq i \leq k-1$, the set non-blockable(s) is the same. Since pt is a provided trace, so $a_{i+1} \in$ non-blockable(s). This shows that all provided traces are non-blockable, and so all traces are non-blockable too.

Second, we prove the direction from right to left by contraposition. We assume that C is not input-deterministic, so there exist traces tr_1 and tr_2 with $ptraces(tr_1) = ptraces(tr_2)$ and $s_0 \overset{tr_1}{\Longrightarrow} s_1$, $s_0 \overset{tr_2}{\Longrightarrow} s_2$ such that non-blockable$(s_1) \neq$ non-blockable(s_2).

Then, we assume that there is a provided event a such that $a \in$ non-blockable(s_1) and $a \notin$ non-blockable(s_2). Now $ptraces(tr_2) \cdot a$ is a provided trace of C, but it is blockable. □

In the following, we present Algorithm 1 to check whether C satisfies the interface property or not. Based on Theorem 3.3.1, the idea is check whether the component automaton is input-deterministic or not. The procedure is to construct the set of states that can be reached by the same provided trace. The algorithm will return **true** or **false**, representing $C \models \Theta$ and $C \not\models \Theta$, respectively. If there exists two states in such set that their non-blockable provided events are different, then the algorithm will return **false**. After traversing all these sets successfully, the algorithm return with **true**.

Algorithm 1: Check whether $C \models \Theta$ or $C \not\models \Theta$

```
Input: $C = (S, s_0, f, P, R, A, \delta)$
Output: true or false
1: if $f$ or div $\in intR(s_0)$ then
2:   return false
3: end if
4: Initialization: $Q_0 = \{s' \mid s' \in intR(s_0)\}$; $todo := \{Q_0\}$; $done := \emptyset$
5: while $todo \neq \emptyset$ do
6:   choose $Q \in todo$; $todo := todo \setminus \{Q\}$; $done := done \cup \{Q\}$;
7:   if $\exists s_1, s_2 \in Q \bullet$ non-blockable$(s_1) \neq$ non-blockable$(s_2)$ then
8:     return false
9:   end if
10:  for each $a \in \bigcup_{s \in Q}$ non-blockable$(s)$ do
11:    $Q' := \bigcup_{s \in Q} \{s' \mid s \overset{a}{\Longrightarrow} s'\}$
12:    if $Q' \notin (todo \cup done)$ then
13:      $todo := todo \cup \{Q'\}$
14:    end if
15:  end for
16: end while
17: return true
```

Now, we give and prove the correctness of Algorithm 1. We show that the algorithm will always terminate and the complexity is exponential with the size of state set in worst case. The return result **true** and **false** represents that component automaton C is input-deterministic or not.

Theorem 3.3.2 (Correctness of Algorithm 1). *Given component automaton C, the algorithm $\mathcal{Y}(C)$*

1. *$\mathcal{Y}(C)$ terminates and the complexity of time in the worst case is exponential with the size of state set S.*
2. *C is input-deterministic iff $\mathcal{Y}(C)$ is* **true**

Proof. The proof is:

1. The size of set 2^S is that of $2^{|S|}$, where $|S|$ is the size of state set S. $2^{|S|}$ is finite because $|S|$ is finite. From Line 4, 6, 12, and 13, we see that *todo* is disjoint with *done*, and the size of *done* increases in every iteration of the while-loop (Line 5-24). Besides, the union $done \subseteq 2^S$. So *todo* will eventually be empty, which means the algorithm will terminate. The worst case is $todo \cup done = 2^S$.
2. we prove the contrapositive statement that C is not *input-deterministic*, iff $\mathcal{Y}(C)$ is **false**.
   - Direction from left to right: if C is not input-deterministic, then there will be two cases: 1). f or **div** is reachable from s_0. we can see the $\mathcal{Y}(C)$ returns **false** at line 1-2; 2). there exists provided trace pt, s_1, and s_2 such that $s_0 \overset{pt}{\Longrightarrow} s_1$ and $s_0 \overset{pt}{\Longrightarrow} s_2$, then non-blockable$(s_1) \neq$ non-blockable(s_2). Let $pt = \langle a_0, \ldots, a_k \rangle$, during every iteration of while-loop, in Line 10, we a_i for $0 \leq i \leq k$ and choose the newly added set Q' in Line 6. Then the algorithm will either return **false**, or there exists Q' that both s_1 and s_2 are in Q'. Then from Line 7, the algorithm will return with **false**.
   - Direction from right to left: if it returns from Line 2, then obviously, C is not input-deterministic. If the algorithm returns **false** at line 8, then there exists a set Q which consists of two state s_1 and s_2 such that non-blockable$(s_1) \neq$ non-blockable(s_2). From Line 11, we trace set Q back to Q_0, and get a provided trace pt such that $s_0 \overset{pt}{\Longrightarrow} s_1$ and $s_0 \overset{pt}{\Longrightarrow} s_2$. So we prove that C is not input-deterministic.

   From the two directions above, we prove that C is input-deterministic iff the algorithm returns **true**

□

In the following, we present an algorithm (see given in Algorithm 2) that, given component automaton C, constructs an interface automaton $\mathcal{I}(C)$ which shares the same non-blockable traces with C.

The aim of this algorithm is to filter out the transitions which may be blocked. We will consider the component automaton C_{foo} graphically shown in Figure 3.10 to illustrate the general idea of constructing a component interface automaton from C_{foo}. First, it is obvious that $\langle b, c\rangle$, and $\langle a, c, a\rangle$ are all blockable. The transition $1 \xrightarrow{c/T_3} 2$ is not blockable if the previous transitions is labeled with a/T_1, and blockable if it is a b transition. This implies that we cannot obtain the interface automaton by removing the possible blockable transitions directly. State 1 needs to be split into different states and these states preserve the non-blockable transitions. Inspired by Algorithm 1, we build set $Q \subseteq S$ which is a set of states that are reachable via any non-blockable provided trace pt from the initial state. Then provided events, which are non-blockable at each state of Q, are non-blockable after provided trace pt. For example, after a-transition, C_{foo} may be in any state of $\{1, 3\}$; after b-transition, it may be $\{1, 4\}$. The next transition of state 1 can be divided based on non-blockable provided events of $\{1, 3\}$ and $\{1, 4\}$. So, we use a pair of a state and a state set as the notion for the states in the expected interface automaton. Here, $(\{1, 3\}, 1)$ and $(\{1, 4\}, 1)$ are states in the interface automaton. The transitions of $(\{1, 3\}, 1)$ preserve the transition of 1 by filtering out the provided events that are blockable in any state of $\{1, 3\}$.

The non-blockable provided event of $\{1, 3\}$ is $\{c\}$, and similarly the state set is $\{0, 2\}$ after c-transition from $\{1, 3\}$. So $(\{0, 2\}, 0)$ and $(\{0, 2\}, 2)$ are also states in the interface automaton. Now the following transitions can be added: $(\{1, 3\}1) \xrightarrow{c/T_3} (\{0, 2\}, 2)$ and $(\{1, 3\}3) \xrightarrow{c/T_6} (\{0, 2\}, 0)$.

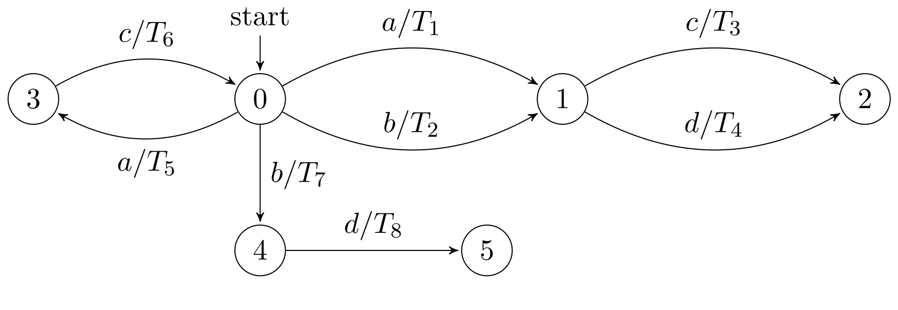


Figure 3.10: Example automaton C_{foo}

Algorithm 2 can traverse all the non-blockable provided events and produce the corresponding states to preserve all the non-blockable transitions. If the error state f or **div** can be reached from the initial state s_0, then the algorithm exits with an empty automaton, which means all provided traces are blockable. Otherwise, each state of $\mathcal{I}(C)$ is of the form (Q, r), where Q is a subset of states of C and $r \in Q$, the initial state is (Q_0, s_0) with $Q_0 = \{s' \mid s' \in intR(s_0)\}$. Q records all reachable states from each state $s' \in Q'$, (suppose (Q', r') has been added as a state of $\mathcal{I}(C)$), by executing a provided event a, where $a \in \bigcap_{s' \in Q'}$ non-blockable(s'). By induction, we can see that all traces of $\mathcal{I}(C)$ are non-blockable. On the other hand, all the states that can be reached from states in Q' via each provided event $b \in \bigcap_{s' \in Q'}$ non-blockable(s') with possible internal events before/after b will consist a Q such that (Q, r) is one state of $\mathcal{I}(C)$. So all non-blockable traces of C are also contained in $\mathcal{I}(C)$ by inductive way.

Algorithm 2: Construction of Interface Automaton $\mathcal{I}(C)$

```
Input: $C = (S, s_0, f, P, R, A, \delta)$
Output: $\mathcal{I}(C) = (S_I, (Q_0, s_0), f, P, R, A, \delta_I)$, where $S_I \subseteq 2^S \times S$
 1: if $f$ or div $\in intR(s_0)$ then
 2:    exit with $\delta_I = \emptyset$
 3: end if
 4: Initialization: $S_I := \{(Q_0, s) \mid s \in Q_0\}$ with $Q_0 = \{s' \mid s' \in intR(s_0)\}$;
    $\delta_I := \emptyset$; $todo := S_I$; $done := \emptyset$
 5: while $todo \neq \emptyset$ do
 6:    choose $(Q, r) \in todo$; $todo := todo \setminus \{(Q, r)\}; done := done \cup \{(Q, r)\}$
 7:    for each $a \in \bigcap_{s \in Q}$ non-blockable$(s)$ do
 8:       $Q' := \{s' \mid s \overset{a}{\Longrightarrow} s', s \in Q\}$
 9:       for each $(r \xrightarrow{a/T} r') \in \delta$ do
10:          if $(Q', r') \notin (todo \cup done)$ then
11:             $todo := todo \cup \{(Q', r')\}$
12:             $S_I := S_I \cup \{(Q', r')\}$
13:          end if
14:          $\delta_I := \delta_I \cup \{(Q, r) \xrightarrow{a/T} (Q', r')\}$
15:       end for
16:       for each $r \xrightarrow{w/T} r'$ with $r' \in Q$ and $w \in A$ do
17:          if $(Q, r') \notin (todo \cup done)$ then
18:             $todo := todo \cup \{(Q, r')\}$
19:             $S_I := S_I \cup \{(Q, r')\}$
20:          end if
21:          $\delta_I := \delta_I \cup \{(Q, r) \xrightarrow{w/T} (Q, r')\}$
22:       end for
23:    end for
24: end while
```

Analysis of Algorithm 2. The main part is while-loop (Line5-24). Besides the initialization statement(Line 4), state set S_I is extended in Line 12 and 19. Provided transitions are added in Line 14 of for-loop (Line 9-15); Internal transitions are added in Line 19 of for-loop (Line 16-22). We use $\mathcal{Q}$ to denote $\{Q \mid (Q,s) \in \mathit{done}\}$ after the algorithm terminates. From Line 12, we can see the state set S_I of $\mathcal{I}(C)$ is $\{(Q,s) \mid Q \in \mathcal{Q}, s \in Q\}$. For simplicity, we write non-blockable$(Q) = \bigcap\limits_{s \in Q}$ non-blockable(s). We now prove the following lemma stating that the set of states that are reached by a non-blockable provided trace is one element in $\mathcal{Q}$

Lemma 3.3.1. *For any non-blockable provided trace pt of C, there exists $Q \in \mathcal{Q}$ such that $Q = \{s \mid s_0 \overset{pt}{\Longrightarrow} s\}$.*

Proof. We prove this by induction on the length of non-blockable provided trace.

The base case follows directly since $Q_0 \in \mathcal{Q}$. Given non-blockable provided trace $pt \cdot a$, by hypothesis induction, there exists $Q \in \mathcal{Q}$ such that $Q = \{s \mid s_0 \overset{pt}{\Longrightarrow} s\}$. By Definition 3.3.6, $a \in \bigcap\limits_{s \in Q}$ non-blockable(s). So, there exists for-loop of Line 7-15 for event a. In Line 8 and Line 10, we can see that $Q' = \{s' \mid s \overset{a}{\Longrightarrow} s', s \in Q\}$ is added. Therefore, $Q' = \{s \mid s_0 \overset{pt \cdot a}{\Longrightarrow} s\}$ and $Q' \in \mathcal{Q}'$.

Thus, the lemma is proved. □

Given a transition $(Q,s) \xrightarrow{a/T} (Q',s')$, we can deduce that $s \xrightarrow{a/T} s'$ and $Q' = \{r' \mid r \overset{a}{\Longrightarrow} r', r \in Q\}$.

From Line 14, it implies that for $Q, Q' \in \mathcal{Q}$ and $a \in$ non-blockable(Q) such that $Q' = \{s' \mid s \overset{a}{\Longrightarrow} s'\}$, if $s \xrightarrow{a/T} s'$ with $s \in Q$ and $s' \in Q'$, then $(Q,s) \xrightarrow{a/T} (Q',s')$. From Line 16-22, it indicates that given $Q \in \mathcal{Q}$, for $s, s' \in Q$ and $w \in A$, if there $s \xrightarrow{w/T} s'$, then $(Q,s) \xrightarrow{w/T} (Q,s')$.

Now we can get the following lemma stating that the non-blockable transitions are preserved in the $\mathcal{I}(C)$.

Lemma 3.3.2. *Given a non-blockable trace tr, if $s_0 \overset{tr}{\Longrightarrow} s$, there exists $(Q_0,s_0) \overset{tr}{\Longrightarrow} (Q,s)$.*

Proof. We prove this by induction on the length of tr. The base case follows directly from Line 16-22. Consider non-blockable trace $tr \cdot (w,T)$ of C, then there exists $s_0 \overset{tr}{\Longrightarrow} s_1$ and $s_1 \xrightarrow{w/T} s_2$. We let pt be $\pi_1(tr) \upharpoonright P$. From Lemma 3.3.1, there exists $Q \in \mathcal{Q}$ such that $Q = \{s \mid s_0 \overset{pt}{\Longrightarrow} s\}$. By hypothesis induction, there exists $(Q_0, s_0) \overset{tr}{\Longrightarrow} (Q, s_1)$. If $w \in A$, then $s_2 \in Q$, so $(Q_0, s_0) \xRightarrow{tr\cdot(w,T)} (Q, s_2)$. If $w \in P$, then there exists $Q' = \{s' \mid s \overset{w}{\Longrightarrow} s', s \in Q\}$, due to $tr \cdot (w, T)$ is non-blockable. That means $(Q, s_1) \xrightarrow{w/T} (Q', s_2)$. So, $(Q_0, s_0) \xRightarrow{tr\cdot(w,T)} (Q', s_2)$.

Thus, the lemma is proved. □

Next, the next lemma shows that for any provide trace of $\mathcal{I}(C)$, the reached states are in same $Q \in \mathcal{Q}$.

Lemma 3.3.3. *For any provided trace pt, if there exists $(Q_0, s_0) \overset{pt}{\Longrightarrow} (Q_1, s_1)$ and $(Q_0, s_0) \overset{pt}{\Longrightarrow} Q_2, s_2$, then $Q_1 = Q_2$.*

Proof. We prove this by induction on length *pt*.

The base case follows directly because Q_0 is set. We consider $pt \cdot a$ such that $(Q_0, s_0) \overset{pt\cdot a}{\Longrightarrow} (Q_1, s_1)$ and $(Q_0, s_0) \overset{pt\cdot}{\Longrightarrow} (Q_2, s_2)$. By induction hypothesis, there exists Q such that $(Q_0, s_0) \overset{pt}{\Longrightarrow} Q, s_1'$ and $(Q_0, s_0) \overset{pt}{\Longrightarrow} (Q, s_2')$. Then there exists $(Q, s_1') \overset{a}{\Longrightarrow} (Q_1, s_1)$ and $(Q, s_2') \overset{a}{\Longrightarrow} (Q_2, s_2)$. From Line 14 and Line 7, it is obvious that $a \in \bigcap_{s \in Q}$ non-blockable(s). Therefore, $Q_1 = Q_2 = \{s' \mid s \overset{a}{\Longrightarrow} s', s \in Q\}$. □

Correctness of Algorithm 2 is given formally in the following theorem that the algorithm will terminate, the result $\mathcal{I}(C) \models \Theta$, and $\mathcal{I}(C)$ preserves all the non-blockable traces of C.

Theorem 3.3.3 (correctness of Algorithm 2)**.** *The following properties holds for Algorithm 2, for any component automaton C:*

1. *The algorithm always terminates. Complexity for worst case is $\mathcal{O}(2^{|S|} * |S|)$*
2. *$\mathcal{I}(C)$ is input deterministic,*
3. *utraces(C) = utraces$(\mathcal{I}(C))$.*

Proof. The proof is:

1. It is similarly to that of Theorem 3.3.2. The termination of the algorithm can be obtained because *todo* will be eventually empty, because the set *done*

increases for each iteration of the loop in the algorithm, and the union of *done* and *todo* is bounded by S_I which is obviously finite. The size of state set S_I is limited by $2^{|S|} * |S|$, and one state is handled during every while loop, so the complexity is $\mathcal{O}(2^{|S|} * |S|)$.

2. If f or **div** is internally reachable from s_0, then $\mathcal{I}(C)$ is empty, so it is input-deterministic.

   From Lemma 3.3.3 and Definition 3.3.8, it is deduced that $\mathcal{I}(C)$ is *input-deterministic.*

3. From Lemma 3.3.2 and input-determinism of $\mathcal{I}(C)$, we see that C and $\mathcal{I}(C)$ have the same non-blockable traces. That is, $utraces(C) = utraces(\mathcal{I}(C))$.

□

Now, consider component automaton C_{foo} shown in Figure 3.10. We build the interface automaton $\mathcal{I}(C_{foo})$, graphically shown in Figure 3.11. The $\mathcal{Q}$ includes $\{0\}$, $\{1,4\}$, $\{1,3\}$, $\{0,2\}$, and $\{2,5\}$. It is obvious that $\mathcal{I}(C_{foo})$ is input-deterministic and preserves all the non-blockable traces of C_{foo}.

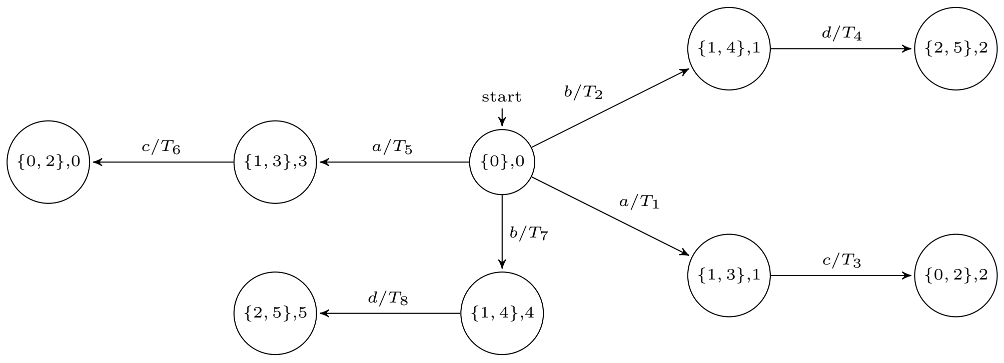


Figure 3.11: Interface automaton $\mathcal{I}(C_{foo})$

Example 3.3.1. *In the internet connection component automaton 3.3, the provided trace $\langle login, read\rangle$ is non-blockable but $\langle login, print\rangle$ may be blocked during execution, because after login is called, the component may transit to state* 3 *at which print is not available. Algorithm 2 generates the interface model of C_{ic}, shown in Fig. 3.12.*

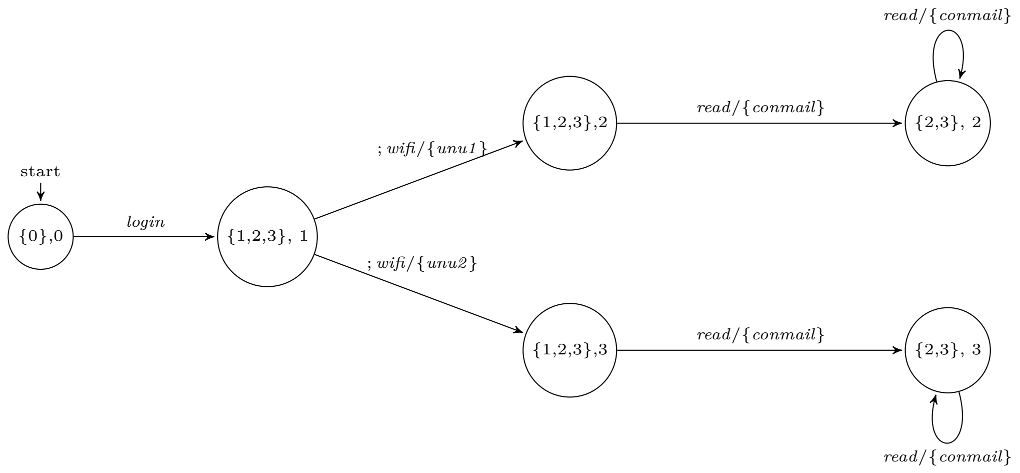


Figure 3.12: Interface model of Component C_{ic}

Again, let's see the automaton shown in Figure 3.9 that we know that c is non-blockable at state 3, but c is blockable globally in the sense that there exists a trace a that $0 \xrightarrow{a} 3$, but $\langle a, c\rangle$ is blockable. We call these events, refusals, that are non-blockable at certain states but may be blocked in traces.

Definition 3.3.9 (Refusals). *Given component automaton C, for each state s of C, there may be several sets Q such that (Q, r) is a state of $\mathcal{I}(C)$. Then, the refusals of state s is*

$$\textit{refusals}(s) = \text{non-blockable}(s) \setminus \bigcup_{(Q,s)\in S_I} \text{non-blockable}(Q, s)$$

Obviously, for input-deterministic automaton, the refusal set at each of its states is empty. We get the proposition that, for any non-blockable provided trace, if it reaches this state, then concatenation with the non refusal event will be non-blockable. .

Proposition 3.3.1. *It directly follows from the definition of refusals:*

1. *For any state s of a component automaton, $\textit{refusals}(s) = \emptyset$,*
2. *For any non-blockable provided trace pt such that $s_0 \overset{pt}{\Longrightarrow} s$. then for any $a \in \text{non-blockable}(s) \setminus \textit{refusals}(s)$, $pt \cdot a$ is non-blockable.*

Remarks. The component interact with its environment in the game way that provided events are determined by the environment and requirement is determined by the component. With the purpose to provide services that are not blockable, we restrict the interface model to provide only non-blockable provided traces. There are two possible uses in interface-based design or component reuse.

– In $\mathcal{I}(C)$, a lot of services may be disabled. This will help the developers to choose another component to reuse or try to develop a component with better interface models.

– In view of independent development or separate of concern, the developers, responsible for searching or developing the required services, can focus on their work on fulfilling the requirement without worrying about affecting the provided services of the component.

3.4 Composition Operation

“Components are for composition” [Szy02]. A component interacts with other components by providing services and requiring services. A closed component provides services without the need to require services from other components. We consider closed components as stable service providers, while open components will provide services under assumptions that the required services are guaranteed. Composition allows for components interacting with each other to build up new components. It also enables reusability and decomposition of components. In this section, we will introduce some basic composition operators of component automata.

Components are composed by connecting the required services of one component to compatible provided services of another component. For example, the Alarm component synchronizes with components Timer, Blink, and Sound by method invocation.

Component automata are composed by synchronizing on *shared events* which are required events of one and provided events of the other.

The requirement of a given provided/internal event is modeled as regular set over required events. Therefore, firstly, we present the composition between the requirement of a given provided/internal events with a component automaton.

3.4.1 Product of Component Automata and Finite State Machine

For a set of sequences T, we use $\mathcal{M}(T)$ to denote the minimal deterministic finite state machine which recognize T [HMU79]. And, $\mathcal{M}(T)$ is of the form $(Q, \Sigma, q_0, F, \sigma)$, where

- Q is the finite set of states,
- $q_0 \in Q$ is the initial state, and $F \subseteq Q$ is the set of final states,
- Σ is the input alphabet,
- $\sigma \subseteq Q \times \Sigma \times Q$ are the transitions.

Suppose a component automaton C_1 with a transition $s_1 \xrightarrow{a/T} s_2$, which means C_1 requires all the traces in T in order to provide a, and another component automaton C_2 that can provide these required services in T. We define *internal product* between $\mathcal{M}(T)$ and C_2 to implement the synchronization of C_1 and C_2 on T, and calculate the new required traces for a. This composition operation is partial, because provided traces of C_2 may not be able to satisfy the requirement specified in

T. If the requirement is satisfied, we denote $C_2 \models_{shared} \mathcal{M}(T)$, and $C_2 \not\models_{shared} \mathcal{M}(T)$ otherwise.

Before giving the formal definition, we illustrate the intuitive idea by one example.

Example 3.4.1. *In Fig. 3.13, the required trace is $\{\langle a, b, c\rangle\}$, and the component automaton provides $\langle a, b\rangle$. And the shared set is $\{a, b\}$. $C \models_{\{a,b\}} M$, and required traces of the composition are $\{\langle x, y, z, c\rangle, \langle x', y, z, c\rangle\}$.*

However, $C \not\models_{\{a,b,c\}} M$, because c is not provided by component C. And also $C \not\models_{\{a,b,x\}} M$, because x can not be provided to component C.

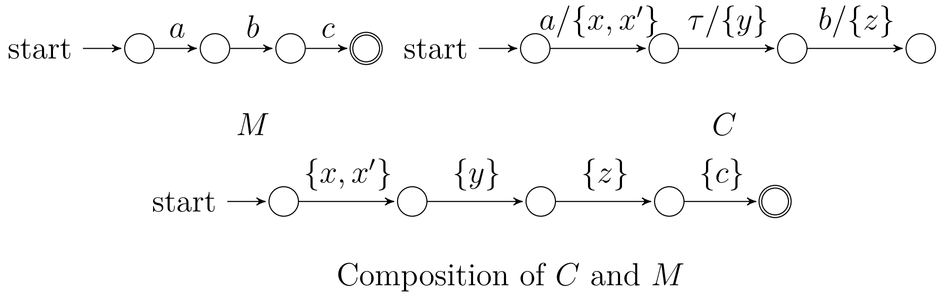


Figure 3.13: Product (1)

Given a finite state machine M and component automaton C, assume that their current states are q and s, respectively and there exists transition $q \xrightarrow{a} q'$ with $a \in$ *shared* of finite state machine M, then provided event a is expected of C. However, if $a \notin$ non-blockable(s), this causes the request of a fail. If $a \in$ non-blockable(s), but there exists transition $s \xrightarrow{a/T} s'$ that T also requires services of *shared*, the invocation of a also fails, because this is deadlock caused by cyclic invocation.

We use *events* to denote the alphabet set for any set or sequences, and for example, *events*(T) means the set of events appearing in T.

Now, we define the composition operation, called *internal product*, between component automaton C and finite state machine M under a given set of events *shared*. It is called internal, because, finite state machine represents a regular set of required events issued from another component which is internally determined by that component.

Definition 3.4.1 (Internal product). *Given a component automaton C and a finite state machine M, the composition of C and M under the event set shared is $C \lhd_{shared} M = (Q', \Sigma', \sigma', q_0, F')$, where*

- *$Q' = S \times Q$, $q'_0 = (s_0, q_0)$;*
- *$\Sigma' = \Sigma_1 \cup \Sigma_2$ where Σ_1 is any set of regular sets over R, and Σ_2 is a set of singleton $\{a\}$ where $a \in (\Sigma \setminus shared)$;*
- *σ' is the smallest set given by the following rules:*
  For reachable state $(s_1, r_1) \in Q'$, and if $s_1 \xrightarrow{a/T} s_2$ with $a \in E \cup A$ and $t_1 \xrightarrow{b} t_2$.
  - *if $events(T) \cap shared \neq \emptyset$ or $b \in shared \wedge b \notin \text{non-blockable}(s_1)$, then $C \not\models_{shared} M$ and exit;*
  - *otherwise if $a \in A$, $(s_1, r_1) \xrightarrow{T} (s_2, r_1) \in \sigma'$;*
  - *otherwise if $b \notin shared$, $(s_1, r_1) \xrightarrow{\{b\}} (s_1, r_2) \in \sigma'$;*
  - *otherwise if $a == b$, $(s_1, r_1) \xrightarrow{T} (s_2, r_2) \in \sigma'$;*
- *$F' = \{(s, r) \mid s \in S, r \in F\}$.*

In the above definition of σ', the extended version of sequence of transitions is, by induction, for reachable state (s, r), and transitions $s \overset{tr}{\Longrightarrow} s'$ of C, $r \overset{\alpha}{\Longrightarrow} r'$ of M, if $ptraces(tr)$ is non-blockable and

$$ptraces(tr) = \alpha \upharpoonright (shared), \text{ and } events(\pi_2(tr)) \cap shared = \emptyset$$

then in the composition, there exists$(s, r) \overset{sq}{\Longrightarrow} (s', r')$ such that

$$sq \upharpoonright \Sigma_1 = \pi_2(tr) \upharpoonright \Sigma_1, \text{ and } sq \upharpoonright \Sigma_2 = \mathcal{R}(\alpha) \upharpoonright \Sigma_2$$

where $\mathcal{R}(\alpha)$ is a sequence obtained by replacing every element a of sequence α by $\{a\}$. Then we get the following lemma.

We deduce the condition for $C \models_{shared} M$ from the composition operation, that is, the conditions for how the automaton C satisfies the requirement declared as M under the share event set . We use $\mathcal{L}(M)$ for the set of sequences that are recognized by M. The lemma states that component C can satisfy requirement M under *shared*, iff the requirement specified by M under *shared* can be satisfied non-blockably by the component without the need to require any services in *shared*.

Lemma 3.4.1. *$C \models_{shared} M$, if and only if*

1. *$\mathcal{L}(M) \upharpoonright shared \subseteq uptraces(C)$, and*
2. *For every $pt \in \mathcal{L}(M) \upharpoonright shared$, $rtraces(pt) \cap shared = \emptyset$.*

Proof. The proof is:

1. Direction " $\Longrightarrow$ " by contraposition.

   Assume that $\mathcal{L}(M) \upharpoonright \mathit{shared} \nsubseteq \mathit{uptraces}(C)$, so there exists $\alpha \cdot a \in \mathcal{L}(M) \upharpoonright \mathit{shared}$ and $\alpha \in \mathit{uptraces}(C)$, but $\alpha \cdot a \notin \mathit{uptraces}(C)$. Then there will be a reachable state (s, r) such that $a \notin$ non-blockable(s) and $r \xrightarrow{a} r'$, that implies $C \not\models_{\mathit{shared}} M$.

   Assume that there exists $pt \cdot a \in \mathcal{L}(M) \upharpoonright \mathit{shared}$ and for all $\mathit{rtraces}(pt) \cap \mathit{shared} = \emptyset$, but $\mathit{rtraces}(pt \cdot a) \upharpoonright \mathit{shared} \neq \emptyset$. This implies that there exists a reachable state (s, r) such that $s \xrightarrow{a/T} s'$ and $r \xrightarrow{a} r'$ with $\mathit{events}(T) \cap \mathit{shared} \neq \emptyset$, so $C \not\models_{\mathit{shared}} M$.

2. Direction " $\Longleftarrow$ " by contraposition.

   If $C \not\models_{\mathit{shared}} M$, from Definition 3.4.1, there exists reachable state (s, r) that $s \xrightarrow{a/T} s'$ where $\mathit{events}(T) \cap \mathit{shared} \neq \emptyset$, then there exists $pt \in \mathcal{L}(M) \upharpoonright \mathit{shared}$ and $\alpha \in \mathit{rtraces}(pt)$ that $\mathit{events}(\alpha) \cap \mathit{shared} \neq \emptyset$.

   Or, $r \xrightarrow{b} r'$ with $b \in \mathit{shared}$, but $b \notin$ non-blockable(s), then there exists $\beta \in \mathcal{L}(M) \upharpoonright \mathit{shared}$ but $\beta \notin \mathit{uptraces}(C)$.

Thus, the lemma is proved. □

The following lemma presents the sufficient and necessary condition for a sequence to be accepted by the internal product, if $C \models_{\mathit{shared}} M$.

Lemma 3.4.2. *In the composition $C \lhd_{\mathit{shared}} M$ such that $C \models_{\mathit{shared}} M$, sequence sq of Σ^* accepted by $C \lhd_{\mathit{shared}} M$, if and only if there exists $\alpha \in \mathcal{L}(M)$ and $tr \in \mathit{utraces}(C)$ such that*

$$\mathit{ptraces}(tr) \upharpoonright P = \alpha \upharpoonright \mathit{shared}$$

$$sq \upharpoonright \Sigma_1 = \pi_2(tr) \upharpoonright \Sigma_1 \text{ and } sq \upharpoonright \Sigma_2 = \mathcal{R}(\alpha) \upharpoonright \Sigma_2.$$

Proof. For sequence sq, let $sq \upharpoonright \Sigma_1$ and $sq \upharpoonright \Sigma_2$ be $\langle T_0, \ldots, T_m \rangle$ and $\langle \{a_0\}, \ldots, \{a_n\} \rangle$, respectively. The proof is:

– Direction " $\Longrightarrow$ " :

  Since $sq \in \mathcal{L}(C \lhd_{\mathit{shared}} M)$, then there should exist α that $\alpha \in \mathcal{L}(M)$. From Lemma 3.4.1, $\alpha \upharpoonright \mathit{shared} \in \mathit{uptraces}(C)$. Then, there exists $tr \in \mathit{utraces}(C)$ such that $\mathit{ptraces}(tr) \upharpoonright P = \alpha \upharpoonright \mathit{shared}$.

There exists (s, q) such that $q \in F$ $(s_0, q_0) \xRightarrow{sq} (s, q)$. From Definition 3.4.1, $s_0 \xRightarrow{tr} s$ and $q_0 \xRightarrow{\alpha} q$ are the transitions involved to generate $(s_0, q_0) \xRightarrow{sq} (s, q)$. So, when projecting sq to alphabet of Σ_1 and Σ_2, the order should be preserved, that is,

$$sq\restriction \Sigma_1 = \pi_2(tr)\restriction \Sigma_1 \text{ and } sq\restriction \Sigma_2 = \mathcal{R}(\alpha)\restriction \Sigma_2.$$

– Direction " $\Longleftarrow$ " :

There exists $s_0 \xRightarrow{tr} s$ and $q_0 \xRightarrow{\alpha} q$ with $q \in F$.

Let $syn = \langle a_0, \dots, a_k\rangle$ be $ptraces(tr) \restriction P$ and $\alpha \restriction shared$. So there exists states $s_1, s_2, \dots, s_k$ of C and $q_1, q_2, \dots, q_k$ of M such that $s_i \xrightarrow{a_i/T_i} ()$ and $q_i \xrightarrow{a_i} ()$ for each $1 \le i \le k$. By Definition 3.4.1, there exists $(s_i, q_i) \xrightarrow{T_i} ()$, if (s_i, q_i) is reachable.

Let $\langle b_1, b_2, \dots, b_m\rangle$ be $sq \restriction \Sigma_2 = \mathcal{R}(\alpha) \restriction \Sigma_2$. So there exists states $q'_1, q'_2, \dots, q'_m$ of M such that $q'_i \xrightarrow{b_i} ()$, for each $0 \le i \le m$. By Definition 3.4.1, there exists $(s', q'_i) \xrightarrow{\{b_i\}} ()$ for each $1 \le i \le m$ and any $s' \in S$, if (s', q'_i) is reachable.

Let $\langle c_1, c_2, \dots, c_n\rangle$ be $\pi_1(tr) \restriction A$. Then, there should be states $s'_1, s'_2, \dots, s'_n$ of C such that $s'_i \xrightarrow{c_i/T'_i} ()$ of C. By Definition 3.4.1, there exists $(s'_i, q') \xrightarrow{T'_i} ()$ for each $1 \le i \le n$ and any state $q' \in Q$, if (s'_i, q') is reachable.

From above, it follows that $(s_0, q_0) \xRightarrow{sq} (s, q)$. This shows $sq \in \mathcal{L}(C \lhd_{shared} M)$.

Thus, the lemma is proved. □

Example 3.4.2. *Consider the requirement of provided service ring in Alarm component. It can be represented as M_{ring} in Figure 3.14. The component automata of Blink is C_{Blink}. The event set shared is $\{BlinkOn\}$. The alphabet set of composition is $\{\{\epsilon\}, \{SoundOn\}\}$. The initial state is $(a, 0)$. $BlinkOn \in$* non-blockable(0)*, and $events(\{\epsilon\}) \cap shared =$. So, there is $(0, a) \xrightarrow{\{\epsilon\}} (1, b)$. At state $(1, b)$, $SoundOn \notin shared$, so, there is $(1, b) \xrightarrow{\{SoundOn\}} (2, b)$ and $(1, b)$ is an accepting state. The language of $C \lhd_{BlinkOn} M_{ring}$ is $\{\langle\{\epsilon\}, \{SoundOn\}\rangle\}$. The concatenation of elements of sequence from the language represents the new requirement for provided service ring.*

3.4.2 Product of Component Automata

We compose two components if their services are disjoint, except a provided service of one may coincide with required service of the other.

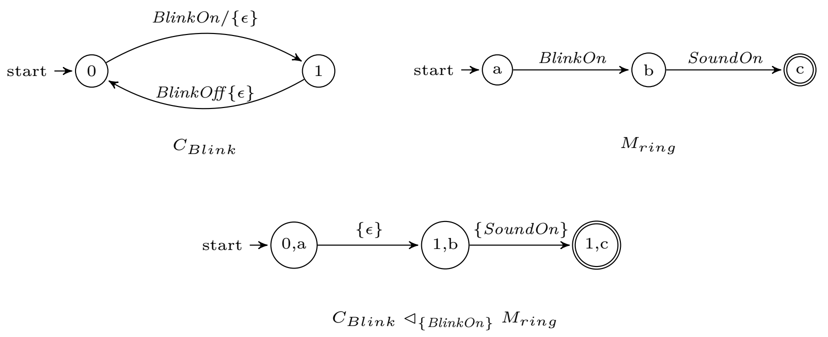


Figure 3.14: Composition of C_{Blink} and M_{ring}

Definition 3.4.2 (Composable). *Two component automata C_1 and C_2 are* composable*, if*

– $(P_1 \cup R_1 \cup A_1) \cap A_2 = \emptyset$

– $(P_2 \cup R_2 \cup A_2) \cap A_1 = \emptyset$

– $P_1 \cap P_2 = \emptyset$

– $R_1 \cap R_2 = \emptyset$

We let shared$(C_1, C_2) = (P_1 \cap R_2) \cup (P_2 \cap R_1)$.

Given component automaton C, we use $C(s)$ to denote the same automaton except the initial state is s, i.e., $(S, s, f, P, R, A, \delta)$ where $s \in S$.

Given two composable components, if the service required by one and provided by the other, this service should be hidden in the composition. The internal events of these components are still internal in the composition. The shared would not be considered as internal events, because the synchronization will happen inside the scope of service body where the required event is. For example, $s \xrightarrow{w/\{a\}} s'$ and $r \xrightarrow{a/T} r'$ will be composed to $(s, r) \xrightarrow{w/T} (s', r')$. Event a happens internally in service body of w. If w is an internal event of a component, it means w can happen in the component and obviously a cannot happen at any state actively.

Now, we define composition operation, called *product*, of component automata. Two component automata synchronize on events in *shared*. Consider component automata C_1 and C_2, if requirement T of provided/internal event w of C_1 contains shared events, in the product, requirement will be updated based on the internal product between $\mathcal{M}(T)$ and component automaton C_2. If $C_2 \not\models \mathcal{M}(T)$, the product will go to f state when w is invoked, because it means the services provided by C_2 is not enabled In the composition, we still use f as error state for any component automaton for simplicity.

Definition 3.4.3 (Product). *Given two composable component automata C_1 and C_2, product $C_1 \otimes C_2$ is defined as $(S, s_0, f, P, R, A, \delta)$, and shared $= (P_1 \cap R_2) \cup (P_2 \cap R_1)$ where*

- *$S = S_1 \times S_2$, $s_0 = (s_0^1, s_0^2)$, and we still use f to denote the error state;*
- *$P = (P_1 \cup P_2) \setminus shared$;*
- *$R = (R_1 \cup R_2) \setminus shared$;*
- *$A = A_1 \cup A_2$;*
- *δ is defined as follows, for any reachable state $(s_1, s_2) \in S$:*
  - *If there is $s_1 \xrightarrow{w/T} s_1'$ of C_1, where $f \notin intR(s_1')$,*
    - *if $C \not\models_{shared} \mathcal{M}(T)$, then $(s_1, s_2) \xrightarrow{w/\{\epsilon\}} f$ is added into δ;*
    - *otherwise, the following set of transitions are added in δ:*

$$\{(s_1, s_2) \xrightarrow{w/T'} (s_1', s_2') \mid \\ T' = \{conc(\ell) \mid q_0 \overset{\ell}{\Longrightarrow} (s_2', t) \text{ in } M, (s_2', t) \in F\}\}$$

where $C_2(s_2) \lhd_{shared} \mathcal{M}(T)$ is (Q, R, q_0, F, σ).

  - *Symmetrically, if there is $s_2 \xrightarrow{w/T} s_2'$ of C_2, transitions are added to δ similarly to the above.*

Notes: $conc(\ell)$ is the regular set obtained by concatenating all the elements of the sequence ℓ.

Example 3.4.3. *We build $C_{Alarm} \otimes C_{Blink}$. The automata we mention in this example are graphically shown in Figure 3.15*

In the product,

$$\begin{aligned}&shared = \{BlinkOn, BlinkOff\},\\ &P = \{set, change, cancel, ring, stop\}, \text{ and}\\ &R = \{Tstart, Tstop, Tadd, Tremove, SoundOn, SoundOff\}.\end{aligned}$$

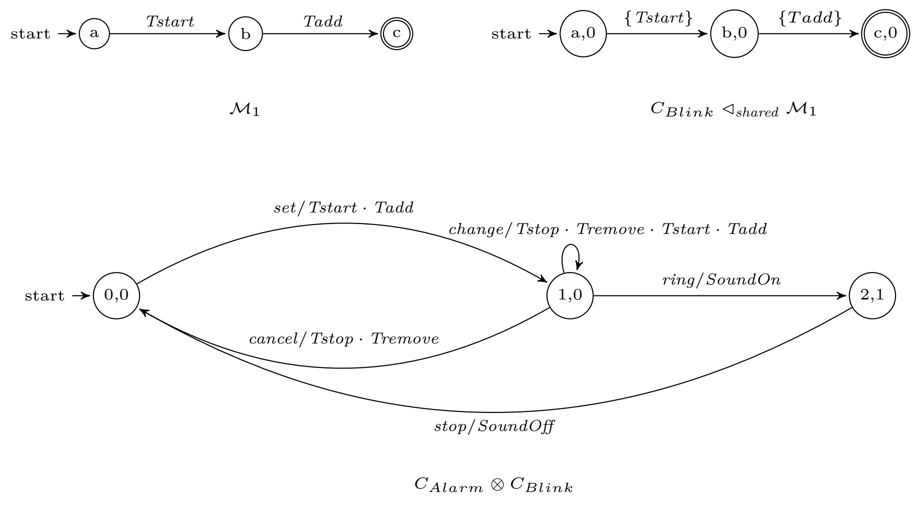


Figure 3.15: Composition of C_{Alarm} and C_{Blink}

Here, we show how transitions are added by presenting two cases here. The others are similar. The initial state is $(0,0)$*, we get* $(0,0) \xrightarrow{set/\,Tstart\cdot Tstop} (1,0)$*, from* $C_{Blink} \lhd_{shared} \mathcal{M}_1$ *where* $\mathcal{M}_1$ *is the finite state machine accepting* $Tstart \cdot Tstop$*. At state* $(1,0)$*, from* $C_{Blink} \lhd_{shared} M_{ring}$*, there is transition* $(1,0) \xrightarrow{ring/SoundOn} (2,1)$*.*

Product operator is defined based on the internal product of a finite state machine for the requirement and a component automaton that provides services. From Lemma 3.4.1, we know the conditions of $C \models_{shared} M$ by trace inclusions. Here, we present a lemma stating a solution for calculating new requirement in the composition.

Lemma 3.4.3. *In the definition of* $C_1 \otimes C_2$*, given a reachable state* (s_1, s_2) *and* $s_1 \xrightarrow{w/T} s'_1$ *such that* $C(s_2) \models_{shared} \mathcal{M}(T)$*, for any set* SQ *of sequences of regular set over* R*, there exists* $(s_1, s_2) \xrightarrow{w/conc(SQ)} (s'_1, s'_2)$*, if and only if for each* $sq \in SQ$ *there exists* $s_2 \overset{tr}{\Longrightarrow} s'_2$ *of* C_2 *and* $\alpha \in T$ *such that* $\alpha \upharpoonright shared \in ptraces(tr)$ *and*

$$sq \upharpoonright \Sigma_1 = \pi_2(tr) \upharpoonright \Sigma_1, \text{ and } sq \upharpoonright \Sigma_2 = \mathcal{R}(\alpha) \upharpoonright \Sigma_2$$

where Σ_1 *is a set of regular sets over* R_2 *and* Σ_2 *is a set of singleton* $\langle a \rangle$ *where* $a \in R_1 \setminus shared$*.* $conc(SQ) = \{conc(sq) \mid sq \in SQ\}$*.*

Proof. This lemma follows directly from Lemma 3.4.2 and Definition 3.4.3. □

From this lemma, the new requirement of transitions added in Definition 3.4.3 can be written

$$\begin{aligned} T' = \{conc(sq) \mid \exists tr \in traces(C(s_2)), \alpha \in T \bullet \\ \alpha \upharpoonright\ shared = ptraces(tr) \wedge \\ sq \upharpoonright \Sigma_1 = \pi_2(tr) \upharpoonright \Sigma_1 \wedge \\ sq \upharpoonright \Sigma_2 = \mathcal{R}(\alpha) \upharpoonright \Sigma_2\} \end{aligned}$$

3.4.3 Plugging

Product is the general composition for any composable components and it is complicated because of the service invocation may happen from both sides. In practice, however, software engineers usually reduce the dependency of components on the environment by plugging closed components into open components. This only requires to check whether services provided by the component can satisfy the requirement declared in the required interface of open components.

We say component C_2 is *pluggable* to C_1, in the composition of C_1 and C_2, component C_2 only provide services to C_1 without requiring any service from C_2.

Definition 3.4.4 (Pluggable). *Given two component automata C_1 and C_2, C_2 is pluggable to C_1, if they are composable, and $P_2 \subseteq R_1$.*

We see that component automata C_{Blink} and C_{Sound} of Blink and Sound components are pluggable to C_{Alarm} of Alarm component. Next, the plugging operation is defined as product of pluggable component automata.

Definition 3.4.5 (Plugging). *Given two automata C_1 and C_2, if C_2 is pluggable to C_1, then the plugging $C_1 \ll C_2$ is $C_1 \otimes C_2$.*

In plugging, $shared = P_2$ $P = P_1$, $R = (R_1 \cup R_2) \setminus P_2$, and $A = A_1 \cup A_2$.

Actually, the example $C_{Alarm} \otimes C_{Blink}$ is also C, and can be written $C_{Alarm} \ll C_{Sound}$.

The following theorem states that the conditions that a provided trace is preserved in the plugging, and the traces of plugging can be directly calculated from traces of the two component automata.

Theorem 3.4.1. *Consider two automata C_1 and C_2 that C_2 is pluggable to C_1. Given a sequence pt of P.*

1. *$pt \in ptraces(C_1 \ll C_2)$, if and only if $pt \in ptraces(C_1)$ and $rtraces_1(pt) \upharpoonright P_2 \subseteq uptraces(C_2)$.*

2. if pt is provided traces of both C_1 and $C_1 \ll C_2$, the relation between $rtraces_1(pt)$ and $rtraces(pt)$ is as follows:

$$\begin{aligned} rtraces(pt) = \{conc(sq) \mid & \exists tr \in traces(C_2), \alpha \in rtraces_1(pt) \bullet \\ & \alpha \upharpoonright P_2 = ptraces(tr) \wedge \\ & sq \upharpoonright \Sigma_1 = \pi_2(tr) \upharpoonright \Sigma_1 \wedge \\ & sq \upharpoonright R_2 = \mathcal{R}(\alpha) \upharpoonright R_2\} \end{aligned}$$

where Σ_1 is set of regular sets of R_1.

Proof. The proof is:

1. This is proved from Lemma 3.4.1.

   Direction $\Longrightarrow$: $pt \in ptraces(C_1 \ll C_2)$, then there must be $pt \in ptraces(C_1)$ and during every step transition $C_2 \models_{P_2} M$ where M is the requirement of element of pt. By Lemma 3.4.1, $traces_1(pt) \upharpoonright P_2 \subseteq uptraces(C_2)$.

   Direction $\Longleftarrow$: because $R_2 \cap P_1 =$, it can be similarly deduced from Lemma 3.4.1.

2. This is proved from Lemma 3.4.3 by induction on the length of transitions.

   The base case directly follows from Lemma 3.4.3. Consider $(s_0^1, s_0^2) \overset{tr}{\Longrightarrow} (s_1^1, s_2^1)$ and $(s_1^1, s_2^1) \xrightarrow{w/T'} (s_2^1, s_2^2)$. There is $s_1^1 \xrightarrow{w/T} s_1^2$. Let pt_1 be $ptraces(tr)$. By induction hypothesis, there is:

$$\begin{aligned} rtraces(pt) = \{conc(sq) \mid & \exists tr \in traces(C_2), \alpha \in rtraces_1(pt) \bullet \\ & \alpha \upharpoonright P_2 = ptraces(tr) \wedge \\ & sq \upharpoonright \Sigma_1 = \pi_2(tr) \upharpoonright \Sigma_1 \wedge \\ & sq \upharpoonright R_2 = \mathcal{R}(\alpha) \upharpoonright R_2\} \end{aligned}$$

   By Lemma 3.4.3, there is:

$$\begin{aligned} T' = \{conc(sq) \mid & \exists tr \in traces(C_2(s_1^2)), \alpha \in T \bullet \\ & \alpha \upharpoonright P_2 = ptraces(tr) \wedge \\ & sq \upharpoonright \Sigma_1 = \pi_2(tr) \upharpoonright \Sigma_1 \wedge \\ & sq \upharpoonright R_2 = \mathcal{R}(\alpha) \upharpoonright R_2\} \end{aligned}$$

   So,

$$\begin{aligned} & rtraces(pt \cdot (w \upharpoonright P)) = rtraces(pt) \cdot T' = \\ & \qquad \{conc(sq) \mid \exists tr \in traces(C_2), \alpha \in rtraces_1(pt) \cdot T \bullet \\ & \qquad \alpha \upharpoonright P_2 = ptraces(tr) \wedge \\ & \qquad sq \upharpoonright \Sigma_1 = \pi_2(tr) \upharpoonright \Sigma_1 \wedge \\ & \qquad sq \upharpoonright R_2 = \mathcal{R}(\alpha) \upharpoonright R_2\} \end{aligned}$$

□

The following corollary directly follows as a special case of Theorem 3.4.1 that C_2 is a closed component automaton. When all required services are provided non-blockably, the plugging can be easily obtained. Note that states in component automata are only symbolic notions to be used to control the order of service invocation. So, in $C_1 \ll C_2$ when R_2 is $\emptyset$ and C_2 provides all the services required by C_1 on P_2 non-blockably, we keep the notions of states of C_1 in the plugging.

Corollary 3.4.1. *Given two automata C_1 and C_2 such that C_2 is pluggable to C_1 and $R_2 = \emptyset$:*

– *given any pt, the relation between rtraces(pt) and rtraces$_1$(pt) is: $rtraces(pt) = \{sq \upharpoonright R \mid sq \in rtraces_1(pt)\}$.*

– *if $rtraces(C_1) \upharpoonright P_2 \subseteq uptraces(C_2)$, then $C_1 \ll C_2 = (S, s_0, f, P_1, R_1 \setminus P_2, A_1 \cup A_2, \delta'_1)$ where δ'_1 is obtained from δ_1 by replacing elements in R_2 with ϵ.*

In the example of $C_{Alarm} \ll C_{Blink}$, we see $rtraces(C_{Alarm}) \upharpoonright P_{Blink} = (BlinkOn \cdot BlinkOff)^* + (BlinkOn \cdot BlinkOff)^* \cdot BlinkOn$. And the required traces are obviously non-blockablein C_{Blink}. So, all the provided traces of C_{Alarm} are preserved in the plugging. From the corollary, the plugging $C_{Alarm} \ll C_{Blink} \ll C_{Sound}$ (shown in Figure 3.16) can be directly obtained by removing the *BlinkOn*, *BlinkOff*, and *SoundOn*, *SoundOff*.

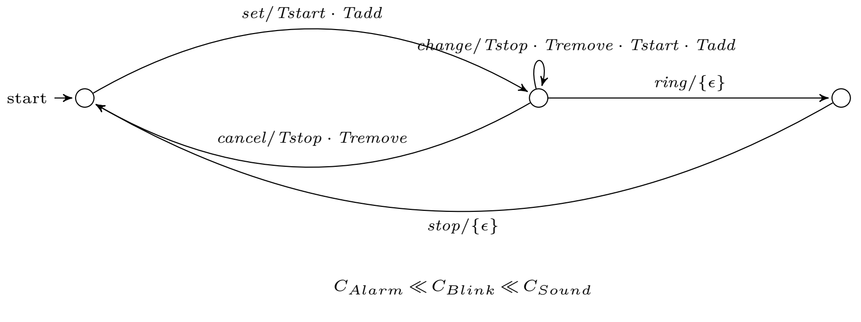

set/ Tstart · Tadd
change/ Tstop · Tremove · Tstart · Tadd
start
ring/ {ε}
cancel/ Tstop · Tremove
stop/ {ε}
$C_{Alarm} \ll C_{Blink} \ll C_{Sound}$

Figure 3.16: Composition of C_{Alarm} and C_{Blink} and C_{Sound}

3.4.4 Composition of Component Interface Automata

The product operation is not closed for component interface automata, that is, the product of two component interface automata may contain blockable traces. This is because of what we call output-non-determinism of components. Components may decide which required trace to choose when providing services. This is sometimes a design error for some components. For example, in a bad designed pay component, it provides *pay* and requires *cash* or *card*. The transitions step will consist transition $(pay, \{cash, card\})$. In this case, to guarantee safe call of *pay*, both *cash* and *card* are required. A "better" design should let the clients to choose the way to pay. Then, it provides *cashpay* and *cardpay* which require *cash* and *card*, respectively.

Here, I use one personal experience as an example to show the idea how new blockable behavior arises in the product of interfaces.

Example 3.4.4. *There are two component interface automata I_1 and I_2, shown in Fig. 3.17. Interface I_1 provides a smart-connect service which can choose either* 3*G or wifi based on which one is faster at run-time. However, there is a internet phone call app, of which the interface I_2 shows that service call is only available when wifi is connected, due to a configuration which aims to use* 3*G as less as possible. So, in product $I_1 \otimes I_2$, $\langle smartCon, call\rangle$ is blockable.*

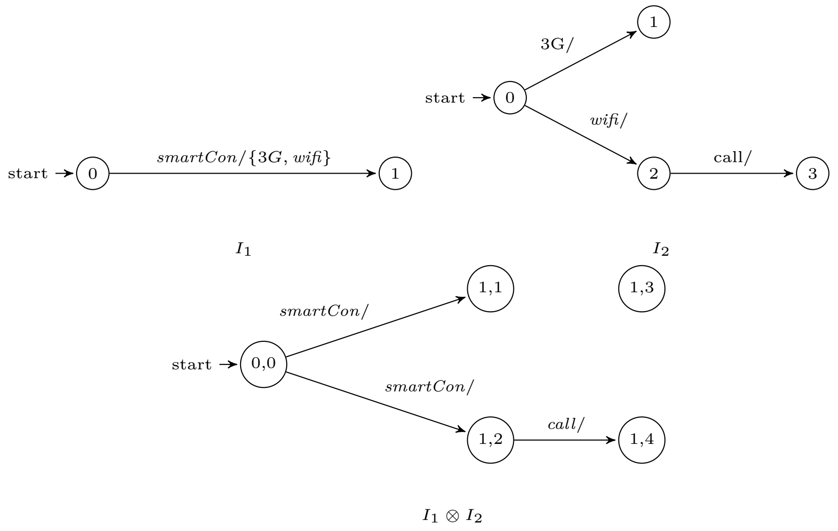


Figure 3.17: Product of two component interface automata

So we introduce a new composition operator of component interface automata based on product and Algorithm 2.

Definition 3.4.6 (Composition). *Given two composable component interface automata I_1 and I_2, the composition $I_1 \parallel I_2$ is $\mathcal{I}(I_1 \otimes I_2)$.*

In Figure 3.17, so $I_1 \parallel I_2$ should be $I_1 \otimes I_2$ by removing the *call* transition. This makes the interface very "limited", but this is still very useful and important to let the clients be aware of this rather than to face failure in making call during run time.

The composition seems restrictive, however, we believe that any services that may be blocked during run time should not be presented to the clients in the interface. The interface property that all behaviors specified in the interfaces should be non-blockable is important in building reliable software system.

3.4.5 Hiding and Renaming

In this part, we define*hiding* and *renaming* for flexible reuse of components. The meaning is intuitive from this composition.

We think the compositions are intuitive enough that we will only give the definitions without going to details about properties, such as traces, blockableness, and interface properties of these two compositions.

Definition 3.4.7 (Hiding). *Given a component automaton C and a set of provided events $E \subseteq P$, hiding E from C is $C\backslash E = (S, s_0, f, P \setminus E, R, A \cup E, \delta')$ where δ' is obtained from δ by removing transitions of E*

Given a alphabet set P' with fresh variables different from P. $f : P \to P'$ is a partial transformation, which renames $p \in P$ as $f[p]$ in P'.

Definition 3.4.8 (Renaming). *Given a component automaton C and a partial injective function $f : P \cup R \cup A \to P' \cup R' \cup A'$ where all P, R, A, P', R', A' are disjoint with each other. Renaming C by f is $f[C] = (S, s_0, f, f[P], f[R], f[A], f[\delta])$, where $f[\delta]$ is the transition set by renaming corresponding label names of each transition by f in δ.*

3.5 Refinement

Then we study refinement relation between component automata. Refinement is one of the key issues in component based development. It is mainly for substitution at interface level. We define a refinement relation by state simulation technique [Mil95]. The intuitive idea is state s' simulates s, if at state s' more provided events are non-blockable, less required traces are required and the next states following the transitions keep the simulation relation, which is similar to alternating simulation in [DH01].

Let C be a component automaton, some non-blockable provided events at a given state s may also be refused. For example, provided event c at state 3 may be refused shown in Fig. 3.9. which is caused by non-determinism after provided event a is called. Therefore, the refusal set at s, i.e., *refusals*(s) in Definition 3.3.9, is a subset of non-blockable(s). Therefore, the refusal set by our definition is a little different from the one in CSP.

Below, we define a strong simulation relation between states. Refinement relation between component automata then will be defined based on the simulation relation between its initial states. So, in the following definition, we will not claim whether these two states and transitions are in the same or different automata. We recall that *refusals*$(s) = \text{non-blockable}(s) \setminus \bigcup_{(Q,s)\in S_I} \text{non-blockable}(Q,s)$. That is also

$$\bigcap_{(Q,s)\in S_I} (\text{non-blockable}(s) \setminus \text{non-blockable}(Q,s)).$$

And we get

$$\text{non-blockable}(s) \setminus \textit{refusals}(s) = \bigcup_{(Q,s)\in S_I} \text{non-blockable}(Q,s)$$

Definition 3.5.1 (Strong simulation). *A binary relation R over the set of states of component automata is a simulation if and only if whenever s_1Rs_2:*

1. *refusals*$(s_2) \subseteq$ *refusals*(s_1);
2. *if $s_2 \xrightarrow{w/} f$ with $w \in A \cup P$, then $s_1 \xrightarrow{w/} f$;*
3. *if $s_1 \xrightarrow{w/T} s_1'$ with $w \in A \cup \text{non-blockable}(s_1) \setminus \textit{refusals}(s_1)$, there exists s_2' and T' such that $s_2 \xrightarrow{w/T'} s_2'$ where $T' \subseteq T$ and $s_1'Rs_2'$;*
4. *for any transitions $s_2 \xrightarrow{w/T'} s_2'$ with $w \in A \cup \text{non-blockable}(s_1) \setminus \textit{refusals}(s_1)$, then there exists s_1' and T such that $s_1 \xrightarrow{w/T} s_1'$ where $T' \subseteq T$ and $s_1'Rs_2'$;*

We say that s_2 simulates s_1, written $s_1 \lesssim s_2$, if $(s_1, s_2) \in R$. C_2 refines *C_1, denoted by $C_1 \sqsubseteq_{alt} C_2$, if there exists a simulation relation R such that $s_0^1Rs_0^2$, where* **div** *or f are not internally reachable from s_0^1 and s_0^2, and $P_1 = P_2$, $R_2 = R_1$, and $A_1 = A_2$.*

First, *refusals*$(s_2) \subseteq$ *refusals*(s_1) is used to assure that s_2 is less likely to allow events that are in blockable traces. Second, we would like to remove the case $s_2 \xrightarrow{a/} f$ and $s_1 \xrightarrow{a/} s_1'$, that means s_2 is better at avoiding the error state. Third, it implies for any non-blockable and non-refusal provided transitions, s_2 can always do same as s_1, that shows s_2 can provide as much as s_1. Fourth, for all transitions that are non-blockable and non-refusal in s_1, the requirement at state s_2 is simulated by s_1, that is, s_1 always requires as much as s_2. By these, we would expect state s_2 can provide as much as s_1 while requiring less, and s_2 is better at avoiding error and blockable transitions.

We get the following proposition.

Proposition 3.5.1. *Given a simulation relation R, for each $(s_1, s_2) \in R$.*

1. *state s_1 is stable, if and only if s_2 is stable.*
2. *s_1 is divergent, if and only if s_2 is divergent.*
3. *if $a \in \text{non-blockable}(s_1) \setminus \mathit{refusals}(s_1)$, then $a \in \text{non-blockable}(s_2)$. That is $\text{non-blockable}(s_1) \setminus \mathit{refusals}(s_1) \subseteq \text{non-blockable}(s_2) \setminus \mathit{refusals}(s_2)$, since $\mathit{refusals}(s_2) \subseteq \mathit{refusals}(s_1)$.*

Proof. The proof is:

1. In the definition, when $w \in A$, it implies that if one has internal transition, the other also has internal transition.
2. because they simulate each other on internal transitions.
3. if $a \notin \text{non-blockable}(s_2)$, there exists $\mathit{transs_2trs'_2}$ with $\mathit{ptraces}(tr) = a$ such that either s'_2 is divergent or error state, or that $a \notin \mathit{out}^P(s'_2)$ where s_2 is stable. From both cases, we can deduce that $a \notin \text{non-blockable}(s_1)$. By contradiction, it is proved.

□

Next, we show that refinement relation is reflexive and transitive.

Proposition 3.5.2. *Refinement is reflexive and transitive.*

Proof. Reflexivity: for any component automaton C, $R = \{(s, s) \mid s \in S\}$ is a simulation relations with $(s_0, s_0) \in R$. So $C \sqsubseteq_{alt} C$.

Transitivity is that, for any C_1, C_2, and C_3 such that $C_1 \sqsubseteq_{alt} C_2$ and $C_2 \sqsubseteq_{alt} C_3$, then $C_1 \sqsubseteq_{alt} C_2$.

Let R_1 and R_2 be the strong simulation for the refinement $C_1 \sqsubseteq_{alt} C_2$ and $C_2 \sqsubseteq_{alt} C_3$, respectively. We would prove that $R = \{(s_1, s_3) \mid \exists s \bullet (s_1, s) \in R_1, (s, s_3) \in R\}$ is a strong simulation relation. For any $(s_1, s_3) \in R$, there exists s such that $(s_1, s) \in R_1$ and $(s, s_3) \in R_2$,

1. $\mathit{refusals}(s) \subseteq \mathit{refusals}(s_1)$ and $\mathit{refusals}(s_3) \subseteq \mathit{refusals}(s)$. So, $\mathit{refusals}(s_3) \subseteq \mathit{refusals}(s_1)$.

2. if $s_3 \xrightarrow{w/} f$, then $s \xrightarrow{w/} f$ which implies $s_1 \xrightarrow{w/} f$.

3. if $s_1 \xrightarrow{w/T_1} s_1'$ with $w \in A \cup \text{non-blockable}(s_1) \setminus \mathit{refusals}(s_1)$, there exists s' and T such that $s \xrightarrow{w/T} s'$ and $(s_1', s') \in R_1$. From last proposition, we know $w \in A \cup \text{non-blockable}(s) \setminus \mathit{refusals}(s)$. Thus, there exists s_3' and T_3 such that $s_3 \xrightarrow{w/T_3} s_3'$, $T_3 \subseteq T$ and $(s', s_3') \in R_2$. So $(s_1', s_3') \in R$.

4. for any $s_3 \xrightarrow{w/T_3} s_3'$ with $w \in A \cup \text{non-blockable}(s_1) \setminus \mathit{refusals}(s_1)$, because $\text{non-blockable}(s_1) \setminus \mathit{refusals}(s_1) \subseteq \text{non-blockable}(s) \setminus \mathit{refusals}(s)$, there exists s' and T that $s \xrightarrow{w/T} s'$, $T_3 \subseteq T$, and $(s', s_3') \in R_2$. Then there also exits s_1' and T_1 such that $s_1 \xrightarrow{w/T_1} s_1'$, $T \subseteq T_1$, and $(s_1', s') \in R_1$. Thus, $(s_1', s_3') \in R$.

□

We see that automaton of $C_{badBlink}$ in Figure 3.6 is refined by C_{Blink} in Figure 3.5. The simulation set is $\{(0,0), (1,1), (2,1)\}$. In $C_{badBlink}$, $\mathit{refusals}(1) = \{\mathit{BlinkOff}\}$. In C_{Blink}, $\mathit{refusals}(1) = \emptyset$.

Simulation in CCS [Mil95] is used to differentiate non-determinism which traces cannot do. For example in the following example (shown in Figure 3.18), s_0 strongly simulates r_0 and the strong simulation relation set is $\{(s_0, r_0), (s_1, r_1), (s_2, r_2)\}$. But r_0 can't simulate s_0. In the view of providing non-blockable traces, we think the one with r_0 as initial state is better. We constrain the simulation on refusal set. Then in our definition, r_0 strongly simulates s_0, that is, $s_0 \lesssim r_0$ with the simulation relation set $\{(s_0, r_0), (s_1, r_1), (s_3, r_1)\}$. $\text{non-blockable}(s_1) \setminus \mathit{refusals}(s_1) = \emptyset$, because $\mathit{refusals}(s_1) = \{b\}$ and $\text{non-blockable}(s_1) = \{b\}$. $\text{non-blockable}(r_1) \setminus \mathit{refusals}(r_1) = \{b\}$, because $\text{non-blockable}(r_1) = \{b\}$ and $\mathit{refusals}(r_1) = \emptyset$.

The above definition is quite similar to the alternating simulation given in [DH01]. But they are different, and main differences include: first of all, in our definition, we only require a pair of states keep the simulation relation w.r.t. the provided services that could not result in deadlock; in addition, we also require a refinement should

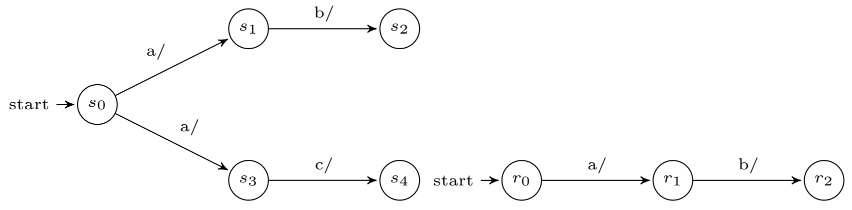


Figure 3.18:

have smaller refusal sets at each location, which is similar to the stable failures model of CSP. Also notice that our refinement is not comparable with the failure refinement nor the failure-divergence refinement of CSP, because of the different requirements on the simulation of provided methods and required methods. However, if we suppose no required methods, our definition is stronger than the failure refinement as well as failure-divergence refinement as we explained above.

The simulation is altering on provided/internal events and requirement. That means, when $s_1 \lesssim s_2$, s_2 can provide as much as s_1 but require less for the simulated provided/internal events. So, in the following example of Figure 3.19, there exists $r'_0 \lesssim s'_0$

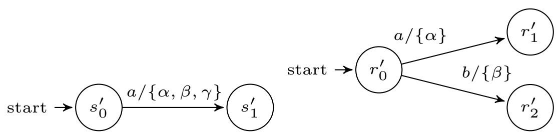


Figure 3.19:

The following theorem indicates the component interface automaton constructed by Algorithm 2 is a refinement of the considered component automaton w.r.t. the above definition, which justifies that we can safely use the resulted component interface instead of the component at the interface level. The theorem also states that the Algorithm 2 preserves the refinement relation. The proof is mainly to find a simulation relation.

Theorem 3.5.1. *The following two properties hold for the refinement relation:*

1. *Given a component automaton C, then $C \sqsubseteq_{alt} \mathcal{I}(C)$.*
2. *Given two component automata C_1 and C_2 such that $C_1 \sqsubseteq_{alt} C_2$, then*

$$\mathcal{I}(C_1) \sqsubseteq_{alt} \mathcal{I}(C_2)$$

.

Proof. We prove the first part by $R = \{(s, (Q, s)) \mid s \in S, (Q, s) \in S_I\}$ and show R is a simulation relation.

For any $sR(Q, s)$,

1. $\mathit{refusals}(Q, s) \subseteq \mathit{refusals}(s)$, because $\mathit{refusals}(Q, s) = \emptyset$, and f is not reachable in (Q, s).
2. If $s \xrightarrow{a/T} s'$ with $a \in$ non-blockable$(s)\backslash\mathit{refusals}(s)$, From definition of $\mathit{refusals}(s)$, there $a \in \bigcup_{(Q',s)\in S_I}$ non-blockable(Q', s). So $a \in$ non-blockable(Q, s) that implies there exists Q' such that $(Q, s) \xrightarrow{a/T} (Q', s')$. Obviously, $s'R(Q', s')$.

   If $s \xrightarrow{;w/T} s'$ with $; w \in A$, then $(Q, s) \xrightarrow{;w/T} (Q, s')$. Obviously, $s'R(Q, s')$,
3. for any $(Q, s) \xrightarrow{e/T} (Q', s')$ with $e \in A \cup$non-blockable$(s) \setminus \mathit{refusals}(Q, s)$. Then $s \xrightarrow{e/T} s'$ and $s'R(Q', s')$.

From above, we see R is a simulation relation. So $C \sqsubseteq_{alt} \mathcal{I}(C)$, because $s_0R(\{s_0\}, s_0)$.

Now we prove the second part of the theorem. Let R be a simulation relation for $C_1 \sqsubseteq_{alt} C_2$ and $(s^1_0, s^2_0) \in R$.

We consider relation R' and it is the smallest set constructed by the following rules:

1. $((Q^1_0, s^1_0), (Q^2_0, s^2_0)) \in R'$,
2. $((Q_1, s_1), (Q_2, s_2)) \in R'$ and $(s'_1, s'_2) \in R'$,

$$\frac{s_1 \xrightarrow{w/T} s'_1 \quad s_2 \xrightarrow{w/T'} s'_2 \quad T' \subseteq T}{((Q_1, s'_1), (Q_2, s'_2)) \in R'} \; w \in A$$

$$\frac{s_1 \xrightarrow{a/T} s'_1 \quad s_2 \xrightarrow{a/T'} s'_2 \quad T' \subseteq T}{(\delta_1(Q_1, a), s'_1), (\delta_2(Q_2, a), s'_2) \in R'} \; a \in \text{non-blockable}(s_1) \setminus \mathit{refusals}(s_1)$$

   where $\delta(Q, a) = \{s' \mid \exists s \in Q, T \bullet s \overset{a}{\Longrightarrow} s'\}$

We need to prove R is a simulation relation and $((Q^1_0, s^1_0), (Q^2_0, s^2_0)) \in R'$, where $(Q^{;}_0 s^1_0)$ and (Q^2_0, s^2_0) are the initial states of $\mathcal{I}(C_1)$ and $\mathcal{I}(C_2)$.

First, it can be proved that if $((Q_1, s_1), (Q_2, s_2)) \in R'$ then $(s_1, s_2) \in R$ by the structural induction. For any $((Q_1, s_1), (Q_2, s_2)) \in R'$,

1. $\mathit{refusals}(Q_1, s_1) = \mathit{refusals}(Q_2, s_2) = \emptyset$, because *refusals* in interface automata are empty. f is not reachable.

2. we prove the next two cases:

   – if $(Q_1, s_1) \xrightarrow{w/T} (Q_1', s_1')$ with $w \in A$, then $Q_1' = Q_1$ and $s_1 \xrightarrow{w/T} s_1'$. Because $(s_1, s_2) \in R$, so there exists s_2' and $T' \subseteq T$ such that $s_2 \xrightarrow{w/T'} s_2'$. So, $(Q_2, s_2) \xrightarrow{w/T'} (Q_2, s_2')$. From the definition, we see $((Q_1, s_1'), (Q_2, s_2')) \in R'$

   – if $(Q_1, s_1) \xrightarrow{a/T} (Q_1', s_1')$ with $a \in \text{non-blockable}(Q_1, s_1) \setminus \mathit{refusals}(Q_1, s_1)$, then $s_1 \xrightarrow{a/T} s_1'$ and $Q_1' = \delta(Q_1, a)$ and $a \in \text{non-blockable}(s_1)$. Because, $\text{non-blockable}(s_1) \setminus \mathit{refusals}(s_1) = \bigcup_{(s,Q) \in S_I^1} \text{non-blockable}(s, Q)$, so $a \in \text{non-blockable}(s_1)$. Then because $(s_1, s_2) \in R$ and $a \in \text{non-blockable}(s_1) \setminus \mathit{refusals}(s_1)$, then there is $s_2 \xrightarrow{a/T'} (s_2')$ such that $T' \subseteq T$ and $(s_1', s_2') \in R$. Let Q_2' be $\delta(Q_2, a)$. Then there exists $(Q_2, s_2) \xrightarrow{a/T'} (Q_2', s_2')$. From the definition of R', we see $((Q_1', s_1'), (Q_2', s_2')) \in R'$.

3. we need to prove the two cases:

   – for any $tranQ_2, s_2 w/T' Q_2, s_2'$ with $w \in A$, it implies that $s_2 \xrightarrow{w/T'} s_2'$. Since $(s_1, s_2) \in R$, so there exists T and s_1' such that $T' \subseteq T$, $s_1 \xrightarrow{w/T} s_1'$, and $(s_1', s_2') \in R$. Then there exists $(Q_1, s_1) \xrightarrow{w/T} (Q_1, s_1')$. So, $((Q_1, s_1'), (Q_2, s_2')) \in R'$.

   – for any $(Q_2, s_2) \xrightarrow{a/T'} (Q_2', s_2')$ with $a \in \text{non-blockable}(Q_1, s_1) \setminus \mathit{refusals}(Q_1, s_1)$, then $s_2 \xrightarrow{a/T'} s_2'$, $Q_2' = \delta(Q_2, a)$ and $a \in \text{non-blockable}(s_1)$. Because, $\text{non-blockable}(s_1) \setminus \mathit{refusals}(s_1) = \bigcup_{(s,Q) \in S_I^1} \text{non-blockable}(s, Q)$, so $a \in \text{non-blockable}(s_1)$. Then, because $(s_1, s_2) \in R$ and $a \in \text{non-blockable}(s_1) \setminus \mathit{refusals}(s_1)$, then there is $s_1 \xrightarrow{a/T} (s_1')$ such that $T' \subseteq T$ and $(s_1', s_2') \in R$. Let Q_1' be $\delta(Q_1, a)$. Then there exists $(Q_1, s_1) \xrightarrow{a/T} (Q_1', s_1')$. From the definition of R', we see $((Q_1', s_1'), (Q_2', s_2')) \in R'$.

The above proof shows that R' is a simulation relation. And from definition of R', then $((Q_0^1, s_0^1), (Q_0^2, s_0^2)) \in R'$. So $\mathcal{I}(C_1) \sqsubseteq_{alt} \mathcal{I}(C_2)$. □

Next, we would discuss about the relation of traces in the refinement relation. First, we need define a relation between traces, which is preorder.

Definition 3.5.2. *We define the following relation between traces. $\preceq$ is a smallest set defined by following rules:*

1. $\epsilon \preceq \epsilon$,
2. $\langle (w,T) \rangle \preceq \langle (w,T') \rangle$, *if* $T' \subseteq T$ *and* $w \in A \cup P$,
3. $tr_1 \cdot tr_2 \preceq tr'_1 \cdot tr'_2$, *if* $tr_1 \preceq tr'_1$ *and* $tr_2 \preceq tr'_2$.

It can be proved by structural induction that this relation is preorder, that is, reflexive and transitive.

The following lemma gives extension version of Definition 3.5.1.

Lemma 3.5.1. *For two component automaton* C_1 *and* C_2, *if* $C_1 \sqsubseteq_{alt} C_2$ *with* R *as the simulation relation.*

1. *If* $s_0^1 \overset{tr_1}{\Longrightarrow} s_1$ *with* $ptraces(tr_1) \in uptraces(C_1)$, *then* $ptraces(tr_1) \in uptraces(C_2)$ *and there exists* s_2 *and* tr_2 *such that* $s_0^2 \overset{tr_2}{\Longrightarrow} s_2$, $tr_1 \preceq tr_2$, *and* $(s_1, s_2) \in R$.
2. *If* $s_0^2 \overset{tr_2}{\Longrightarrow} s_2$ *with* $ptraces(tr_2) \in uptraces(C_1)$, *there exists* s_1 *and* tr_1 *such that* $s_0^1 \overset{tr_1}{\Longrightarrow} s_1$, $tr_1 \preceq tr_2$, *and* $(s_1, s_2) \in R$.

Proof. The proof is by induction on the length of transitions. The base case follows directly from Definition 3.5.1.

1. We consider $s_0^1 \overset{tr_1 \cdot (w/T)}{\Longrightarrow} s_1$ with $ptraces(tr_1) \cdot (w \upharpoonright P) \in uptraces(C_1)$. Then there exist $s_0^1 \overset{tr_1}{\Longrightarrow} s'_1$ and $s'_1 \xrightarrow{w/T} s_1$. By induction hypothesis, $ptraces(tr_1) \in uptraces(C_2)$ and there exist s'_2 and tr_2 such that $s_0^2 \overset{tr_2}{\Longrightarrow} s'_2$, $tr_1 \preceq tr_2$, and $(s'_1, s'_2) \in R$. By Propositions 3.5.1 and 3.3.1, $uptraces(tr_1) \cdot w \in uptraces(C_2)$, if $w \in P$. Because $ptraces(tr_1) \cdot w$ is non-blockable, if $w \in P$, then $w \notin refusals(s'_1)$. So by Definition 3.5.1. There exist s_2 and T' such that $s'_2 \xrightarrow{w/T'} s_2$, $T' \subseteq T$, and $(s_1, s_2) \in R$. So $s_0^2 \overset{tr_2 \cdot (w,T')}{\Longrightarrow} s_2$, $tr_1 \cdot (w,T) \preceq tr_2 \cdot (w,T')$, and $(s_1, s_2) \in R$.
2. We consider $s_0^2 \overset{tr_2 \cdot (w/T')}{\Longrightarrow} s_2$ with $ptraces(tr_1) \cdot (w \upharpoonright P) \in uptraces(C_1)$. Then there exist $s_0^2 \overset{tr_2}{\Longrightarrow} s'_2$ and $s'_2 \xrightarrow{w/T'} s_2$. By induction hypothesis, there exist s'_1 and tr_1 such that $s_0^2 \overset{tr_2}{\Longrightarrow} s'_2$, $tr_1 \preceq tr_2$, and $(s'_1, s'_2) \in R$. Because $ptraces(tr_1) \cdot w$ is non-blockable, if $w \in P$, then $w \notin refusals(s'_1)$. So by Definition 3.5.1.

There exist s_1 and T such that $s'_1 \xrightarrow{w/T} s_1$, $T' \subseteq T$, and $(s_1, s_2) \in R$. So $tr_1 \cdot (w, T) \preceq tr_2 \cdot (w, T')$, $s^1_0 \xRightarrow{tr_1 \cdot (w,T)} s_1$, and $(s_1, s_2) \in R$.

□

From the lemma, we prove the following theorem shows that a refined component have more non-blockable provided traces and less required traces.

Theorem 3.5.2. *Given two component automata C_1 and C_2, if $C_1 \sqsubseteq_{alt} C_2$, then*

1. *$uptraces(C_1) \subseteq uptraces(C_2)$,*
2. *for any non-blockable trace $tr_1 \in utraces(C_1)$, then there exists $tr_2 \in utraces(C_2)$ and $tr_1 \preceq tr_2$.*
3. *for any non-blockable trace $tr_2 \in utraces(C_2)$ if $ptraces(tr_2) \in uptraces(C_1)$, then there exists non-blockable $tr_1 \in utraces(C_1)$ such $tr_1 \preceq tr_2$.*

Proof. The theorem naturally follows from Lemma 3.5.1 and Proposition 3.5.1. □

From this theorem, we also see that $uptraces(C_1) \subseteq uptraces(C_2)$ and for any $pt \in uptraces(C_1)$, there is $rtraces_1(C_2) \subseteq rtraces_2(C_1)$.

The following theorem states that the refinement relation of interfaces is preserved by the composition operator over component automata. Composition operator is monotonic with refinement relation, that is, we can always replace a component I_1 with a more refined one I_2 such that $I_1 \sqsubseteq_{alt} I_2$, compositions with the same component automaton preserves the refinement relation that $I_1 \otimes C \sqsubseteq_{alt} I_2 \otimes C$. However, the general version, if $C_1 \sqsubseteq_{alt} C_1$ and $C_2 \sqsubseteq_{alt} C'_2$, then $C_1 \otimes C_2 \sqsubseteq_{alt} C'_1 \otimes C'_2$, does not hold. This is discussed in next section.

Theorem 3.5.3. *Given a component automaton C and two component interface automata I_1 and I_2 such that C is composable with C_1 and C_2, if $I_1 \sqsubseteq_{alt} I_2$, then $I_1 \otimes C \sqsubseteq_{alt} I_2 \otimes C$.*

Proof. Assume the simulation relation of $I_1 \sqsubseteq_{alt} I_2$ is R. We first prove the relation $R' = \{((s_1, s), (s_2, s)) \mid (s_1, s_2) \in R \text{ and } s \in S\}$ has the following property.

Consider $((s_1, r), (s_2, r)) \in R$. By Lemma 3.4.1, (s_2, r) leads to the error state implies (s_1, r) leads to the error states. We prove by considering the following two cases.

First, C_1 and C_2 requires services from C. If $(s_1, r) \xrightarrow{w/T'_1} (s'_1, r')$, we assume that $s_1 \xrightarrow{w/T_1} s'_1$ and $r \xRightarrow{tr} r'$. C_1 and C_2 are non-blockable component automata, so $w \in A \cup \text{non-blockable}(s_1) \setminus \mathit{refusals}(s_1)$. By simulation, there exists $s_2 \xrightarrow{w/T_2} s'_2$ that $T'_2 \subseteq T'_1$. Then $(s_2, r) \xrightarrow{w/T'_2} (s'_2, r)$ that $T'_2 \subseteq T'_1$, by Lemma 3.4.3.

For any $(s_2, r) \xrightarrow{w/T'_2} (s'_2, r')$ with $w \in A \cup \text{non-blockable}(s_2)$ and $w \notin \mathcal{F}(s_1, r)$, then there exists $s_2 \xrightarrow{w/T_2} s'_2$, $r \xRightarrow{tr} r'$. Then $w \in A \cup \text{non-blockable}(s_1)$ and $w \notin \mathcal{F}(s_1)$. By simulation relation, there exists $s_1 \xrightarrow{w/T_1} s'_1$ with $T_2 \subseteq T_1$. Then $(s_1, r) \xrightarrow{(w/T'_1)} (s'_1, r')$ where $T'_2 \subseteq T'_1$ by Lemma 3.4.3.

Second, C require services from C_1 and C_2, respectively. If $(s_1, r) \xrightarrow{w/T'_1} (s'_1, r')$, we assume that $s_1 \xRightarrow{tr_1} s'_1$ and $r \xRightarrow{w/T} r'$. Because C_1 and C_2 are two non-blockable component automata, there exists $s_2 \xRightarrow{tr_2} s'_2$ that $\pi_2(tr_2) \subseteq \pi_2(tr_1)$ and $\pi_1(tr_2)\upharpoonright_{P_2} = \pi_1(tr_1)\upharpoonright_{P_1}$ by simulation. Then $(s_2, r) \xrightarrow{w/T'_2} (s'_2, r')$ that $T'_2 \subseteq T'_1$, by Lemma 3.4.3.

For any $(s_2, r) \xrightarrow{w/T'_2} (s'_2, r')$, then there exists $s_2 \xRightarrow{tr_2} s'_2$ and $r \xrightarrow{w/T} r'$. Because C_1 and C_2 are non-blockable component automata, there exists $s_1 \xRightarrow{tr_1} s'_1$ that $\pi_2(tr_2) \subseteq \pi_2(tr_1)$ and $\pi_1(tr_2)\upharpoonright_{P_2} = \pi_1(tr_1)\upharpoonright_{P_1}$ by simulation. Then $(s_1, r) \xrightarrow{(w/T'_1)} (s'_1, r')$ where $T'_2 \subseteq T'_1$ by Lemma 3.4.3.

$\mathit{refusals}(s_2) \subseteq \mathit{refusals}(s_1)$, then $\mathit{refusals}(s_2, r) \subseteq \mathit{refusals}(s_1, r)$.

For both cases, $s'_1 R s'_2$, so $(s'_1, r')\mathcal{R}'(s'_2, r')$, this implies that $\mathcal{R}'$ is an simulation relation. □

Now, we show that the refinement relation is preserved in the composition of interface automata.

Corollary 3.5.1. *Given two component interface automata I_1, I_2, I'_1, and I'_2 such that $I_1 \sqsubseteq_{alt} I'_1$ and $I_2 \sqsubseteq_{alt} I'_2$, if I_1 is composable with I_2, then I'_1 is composable with I'_2 and $I_1 \parallel I_2 \sqsubseteq_{alt} I'_1 \parallel I'_2$.*

Proof. Because the refinement relation requires they have same provided, required, and internal events. So, I'_1 is composable with I'_2 too. $I_1 \otimes I_2 \sqsubseteq_{alt} I'_1 \otimes I'_2$ from Theorem 3.5.3. By Lemma 3.5.1, $\mathcal{I}(I_1 \otimes I_2) \sqsubseteq_{alt} \mathcal{I}(I'_1 \otimes I'_2)$, that is $I_1 \parallel I_2 \sqsubseteq_{alt} I'_1 \parallel I'_2$. □

Refinement relation is not preserved in the composition of component automata. However, we show that plugging is monotonic with refinement.

Theorem 3.5.4. *Given four component automata C_1, C_1', C_2, and C_2', such that $C_1 \sqsubseteq_{alt} C_1'$ and $C_2 \sqsubseteq_{alt} C_2'$, if C_2 is pluggable to C_1 and C_2' is pluggable to C_1', then*

$$C_1 \ll C_2 \sqsubseteq_{alt} C_1' \ll C_2'.$$

Proof. Let R_1 and R_2 be the strong simulations for $C_1 \sqsubseteq_{alt} C_1'$ and $C_2 \sqsubseteq_{alt} C_2'$. First we prove $(s_1, s_1') \in R_1$ and $(s_2, s_2') \in R_2$. Let $R \subseteq \{((s_1, s_2), (s_1', s_2')) \mid (s_1, s_1') \in R_1, (s_2, s_2') \in R_2\}$ be a set of relations defined by following rules.

1. s_0^1, s_0^2, r_0^1, and r_0^2 are initial state of C_1, C_2, C_1', and C_2', respectively.

$$\frac{}{((s_0^1, s_0^2), (r_0^1, r_0^2)) \in R}$$

2. For $((s_1, s_2), (s_1', s_2')) \in R$,

   – if $w \in A$, $(s_1, s_2) \xrightarrow{w/T} (r_1, r_2)$ and $T' \subseteq T$ such that $(s_1', s_2') \xrightarrow{w/T'} (r_1', r_2')$, $(r_1, r_1') \in R_1$, and $(r_2, r_2') \in R_2$, or

   – if $a \in$ non-blockable$(s_1, s_2) \setminus$ *refusals*(s_1, s_2), $(s_1, s_2) \xrightarrow{a/T} (r_1, r_2)$, and $T' \subseteq T$ such that $(s_1', s_2') \xrightarrow{a/T'} (r_1', r_2')$, $(r_1, r_1') \in R_1, (r_2, r_2') \in R_2$,

   then,

$$((r_1, r_2), (r_1', r_2')) \in R$$

Similarly, R is a simulation relation by structural induction. So, $C_1 \ll C_2 \sqsubseteq_{alt} C_1' \ll C_2'$.

□

3.6 Discussion and Conclusion

3.6.1 Internal Transitions

The component follows the run-to-complete semantics of the provided services. An invocation to the provided services is executed in the atomic way, i.e. once one provided service is invoked, before its completion no other invocation to provided services can be accepted. However, in composition with other components, *internal transitions* may interleave the required traces. Next, we will discussion about how internal transitions are kept on interface model.

Internal transitions are kept on interface model mainly because the required traces for internal transitions cannot be removed or be put to the previous or next transitions, which can change the scope for run-to-complete, and we show these in the following examples.

The component automaton C with internal transition is shown in Fig. 3.20. First, we consider removing the internal transition by adding the required traces to the previous transitions. Then we can interpret the "required traces" as causality traces which means that invocation to provided event a may cause $\langle x, y\rangle$. It is obvious to see that a can get blocked in $C' \otimes C_1$, while non-blockable in $C \otimes C_1$

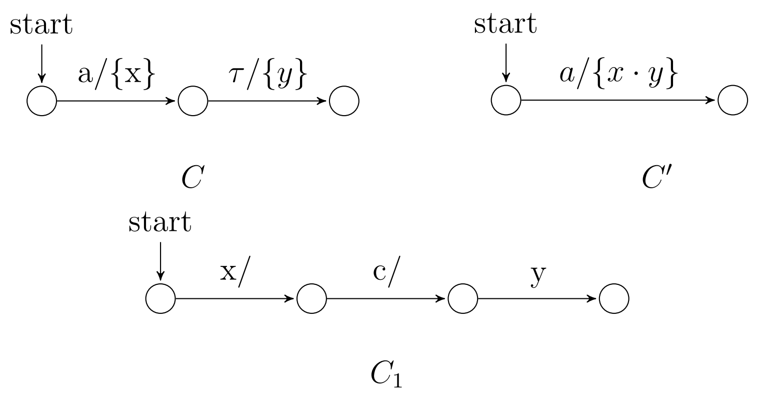


Figure 3.20: Traces that are caused

Let's automata shown in Fig. 3.21, if we put the required traces to the next transition, we can interpret as required traces. It means that in order to call a, the environment should provide x before. However, we can see that $C' \otimes C_1$ is empty, while $C \otimes C_1$ is shown in Fig. 3.21.

Besides, even in product of C_1 and C_2, we claim that the internal events are $A_1 \cup A_2$. However, strictly, it is a subset of $A_1 \cup A_2$. Internal events may happen independently and it is determined by the component when to take the internal step. However after being composed with anther component, some "internal" events may disappear. For example, in a component, there is $0 \xrightarrow{a/T_0} 1 \xrightarrow{;b/T_1} 1 \xrightarrow{c/T_3} 2$. If in another component for provided event p which requires $a \cdot c$, then internal event $;b$ will be moved inside the scope of provided event p. Then $;b$ is not internal event any more and can be abstracted away without affecting the composition.

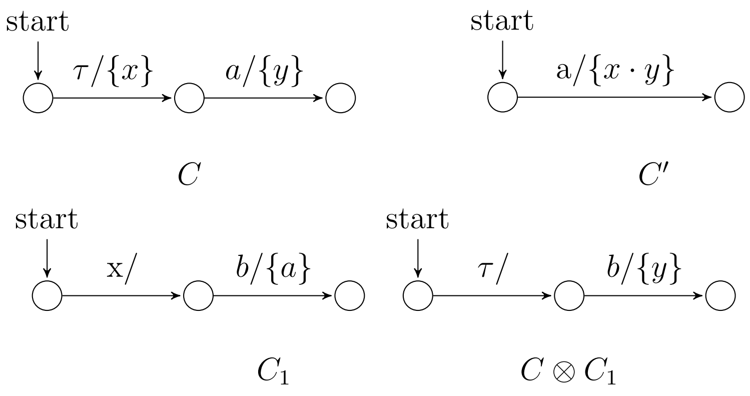


Figure 3.21: Traces that are required

3.6.2 Substitution

The refinement relation we defined is proved for interface substitution, not any component automaton. Let's consider one counter example shown in Fig. 3.22 that from the intuitive idea, we know that $\mathcal{I}(C_1) \parallel \mathcal{I}(C_2)$ is refined by $C_1 \otimes C_2$, which shows that refinement relation is not preserved in the general composition of component automata. So further study on refinement, specifically substitution of general component automata, is left for future work.

3.6.3 Summary

We have presented a model of components that abstracts the data states away, thus focusing only on interactions and study the non-determinism caused by both internal behavior and non-deterministic choice of required events. Thus, we define the interface property as all the services provided are non-blockable with all the required services are given. We then study the interface property is equivalent to input-determinism which states that after any transitions with same sequence of provided events, the component will be at states that have same non-blockable provided events. An algorithm is developed to check whether a given component automaton satisfies the interface property or not. Further, an algorithm is developed to generate the interface model for any given component automaton while preserving all the non-blockable behaviors.

Components are composed by service invocation, thus, component automata syn-

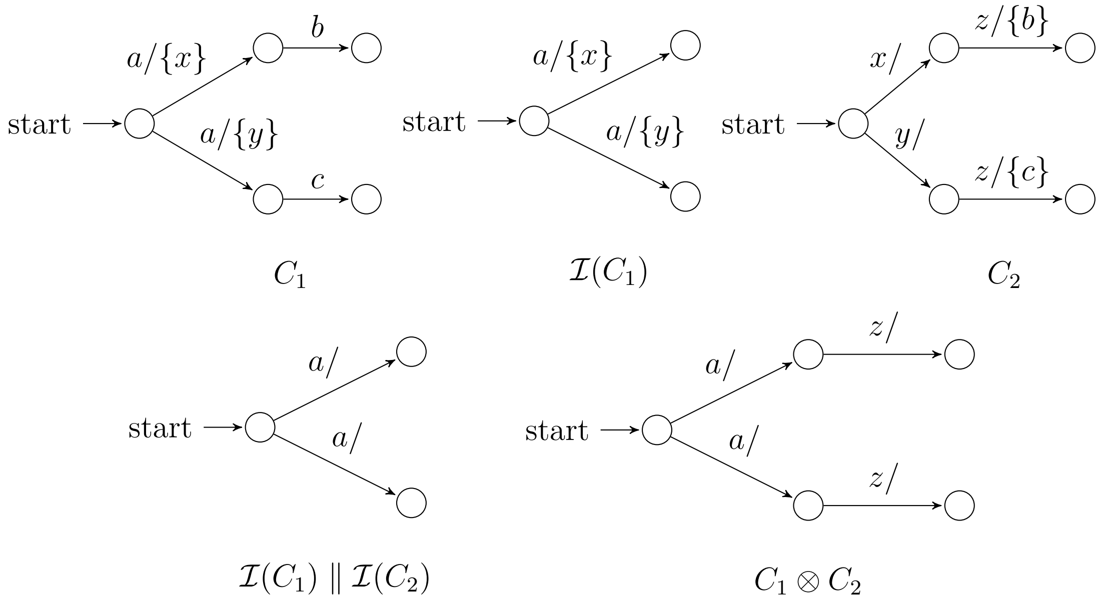


Figure 3.22: Component substitution

chronize on shared events that are provided in one and required in the other. One specific composition operation, called plugging, is defined for a closed component and an open component. A refinement relation between component automata is defined by state simulation technique with the intuitive idea that a refined component would provide more services while requiring less.

Chapter 4

Trace-based Model of Components

In the previous chapter, we gave operational descriptions of the interaction behaviors of components by automata-based models. In this chapter, we will present a trace-based model of components, called *failure models*, which can be taken as denotational descriptions of components. The basic element of the trace-based model consists of a sequence of service invocations and a set of service invocations that may be blocked. The related concepts can also be defined for the failure model, such as input-determinism, non-blockableness, composition operation, and refinement, etc. For any component, the failure model can be directly obtained from the automata-based model and it is proved that the concepts defined in these two models are consistent.

In Chapter 2, the *publication* of a component specifies what a component provides and requires, and also the protocol of the dependency order of invocation to provided and required methods. The order of method invocations of a component can be deduced from the guards of methods and the initial state.

4.1 Trace-based Model

In Chapter 3, traces and refusals are defined for component automata. In this part, we use these as basic elements to describe the possible interaction behaviors that are allowed and refused in the components.

Same as in component automata, we use P, R, and A as the set of events representing invocation to provided, required, and internal services. The service invocation is a pair of provided/internal event and a set of sequences of required events. For example, in a buffer component, the method body of *put* includes a required method call *write*. The execution of a complete invocation to *put* is modeled as a pair $(put, write)$. Generally, the alphabet set of all provided/internal invocations is $\Sigma(P, A, R) = (P \cup A) \times \mathbb{R}$ where $\mathbb{R}$ is the set of non-empty regular sets over R.

In the following, without explicit saying, we use P_i, R_i, and A_i as the set of provided, required, and internal events, respectively for component M_i.

Definition 4.1.1 (Trace)**.** *In component M, a trace is a sequence of elements in $\Sigma(P, A, R)$. The set of all traces of M is denoted as $traces(M)$, and traces are prefixed closed in $traces(M)$.*

The traces of component M are all the possible interaction behaviors, but some may be blocked due to non-determinism caused by internal event, non-deterministic choice of required events, and abstraction. Traces are not enough to express non-determinism compared with state transition systems.

For example, $\langle w, a\rangle$ is a trace in the closed component described in Fig. 4.1(I), but it may be blocked due to non-deterministic choice between the next states after $\langle a\rangle$. And similarly the trace $\langle w\rangle$ may be blocked in the closed component described in Fig. 4.1(II), because the component may transit to state 1 due to internal event.

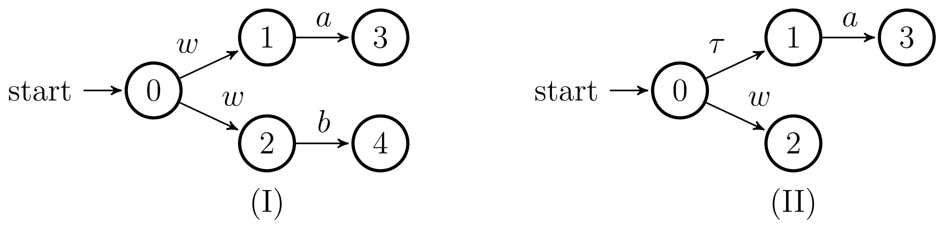


Figure 4.1: Non-determinism

Another problem of traces is that some trace may directly lead to the deadlock state or livelock state, like in the component described in Fig. 4.2(I) and (II).

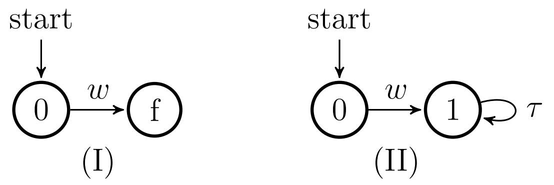


Figure 4.2: Deadlock and Livelock

In CSP [HH98, Ros98], the set *refusals* is introduced to describe the set of events that are refused. For example, both $\{a\}$ and $\{b\}$ are refusal in Fig. 4.1, but $\{a, b\}$ is not, because a and b cannot be both refused after $\langle w \rangle$. In practice, we should avoid such uncertainty, so we call $non-blockable$ all the events that cannot be refused. A pair of a trace and a set of non-blockable events is called a failure of the component.

Definition 4.1.2 (Failure)**.** *A pair $(tr, \mathcal{N}) \in traces(M) \times 2^P$ is called a failure of component M, if after the execution of trace tr, only provided events in $\mathcal{N}$ are non-blockable. All the failures of component M is denoted as $failures(M)$ and given a failure $(tr, X) \in failures(M)$, then there exists (a, T) with $a \in X$ and X' such that $(tr\hat{~}\langle (a, T) \rangle, X') \in failures(M)$.*

The definition implies that given $(tr, \mathcal{N}_1)$ and $(tr, \mathcal{N}_2)$ of component M, then $\mathcal{N}_1 = \mathcal{N}_2$. So, given a failure (tr, X), we denote $X = \mathcal{N}(tr)$.

4.1.1 Failure Model of Components

Now, a formal definition of failure model of components is given in the following.

Definition 4.1.3 (Failure model of components)**.** *A failure model of component M is a tuple $(P, R, A, failures(M))$ where P, R, and A are the set of provided, required, and internal events, respectively, and $failures(M) \subseteq 2^{\Sigma(P,A,R) \times 2^P}$ is the failure set.*

In the following, the failure model of M_i is denoted as $(P_i, R_i, A_i, \mathcal{F}(M_i))$ for any subscript i.

Given a failure model M, it is easy to derive all the traces of M, that is,

$$traces(M) = \{tr \mid (tr, X) \in failures(M)\}.$$

A provided trace of component is a possible sequence of events that the component can provide to the environment, that is, pt is a provided trace, if there exists a trace $tr \in traces(M)$ and $pt = tr \upharpoonright P$. We also say the provided trace of tr is pt, denoted as $pt = ptraces(tr)$. The set of all the provided traces of component M is written as $ptraces(M)$.

Given a provided trace pt of component M, the required traces of pt is

$$rtraces(pt) = \{conc(sq) \mid \exists tr \bullet pt = \pi_1(tr) \wedge sq = \pi_2(tr)\}$$

. The set of all required traces of component M is written as $rtraces(M)$.

Similarly, we say a provided trace is *non-blockable*, if it can be called without being blocked by the component with all the required traces are provided.

Definition 4.1.4 (non-blockableness)**.** *A provided trace $pt = \langle a_0, \cdots, c_k\rangle$ with $k > 0$ is* non-blockable*, if for any prefix $pt' = \langle a_0, \cdots, c_i\rangle$ with $k > i > 0$ of pt and for any failure $(sq, X) \in \mathcal{F}(M)$ such that $pt' = ptraces(sq)$, we have $a_{i+1} \in X$. A trace tr is non-blockable, if $ptraces(tr)$ is non-blockable.*

Given a provided trace pt, we denote non-blockable(pt) as $\bigcap\{\mathcal{N}(tr) \mid ptraces(tr) = pt\}$.

For the failure model M, the set of all non-blockable provided traces is written as *uptraces*(M) and the set of all non-blockable traces is written as *utraces*(M).

The failure model is a denotational description of components and it may also contain traces that may be blocked. So we present an input-deterministic failure model in which all traces are non-blockable.

Definition 4.1.5. *A failure model M is input-deterministic, also called interface model, if all the traces are non-blockable.*

In last chapter, components are modeled as component automata, and we give the definition of failures for component automata.

Definition 4.1.6 (failure of component automata)**.** *Let C be a component automaton, $(tr, X) \in traces(C) \times 2^P$ is a failure, if*

$$tr \in traces(C) \text{ and } X = \bigcap_{s_0 \overset{tr}{\Longrightarrow} s} \text{non-blockable}(s).$$

All the failures of C is written as failures(C).

Next, we give the failure model of components described as component automata.

Definition 4.1.7 (failure model of component automata)**.** *The failure model of component automaton C is $(P, R, A, failures(C))$ and written as $[\![C]\!]_F$.*

The following theorem states the failure model is consistent with component automata in traces, provided traces, and input-determinism.

Theorem 4.1.1. *Let C be a component automaton and $[\![C]\!]_F$ is the failure model of C. So,*

– *$traces(C) = traces([\![C]\!]_F)$, $ptraces(C) = ptraces([\![C]\!]_F)$,*

– *C is input-deterministic, if and only if $[\![C]\!]_F$ is input-deterministic.*

Proof. For any trace $tr \in traces(C)$, it implies that there exists state s such that $s_0 \stackrel{tr}{\Longrightarrow} s$, so there is a failure $(tr, X) \in failures(C)$. Then $tr \in traces([\![C]\!]_F)$. For any trace $tr \in traces[\![C]\!]_F$, then there exists X such that $(tr, X) \in failures([\![C]\!]_F)$, so $(tr, X) \in failures(C)$. That means $tr \in traces(C)$. From above, $traces(C) = traces([\![C]\!]_F)$. Then it is obvious that $ptraces(C) = ptraces([\![C]\!]_F)$.

Since the input-determinism is equivalent with that all traces are non-blockable, we only need to prove that for any provided trace pt that pt is non-blockable in C, if and only if pt is non-blockable in $[\![C]\!]_F$. First, we prove the direction from left to right. Let pt be $\langle a_1, \cdots, a_k\rangle$ with $k \geq 0$. For any $k > i \geq 0$ and any state s that $s_0 \stackrel{tr}{\Longrightarrow} s$ with $ptraces(tr) = \langle a_0, \cdots, a_i\rangle$, $a_{i+1} \in$ non-blockable(s). So there exists a failure $(tr, X) \in failures(C)$ such that $a_{i+1} \in X$. By Definition 4.1.4, we get pt is non-blockable in $[\![C]\!]_F$. The other direction can be proved similarly. So, C is input-deterministic if and only if $[\![C]\!]_F$ is input-deterministic. □

4.1.2 Plugging Operation

In this part, we present the plugging operation of failure models. In component-based design, the final goal for component software designers is to make components closed, which means that components provide services or methods without relying on other components so that they can be reused directly.

Definition 4.1.8 (pluggable)**.** *Given two failure models M_1 and M_2, M_2 is pluggable to M_1, if the following conditions are satisfied*

– *$P_2 \subseteq R_1$ and $R_2 = \emptyset$,*

– *$A_2 \cap (P_1 \cup R_1 \cup A_1) = \emptyset$*

The plugging operation is to check whether the requirement of open components can be satisfied by the given closed components and return the composite components.

Definition 4.1.9 (Plugging)**.** *Given two failure models M_1 and M_2, and M_2 is pluggable to M_1, the plugging $M_1 \ll M_2$ is given by*

– *$P = P_1$,*

- $R = R_1 \setminus P_2$,
- $A = A_1 \cup A_2$,
- $(tr, X) \in \mathit{failures}(M_1 \ll M_2)$, *if*
  - $(tr_1, X_1) \in \mathit{failures}(M_1)$,
  - $\mathit{rtraces}(tr_1) \upharpoonright P_2 \subseteq \mathit{uptraces}(M_2)$,
  - $tr = tr_1 \downharpoonright P_2$,
  - $X = \{a \in X_1 \mid tr^\frown\langle(a, T)\rangle \in \mathit{traces}(M_1) \wedge \mathit{rtraces}(tr^\frown\langle(a, T)\rangle) \upharpoonright P_2 \subseteq \mathit{uptraces}(M_2)\}$.

The plugging component C_1 is compatible with the plugged component C_2, if C_1 can provide all the services required by C_2 non-blockably.

Definition 4.1.10 (Compatibility)**.** *Given two failure models of components M_1 and M_2 such that M_2 is pluggable to M_1, we say M_2 is compatible with M_1 if $\mathit{rtraces}(M_1) \upharpoonright P_2 \subseteq \mathit{uptraces}(M_2)$.*

We see that the compatibility checking is easily done by trace-inclusion checking.

The following theorem shows the compositional properties of failure models.

Theorem 4.1.2. *Given component C_1 and C_2, if C_2 is pluggable to C_1, then $[\![C_2]\!]_F$ is pluggable to $[\![C_1]\!]_F$, and $[\![C_1]\!]_F \ll [\![C_2]\!]_F = [\![C_1 \ll C_2]\!]_F$.*

Proof. If C_2 is pluggable to C_1, it is trivial that $[\![C_2]\!]_F$ is pluggable to $[\![C_1]\!]_F$ by Definition 3.4.4 and Definition 4.1.9. The next part of the theorem follows directly from Theorem 3.4.1. □

4.2 Refinement

In Chapter 3, refinement is defined by state simulation. In this part, we study the refinement relation between components by failure set. The preorder of traces defined in Definition 3.5.2 is used to give the following definition.

Definition 4.2.1 (Failure refinement)**.** *Given two failure models $M_1 = (P_1, R_1, A_1, \mathcal{F}_1)$ and $M_2 = (P_2, R_2, A_2, \mathcal{F}_2)$, M_2 is a refinement of M_1, denoted as $M_1 \sqsubseteq_f M_2$, if*

– *$P_1 = P_2$, $R_1 = R_2$, and $A_1 = A_2$;*

– *$uptraces(M_1) \subseteq uptraces(M_2)$.*

– *Given any $pt \in uptraces(M_1)$,*

  – *if $tr_2 \in utraces(M_2)$ with $ptraces(tr_2) = pt$, there exists $tr_1 \in utraces(M_1)$ such that $tr_2 \preceq tr_1$ and $\mathcal{N}(tr_2) \setminus \text{non-blockable}(ptraces(tr_2)) \subseteq \mathcal{N}(tr_1) \setminus \text{non-blockable}(ptraces(tr_1))$;*

  – *if $tr_1 \in utraces(M_1)$ with $ptraces(tr_1) = pt$, there exists $tr_2 \in utraces(M_2)$ such that $tr_2 \preceq tr_1$ and $\mathcal{N}(tr_2) \setminus \text{non-blockable}(ptraces(tr_2)) \subseteq \mathcal{N}(tr_1) \setminus \text{non-blockable}(ptraces(tr_1))$;*

The next theorem states that the refinement relation of failure models is consistent with that of automata-based models

Theorem 4.2.1. *Let C_1 and C_2 be two component automata, $C_1 \sqsubseteq_{alt} C_2$, iff $[\![C_1]\!]_F \sqsubseteq_f [\![C_2]\!]_F$.*

Proof. The proof is:

– " $\Rightarrow$ " : It is naturally from Theorem 3.5.2.

– " $\Leftarrow$: " We introduce relation $\mathcal{R} = \{(s_1, s_2) \mid ptraces(tr_1) = ptraces(tr_2) = pt \in uptraces(C_1) \wedge tr_1 \preceq tr_2 \wedge s_0^1 \overset{tr_1}{\Longrightarrow} s_1 \wedge s_0^2 \overset{tr_2}{\Longrightarrow} s_2\}$, which is obviously an alternating simulation. So $C_1 \sqsubseteq_{alt} C_2$.

□

4.3 Summary

This chapter presents a failure model of component software. Though, we have similar definitions in Chapter 3, in this chapter, we want to model and describe components by the failure set instead of deriving from the automata-based models. The atomic behavior of the component is the procedure of a service request from the environment or internally triggered by the component itself, and a set of possible sequences of services invocations which are provided by the environment. Such atomic behaviors should happen in the certain orders constrained by the flow of control, so the interaction behaviors of the component are sequences of atomic behaviors, which are called traces. The component may end up in deadlock or livelock due to non-determinism which may be caused by the internal behavior or non-deterministic choice of the required services. Non-blockable events are used to guarantee avoidance of the deadlock or livelock.

The failure model of components can be derived from the automata-based models. Components are composed by plugging the component which provide services to the components which need these services. Two components are compatible if the requirement of the plugged component is satisfied by the plugging component. The failure model supports easier compatibility checking by the checking trace inclusion of non-blockable provided traces of one closed component and required traces of the plugged open component.

The refinement relation between failure models is defined based on trace inclusion and proved to be consistent with state simulation in component automata.

There are some challenging work left for the future, such as the general composition operation of failure models, and the completeness and soundness properties.

Chapter 5

Coordination

In this chapter, we will present two kinds of software entities to adapt the use of software components more flexible.

5.1 Process Component

Process components (processes, for short) of rCOS are well introduced and studied in Section 2.4. In this section, we will present the automata-based model for the processes and automata-based composition operation of processes. A process behaves as a client of services and make service invocations actively based on its own control of flow.

The interface models of components we have studied in Chapter 3 describe the interaction behaviors of components by the provided/required relations. A software component acts as service provider and it behaves in a passive way that it starts to work only when the provided services are invoked by the environment and it then may need to require services from the environment to satisfy the requirement of the environment.

The process component is built

5.1.1 Process Automata

The events issued by processes are called *actions.* Required events in component automata should be guaranteed when the corresponding provided events are triggered, otherwise, deadlock will happen, while actions of processes will happen whenever the they are provided and enabled by the components.

Definition 5.1.1 (Process Automata)**.** *A process automaton is modeled as automaton $P = (Q, q_0, A, \delta)$ where Q is a set of states, $q_0 \in Q$ is the initial state, A is the alphabet set of actions, and δ is the transition relation.*

5.1.2 Coordination

A process coordinates a component by actively triggering the provided events of the component.

Definition 5.1.2 (Coordination)**.** *Given a component automaton $C = (S, s_0, f, P, R, A, \delta)$ and a process automaton $P = (Q, q_0, E, \delta_P)$, C is coordinatable by P if $E \subseteq P$, and coordination C' is $C \times P = (S', s'_0, f, P', R, A', \delta')$ where*

- *$S' = S \times Q$, $s'_0 = (s_0, q_0)$, and (f, q) for any $q \in Q$ is noted as f;*
- *$P' = P \setminus E$;*
- *$R' = R$;*
- *$A' = A \cup E$;*
- *transition set δ' is defined as the smallest set by the following rules. For any reachable state (s, q) where $s \xrightarrow{m/T} s'$ and $q \xrightarrow{e} q'$,*
  1. *if $m == e$, then $(s, q) \xrightarrow{;m/T} (s', q')$,*
  2. *if $m \in P'$, then $(s, q) \xrightarrow{m/T} (s', q)$.*

For example, given two one-place buffer, one may want to connect these two buffers to build a buffer with a larger capacity.

Example 5.1.1. *Consider the example of component alarm, blink, and sound shown in Figures 3.5 and 3.2. Component timer provides all services that component alarm needs. So the composition of these components is a closed component shown in Figure 5.2. We consider a process P_{Ring} which will internally and autonomously invoke ring based on internal clock.*

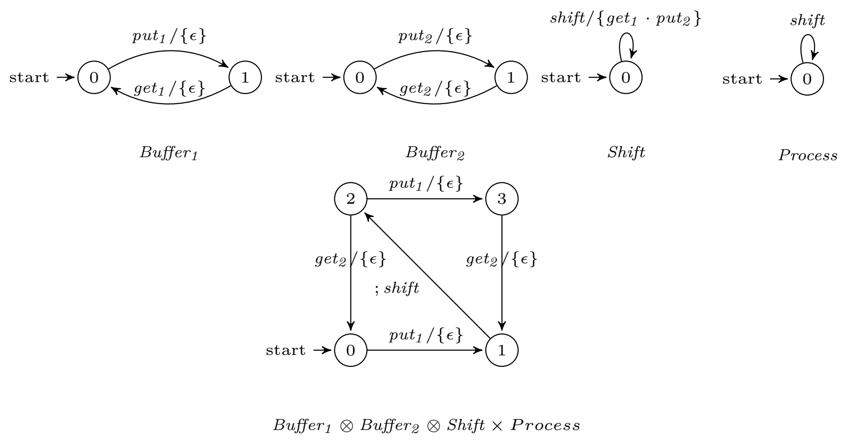


Figure 5.1: one-place buffers and coordination

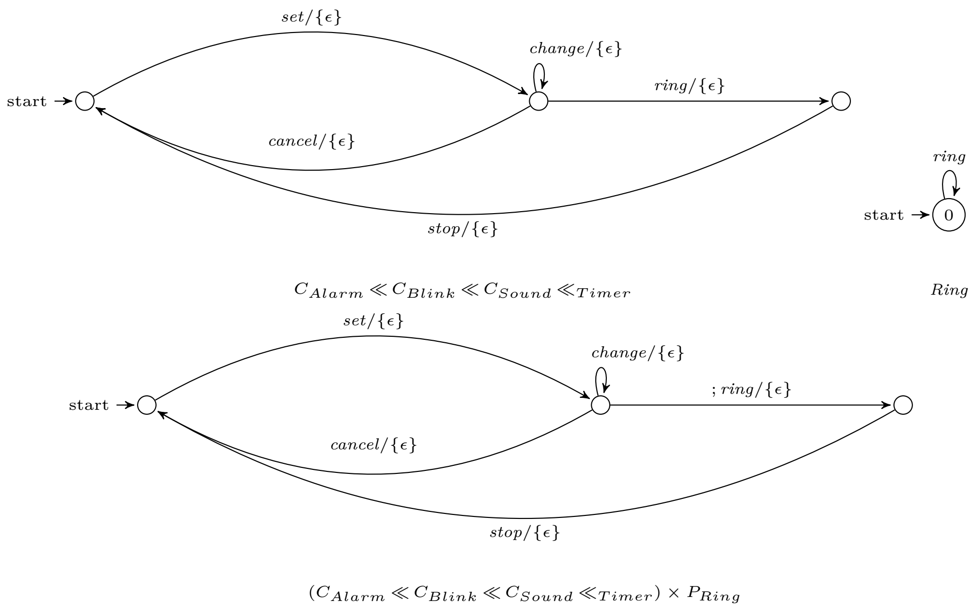


Figure 5.2: Composition of C_{Alarm} and C_{Blink} and C_{Sound} and C_{Timer}

5.2 Coordinator Component

In this part, we introduce a coordinator component which is used to adapt the components to be reused by obeying new constraints. Components are composed in the way that they synchronize on services that are provided by one and required by the other, and behave independently otherwise. A coordinator is used to avoid invocations to services that the components want to hide from the environment. Coordination operation between components and coordinators is defined and an algorithm is developed to synthesize a coordinator for any given component such that the coordinator could make the component avoid possible deadlock.

The allowed or unblockable order of invocation to provided services is implicitly implemented in the components, that is, computation is mixed with protocol. As in the rCOS components introduced in Chapter 2, a *guard* is used to control accessibility to services in the provided interface. This limits reuse of components and also is against the principle of separation of concern.

For example, consider a one-place buffer which provides services *put* and *get* which, in turn, require *write* and *read* from the memory, respectively. If it is designed and implemented as in Fig. 5.3(B_1), then this buffer can only be reused in the context with strict constraint about the order of *put* and *get*. However, if it is designed as in Fig. 5.3(B_2), then this buffer can be more widely reused. Software engineers can reuse buffer B_2 by designing a specific component as the protocol which is simpler, because it considers only the control of flow without worrying about the functionality or computation of services. This specific component can be called a filter, adapter, or connector, and we use the term *coordinator* in this thesis. In the example shown in Fig. 5.3, we can see that coordination of B_2 by C_1 is exactly equivalent with buffer B_1 in the sense of functionality of services and protocol. And buffer B_2 can be also reused in the context where data is not wiped after invocation to *get* and can be continuously reused until the data is updated by invocation to *put*, which is achieved by coordinating the buffer by coordinator C_2 shown in Fig. 5.3.

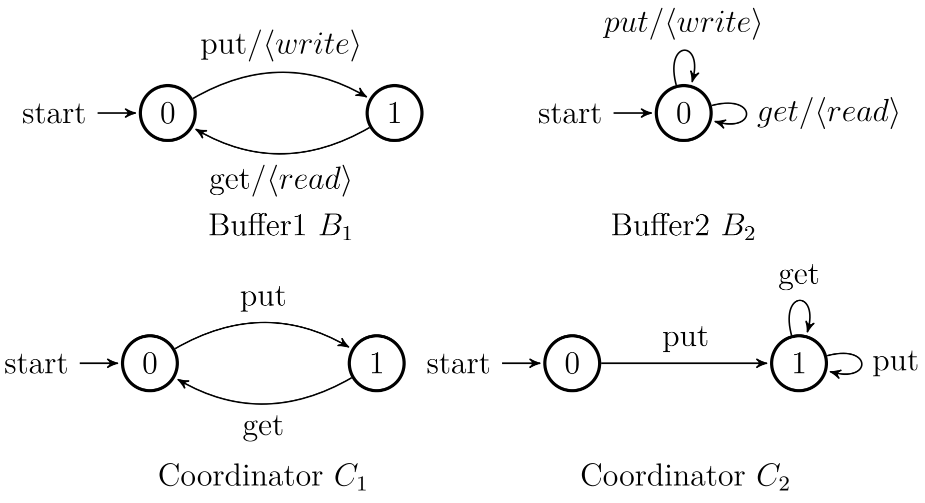
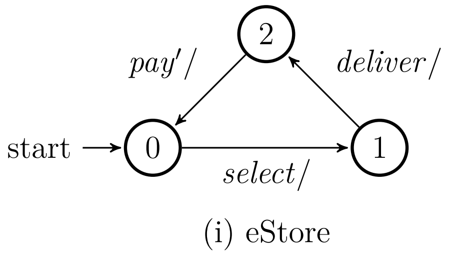


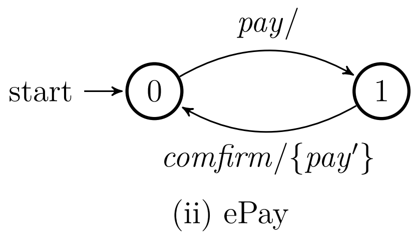
Figure 5.3: one-place buffer

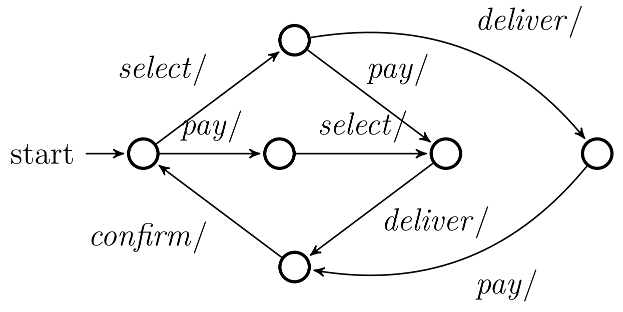
The above examples shows that separation of computation and protocol can increase reuse of components. Now, we use a more practical example showing that a coordinator can be used to filter out services that may cause unfairness.

Example 5.2.1. *Consider an online marketplace system which provides a consumer-*

to-consumer platform for retail stores. It consists of stores and a payment component trusted by both stores and clients. The store component, called eStore*, is presented in Fig. 5.4(i). It provides services select, pay′, and deliver, which model selecting items, obtaining the money from payment component, and delivering the paid items to the clients, respectively. The payment component, called* ePay*and shown in Fig. 5.4(ii), provides services pay and confirm which model receiving money from the clients and being confirmed by the client after the items are received. It requires service pay′ that the component will transfer the money to the store. The composition of* eStore *and* ePay *is in Fig. 5.4(iii).*







(iii) $eStore \otimes ePay$

Figure 5.4: online shopping system

In the above example, the provided trace $\langle select \cdot deliver\rangle$ is allowed, which means that the store may not get paid even if it delivers the items bought by the clients. So such online marketplace system is unsafe for the store retailers. We introduce a kind of specific components, called *coordinator*, to filter out services provided by components that should not be allowed.

A coordinator is modeled as a labeled transitions system, the formal definition is given below.

Definition 5.2.1 (coordinator)**.** *A coordinator F is a deterministic labeled transition system (Q, q_0, E, σ), where*

– *Q is the set of states with $q_0 \in Q$ as the initial state;*

– *E is the set of active events;*

– *σ is a set of transition.*

Similarly, the set of traces of coordinator F, written as *traces*(F), is

$$\{\langle a_0, a_1, \cdots, a_k\rangle \mid q_0 \xrightarrow{a_0} \cdots \xrightarrow{a_k} q_{k+1}\}.$$

The default signature of F_i is $(Q_i, q_0^i, E_i, \sigma_i)$ for any subscript i.

Example 5.2.2. *In order to filter out the unexpected provided traces in Fig. 5.4(iii), we can exploit coordinator F shown in Fig. 5.5.*

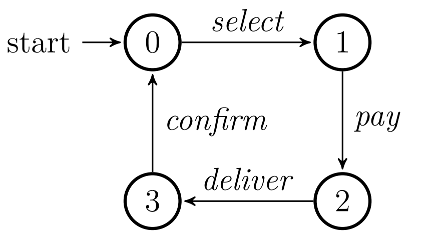


Figure 5.5: Coordinator F

5.2.1 Coordination Operation

Components are coordinated in the way that all the sequences of services provided should also obey the constraint of the coordinators. The formal definition is given below.

Definition 5.2.2 (Coordination)**.** *Given a component automaton C and a coordinator F, we say C is coordinatable by F, if $E \subseteq P$, and the coordination of C by F, $C \ltimes F$, is a component automaton $(S', s'_0, f, P', R', A', \delta')$, where*

– *$S' = S \times Q$, $s'_0 = (s_0, q_0)$, and (f, q) with any q is denoted as f;*

– *$P' = P$, $R' = R$, and $A' = A$;*

– *δ' is a set of transitions complying with the following rules:*

  – *if $s \xrightarrow{w/T} s'$ and $t \xrightarrow{w} t'$, then $(s,t) \xrightarrow{w/T} (s', t')$;*

– $s \xrightarrow{w/T} s'$ *with* $w \notin E$, *then* $(s,t) \xrightarrow{w/T} (s',t)$.

The following theorem shows the failure set of coordination.

Theorem 5.2.1. *Given component automaton C and coordinator F, if C is coordinatable by F, the failure set for coordination $C \ltimes F$, $failures(C \ltimes F)$ is $\{(tr, D \setminus D') \mid (tr, D) \in failures(C), \pi_1(tr){\upharpoonright} E \in traces(F), D' = \{a \mid \pi_1(tr){\upharpoonright} E^\frown\langle a\rangle \notin traces(F)\}\}$*

Proof. □

The following corollary is trivially derived from Theorem 5.2.1.

Corollary 5.2.1. *Given component automaton C and coordinator F, if C is coordinatable by F, then*

- $traces(C \ltimes F) = \{tr \mid tr \in traces(C) \wedge ptraces(C) \subseteq traces(F)\}$,
- $ptraces(C \ltimes F) = \{tr \mid tr \in ptraces(C) \wedge ptraces(C) \subseteq traces(F)\}$,
- $utraces(C \ltimes F) = \{tr \mid tr \in utraces(C) \wedge ptraces(C) \subseteq traces(F)\}$, *and*
- $uptraces(C \ltimes F) = \{tr \mid tr \in uptraces(C) \wedge ptraces(C) \subseteq traces(F)\}$.

Example 5.2.3. *Now, we can see how the component eStore ⊗ ePay shown in Fig. 5.4(iii) is coordinated by coordinator in Fig. 5.5. The result is presented in Fig. 5.6.*

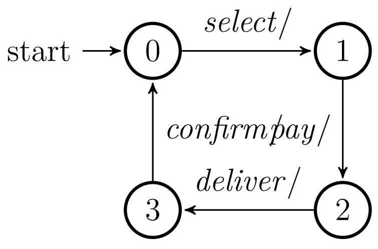


Figure 5.6: Coordination of $(eStore \otimes ePay) \ltimes F$

5.2.2 Synthesizing Interface Coordinator for Component Automata

In this part, we will show that given any component automaton C, there exits a coordinator F such that coordination of C by F is equivalent the component interface automaton $\mathcal{I}(C)$ constructed by Algorithm 2. In practice, however, it can be implemented by designing a coordinator which can filter out all the possible blockable traces.

We now present a procedure in Algorithm 3 that, given a component automaton, constructs a coordinator which only records unblockable provided traces. The basic idea is similar to the construction of a deterministic automaton from a non-deterministic one, and the only difference is that in the algorithm only the deterministic traces are kept.

Algorithm 3: Construction of Interface coordinator

```
Input: $C = (S, s_0, f, P, R, A, \delta)$
Output: $\mathcal{G}(C) = (Q, q_0, E, \sigma)$
 1: Initialization: $q_0 = \{s' \mid s' \in intR(s_0)\}, Q := \{q_0\}, E := P, \sigma := \emptyset$,
    $todo := \{q_0\}$
 2: while $todo \neq \emptyset$ do
 3:   choose one $q \in todo$ and $todo := todo \setminus \{q\}$
 4:   for each $a \in \bigcap_{s\in q}$ non-blockable($s$) do
 5:     let $q'$ be $\{s' \mid s \in q \bullet s \overset{\langle a\rangle}{\Longrightarrow} s'\}$
 6:     if $q' \notin Q$ then
 7:       add $q'$ to $Q$ and $todo$
 8:     end if
 9:     $\sigma := \sigma \cup \{q \xrightarrow{a} q'\}$.
10:   end for
11: end while
```

Lemma 5.2.1. *Given any component automaton C, let $\mathcal{G}(C) = (Q, q_0, E, \sigma)$. Then $q_0 \overset{sq}{\Longrightarrow} q$, iff*

$$q = \{s' \mid sq \in uptraces(C) \wedge s_0 \overset{sq}{\Longrightarrow} s'\}$$

.

Proof. We prove the following by induction on sq.

– $\Rightarrow$. The base case follows trivially. For $q_0 \overset{sq\hat{}\langle a\rangle}{\Longrightarrow} q_1$, then there exists q_2 such that $q_0 \overset{sq}{\Longrightarrow} q_2$ and $q_2 \xrightarrow{a} q_1$. By induction hypothesis, $q_2 = \{s' \mid sq \in uptraces(C) \wedge s_0 \overset{sq}{\Longrightarrow} s'\}$. From Line 9 and Lines 4-8 of Algorithm 3, we have $a \in \bigcap_{s\in q_2}$ non-blockable(s), then $sq\hat{}\langle a\rangle \in uptraces(C)$ because of $sq \in uptraces(C)$. And $q_1 = \{s' \mid s \in q_2 \bullet s \overset{\langle a\rangle}{\Longrightarrow} s'\}$, that is $q_1 = \{s' \mid sq\hat{}\langle a\rangle \in uptraces(C) \wedge s_0 \overset{sq}{\Longrightarrow} s'\}$. This proves the direction " $\Rightarrow$ ".

– $\Leftarrow$. The base case follows trivially. For a state $q = \{s' \mid sq\hat{}\langle a\rangle \in \mathit{uptraces}(C) \wedge s_0 \xRightarrow{sq\hat{}\langle a\rangle} s'\}$, then, there exists $q' = \{s' \mid sq \in \mathit{uptraces}(C) \wedge s_0 \xRightarrow{sq} s'\}$. Then $q = \{s' \mid s \in q' \bullet s \xRightarrow{\langle a\rangle} s'\}$ and $a \in \bigcap_{s\in q'}$ non-blockable(s), because $sq\hat{}\langle a\rangle \in \mathit{uptraces}(C)$. By induction hypothesis, we have $q_0 \xRightarrow{sq} q'$. And, $q = \{s' \mid s \in q' \bullet s \xRightarrow{\langle a\rangle} s'\}$. From line 4-9 of Algorithm 3, there exists $q' \xrightarrow{a} q$, then $q_0 \xRightarrow{sq\hat{}\langle a\rangle} q$. This proves the direction " $\Leftarrow$ ".

$\square$

Three key correctness properties of the algorithm are stated in the following theorem.

Theorem 5.2.2 (Correctness of Algorithm 3)**.** *Given any component automaton C, the following properties hold for Algorithm 3:*

– *the algorithm always terminates;*

– *$\mathcal{G}(C)$ is deterministic;*

– *$\mathit{traces}(C \ltimes \mathcal{G}(C)) = \mathit{utraces}(C)$.*

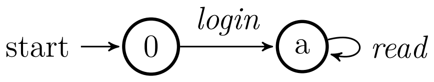
Proof. – The termination of the algorithm is obtained, because *todo* will eventually be empty: the size of power set of state S is bounded, only fresh state is added to *todo*, and for each iteration of the loop a state from *todo* is removed.

– Assume that there exists $q \xrightarrow{a} q_1$ and $q \xrightarrow{a} q_1$, then from Algorithm 3, we have $q_1 = q_2 = \{s' \mid s \xRightarrow{tr} s'$, with $s \in q \bullet \mathit{ptraces}(tr) = \langle a\rangle\}$. So $\mathcal{G}(C)$ is deterministic.

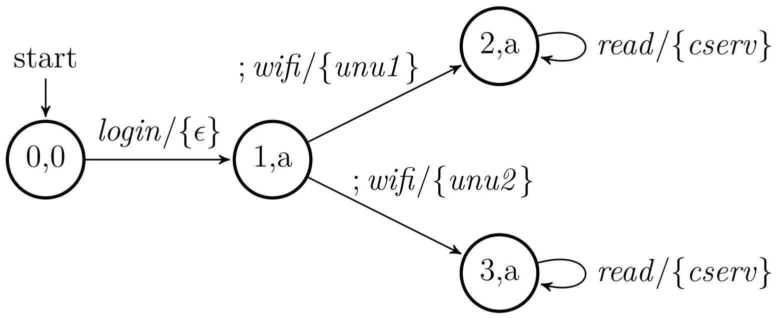
– By Lemma 5.2.1, we can get $\mathit{traces}(\mathcal{G}(C)) = \mathit{uptraces}(C)$. By Corollary 5.2.1, $\mathit{traces}(C \ltimes \mathcal{G}(C)) = \{tr \mid tr \in \mathit{traces}(C) \wedge \mathit{ptraces}(C) \subseteq \mathit{traces}(\mathit{coordinator}(C))\}$, and because $\mathit{traces}(\mathit{coordinator}(C)) = \mathit{uptraces}(C)$, so $\mathit{traces}(C \ltimes \mathcal{G}(C)) = \mathit{utraces}(C)$.

$\square$

Example 5.2.4. *The component automaton in Fig.* **??** *is not input-deterministic. A coordinator shown in Fig. 5.7(i) is obtained by Algorithm 3. We use state a as shorthand for $\{1, 2, 3\}$. The coordination of $C_{ic} \ltimes \mathcal{G}(C)$ is given in Fig. 5.7(ii)*



(i) Coordinator $\mathcal{G}(C_{ic})$



(ii) $C_{ic} \ltimes \mathcal{G}(C_{ic})$

Figure 5.7: Coordination of Component C_{ic} by a synthesized coordinator

5.3 Summary

In this chapter, we have motivated and introduced process and coordinator components, which can be used to coordinate services provided among components and filter out certain sequence of service invocation that may violate the security policy in a new context. Process components are used to actively invoke services provided by software components and coordinator components increase the flexibility of reusing components by constraining certain provided services. The coordination operation is defined and an algorithm is developed to produce an interface coordinator for any component automaton to obtain the interface model of the coordinator-automaton composition.

However, coordination is widely studied in component-based and service-oriented software development. Reo is a channel-based coordination language that can be used to coordinate and orchestrate components by connectors [Arb04, JA12, BSAR06]. BIP is formal framework for component-based design that separates computation and interaction, and the order of transitions is controlled by priority set of events in which interaction events are ordered [BB13, BBS06, BS08, BSS09, GGMC$^+$07, GS12]. In [CGP09], filters are introduced to make web services avoid deadlock. The main difference is that we provide a way of coordinating by actively invoking services or constraining services of one component instead of modeling the connectors among components

Chapter 6

Conclusion and Future Work

Nowadays, software system is growing more and more complex and widely used in our daily life and work. Developing large reliable software system effectively is a challenge in software engineering. Component-based software development is considered as a solution. Components are composed and reused through interfaces, thus a well founded interface theory is important to support component reuse, composition, verification, and substitutability. In this thesis, we focus on sequences of method invocation between components, called interaction protocols.

In this thesis, we present an interface theory of software components in component-based software development. We introduce the rCOS methodology and define full semantics model of rCOS components and derive a general labeled transition system of components. Then, we introduced the automata-based and trace-based models of component software motivated by the needs to describe interaction behaviors and check compatiblity , and bring up a kind of input-deterministic models as interfaces which assure the non-blockableness of services provided by the component. A provided service may be blocked for several reasons, for example, the guard of this service is false; invocation to such service leads the component to a state at which there is internal loop or to the error state. We have presented an algorithm to check whether any given component automaton is input-deterministic and also an algorithm to obtain the interface model of the given component automaton. Correctness of the algorithms are proved.

Refinement relation defined by state simulation technique is developed and proved to be sound for interface substitution. A refined component should be able to provide more services while requiring less, and more likely to avoid deadlock or livelock.

The automata-based model is *operational* description of interaction behaviors of components. A denotational model, called *failure model* of software components is defined for easier checking of compatiblity of components. The failure model contains sequences of allowable alternating provided/required events and a set of events that are not blocked. Refinement based on trace inclusion is defined for the failure models. Plugging composition is proved to be consistent between failure models and automata-based models.

Instead of modeling connectors between components, we present automata-based model of two coordination components for software components. Process component coordinates components by actively invoking provided services of components and internalizing these services. A coordinator component, which is a simple automaton, is introduced and proved to be able to filter out the blockable services provided by the coordinated components.

The future work can go in the following directions.

Refinement Further work is needed to study the relation between $C_1 \otimes C_2$ and $C'_1 \otimes C'_2$ where C_1 and C_2 are refined by C'_1 and C'_2, respectively. A refinement relation which can preserve both safety and liveness property is also needed.

Extension of expressiveness of required interface In this thesis, we assume that invocation to required services is totally determined by components, that is non-deterministic choice for the environment, so that in order to guarantee non-blockableness of provided traces, the environment must provide all the required traces. We would try to extend the expressiveness of the required interface, i.e, a possible subset of required traces can guarantee the related provided service.

Failure model We have defined plugging operation of failure models and a general composition operation of failure models is needed.

Extension with timing characteristics In the alarm component, we see that the *ring* service should be triggered based on the time. We would extend our model for component software with hard and soft timing assumption and guarantees, which support timing, deadlock, and scheduling analysis of applications in the presence of timed requirement.

Bibliography

[AD94] R. Alur and D.L. Dill. A theory of timed automata. *Theoretical computer science*, 126(2):183–235, 1994.

[AG97] Robert Allen and David Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 6(3), 1997.

[AHKV98] Rajeev Alur, Thomas Henzinger, Orna Kupferman, and Moshe Vardi. Alternating refinement relations. In Davide Sangiorgi and Robert de Simone, editors, *CONCUR'98 Concurrency Theory*, volume 1466 of *Lecture Notes in Computer Science*, pages 163–178. Springer Berlin / Heidelberg, 1998.

[Arb04] Farhad Arbab. Reo: a channel-based coordination model for component composition. *Mathematical Structures in Computer Science*, 14:329–366, June 2004.

[BB13] Eduard Baranov and Simon Bliudze. Extended connectors: Structuring glue operators in bip. *Electronic Proceedings in Theoretical Computer Science*, 131:20–35, October 2013.

[BBS06] A Basu, M Bozga, and J Sifakis. Modeling heterogeneous real-time components in bip. In *Software Engineering and Formal Methods, 2006. SEFM 2006. Fourth IEEE International Conference on*, pages 3–12, 2006.

[BG77] R. Burstall and J. Goguen. Putting theories together to make specifications. In Raj Reddy, editor, *Proc. 5th Intl. Joint Conf. on Artificial Intelligence*, pages 1045–1058, Department of Computer Science, Carnegie-Mellon University, USA, 1977.

[BHM05] Tomás Barros, Ludovic Henrio, and Eric Madelaine. Behavioural models for hierarchical components. pages 154–168, 2005.

[BMH06] Bill Burke and Richard Monson-Haefel. *Enterprise JavaBeans 3.0 (5th Edition)*. O'Reilly Media, Inc., 2006.

[BMP$^+$07] Ananda Basu, Laurent Mounier, Marc Poulhies, Jacques Pulou, and Sifakis, Joseph. Using bip for modeling and verification of networked systems–a case study on tinyos-based networks. pages 257–260, 2007.

[Box98] Don Box. *Essential COM.* Reading, Mass. : Addison Wesley, 1998.

[BS08] Simon Bliudze and Sifakis, Joseph. A notion of glue expressiveness for component-based systems. pages 508–522, 2008.

[BSAR06] Christel Baier, Marjan Sirjani, Farhad Arbab, and Jan J. M. M. Rutten. Modeling component connectors in reo by constraint automata. volume 61, pages 75–113. Elsevier, 2006.

[BSS09] M D Bozga, V Sfyrla, and J Sifakis. Modeling synchronous systems in bip. In *Proceedings of the seventh ACM ...*, 2009.

[BvW94] RJR Back and Joakim von Wright. Trace refinement of action systems. 1994.

[CGP09] Giuseppe Castagna, Nils Gesbert, and Luca Padovani. A theory of contracts for web services. *ACM Transactions on Programming Languages and Systems*, 31(5):1–61, June 2009.

[CHH$^+$08] Zhenbang Chen, Abdel Hakim Hannousse, Dang Van Hung, Istvan Knoll, Xiaoshan Li, Yang Liu, Zhiming Liu, Qu Nan, Joseph C. Okika, Anders P. Ravn, Volker Stolz, Lu Yang, and Naijun Zhan. Modelling with relational calculus of object and component systems–rCOS. In A. Rausch, R. Reussner, R. Mirandola, and Frantisek Plasil, editors, *The Common Component Modeling Example,* volume 5153 of *Lecture Notes in Computer Science*, chapter 3, pages 116–145. Springer, Berlin, 2008.

[CHL06] Xin Chen, Jifeng He, and Zhiming Liu. Component coordination in rCOS. Technical Report 335, UNU-IIST, P.O. Box 3058, Macao SAR, China, May 2006.

[CLL$^+$07] Zhenbang Chen, Xiaoshan Li, Zhiming Liu, Volker Stolz, and Lu Yang. Harnessing rcos for tool support: the cocome experience. In *Formal methods and hybrid real-time systems*, pages 83–114. Springer, 2007.

[CLM07] Xin Chen, Zhiming Liu, and Vladimir Mencl. Separation of concerns and consistent integration in requirements modelling. In *SOFSEM*, pages 819–831, 2007.

[CLR$^+$09] Zhenbang Chen, Zhiming Liu, Anders P. Ravn, Volker Stolz, and Naijun Zhan. Refinement and verification in component-based model driven design. *Science of Computer Programming*, 74(4):168–196, February 2009.

[CLS$^+$07] Zhenbang Chen, Zhiming Liu, Volker Stolz, Lu Yang, and Anders P. Ravn. A refinement driven component-based design. In *12th Intl. Conf. on Engineering of Complex Computer Systems (ICECCS 2007)*, pages 277–289, 2007.

[CLS08] Zhenbang Chen, Zhiming Liu, and Volker Stolz. The rCOS tool. In John Fitzgerald, Peter Gorm Larsen, and S. Sahara, editors, *Modelling and Analysis in VDM: Proceedings of the Fourth VDM/Overture Workshop*, number

CS-TR-1099 in Technical Report Series, pages 15–24, Newcastle, 2008. University of Newcastle upon Tyne.

[CM88] K. Mani Chandy and Jayadev Misra. *Parallel Program Design: A Foundation.* Addison-Wesley, Reading, 1988.

[CMS09] Zhenbang Chen, Charles Morisset, and Volker Stolz. Specification and validation of behavioural protocols in the rCOS modeler. In Farhad Arbab and Marjan Sirjani, editors, *Proceedings of 3rd IPM International Conference on Fundamentals of Software Engineering,* volume 5961 of *Lecture Notes in Computer Science*, pages 387–401, Berlin, 2009. Springer.

[CSVC11] Ivica Crnkovic, Séverine Sentilles, Aneta Vulgarakis, and Michel RV Chaudron. A classification framework for software component models. *Software Engineering, IEEE Transactions on*, 37(5):593–615, 2011.

[CZ06] Xin Chen and Naijun Zhan. A model of component-based programming. Technical report, 2006.

[DAH01] Luca De Alfaro and Thomas A Henzinger. Interface automata. *ACM SIGSOFT Software Engineering Notes*, 26(5):109–120, 2001.

[DAH05] Luca De Alfaro and Thomas A Henzinger. Interface-based design. 2005.

[DFKL13] Ruzhen Dong, Johannes Faber, Wei Ke, and Zhiming Liu. rcos: Defining meanings of component-based software architectures. 8050:1–66, 2013.

[DFL$^+$12] Ruzhen Dong, Johannes Faber, Zhiming Liu, Jiri Srba, Naijun Zhan, and Jiaqi Zhu. Unblockable compositions of software components. In *Proceedings of the 15th ACM SIGSOFT symposium on Component Based Software Engineering*, CBSE '12, pages 103–108, New York, NY, USA, 2012. ACM.

[DH01] Luca DeAlfaro and Thomas Henzinger. Interface theories for component-based design. In Thomas Henzinger and Christoph Kirsch, editors, *Embedded Software*, volume 2211 of *LNCS*, pages 148–165. Springer, 2001.

[DS90] Edsger W. Dijkstra and Carel S. Scholten. *Predicate Calculus and Program Semantics.* Springer-Verlag, New York, 1990.

[DZ14] Ruzhen Dong and Naijun Zhan. Towards a failure model of software components. In José Luiz Fiadeiro, Zhiming Liu, and Jinyun Xue, editors, *Formal Aspects of Component Software*, volume 8348 of *Lecture Notes in Computer Science*, pages 119–136. Springer International Publishing, 2014.

[DZZ13] Ruzhen Dong, Naijun Zhan, and Liang Zhao. An interface model of software components. volume 8049 of *Lecture Notes in Computer Science*, pages 159–176. Springer Berlin Heidelberg, 2013.

[EGP08] Michael Emmi, Dimitra Giannakopoulou, and Corina S. Pasareanu. Assume-guarantee verification for interface automata. In Jorge Cuéllar, T. S. E.

Maibaum, and Kaisa Sere, editors, *FM*, volume 5014 of *LNCS*, pages 116–131. Springer, 2008.

[Fis00] C. Fischer. *Combination and Implementation of Processes and Data: from CSP-OZ to Java.* PhD thesis, University of Oldenburg, 2000.

[GGMC+07] Gregor Gössler, Graf, Susanne, Mila Majster-Cederbaum, Moritz Martens, and Sifakis, Joseph. An approach to modelling and verification of component based systems. pages 295–308, 2007.

[GPB02] Dimitra Giannakopoulou, Corina S Pasareanu, and Howard Barringer. Assumption generation for software component verification. In *Automated Software Engineering, 2002. Proceedings. ASE 2002. 17th IEEE International Conference on*, pages 3–12. IEEE Computer Society, 2002.

[Gro06] Object Management Group. Corba component model 4.0 specification. Specification Version 4.0, Object Management Group, April 2006.

[GS05] Gregor Gössler and Sifakis, Joseph. Composition for component-based modeling. *Science of Computer Programming*, 55(1):161–183, 2005.

[GS12] Gregor Gössler and Gwen Salaün. Realizability of choreographies for services interacting asynchronously. pages 151–167, 2012.

[HH98] C. A. R. Tony Hoare and Jifeng He. *Unifying theories of programming*, volume 14. Prentice Hall, 1998.

[HLL05a] Jifeng He, Xiaoshan Li, and Zhiming Liu. Component-based software engineering. In *ICTAC*, pages 70–95, 2005.

[HLL05b] Jifeng He, Zhiming Liu, and Xiaoshan Li. A theory of contracts. Technical report, 2005.

[HLL06a] Jifeng He, Xiaoshan Li, and Zhiming Liu. rcos: A refinement calculus of object systems. *Theor. Comput. Sci.*, 365(1-2):109–142, 2006.

[HLL06b] Jifeng He, Xiaoshan Li, and Zhiming Liu. A theory of reactive components. *Electronic Notes in Theoretical Computer Science*, 160:173–195, March 2006.

[HLL+12] John Hatcliff, Gary T. Leavens, K. Rustan M. Leino, Peter Müller, and Matthew Parkinson. Behavioral interface specification languages. *ACM Comput. Surv.*, 44(3):16:1–16:58, June 2012.

[HMU79] J.E. Hopcroft, R. Motwani, and J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation*, volume 2. Addison-Wesley, 1979.

[HO02] Jochen Hoenicke and Ernst-Rüdiger Olderog. Combining specification techniques for processes, data and time. In Michael J. Butler, Luigia Petre, and Kaisa Sere, editors, *IFM*, volume 2335 of *Lecture Notes in Computer Science*, pages 245–266. Springer, Heidelberg, 2002.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[Hoa85] C. A. R. Tony Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[JA12] Sung-Shik Jongmans and Farhad Arbab. Overview of thirty semantic formalisms for reo. *Scientific Annals of Computer Science*, 22(1):201–251, May 2012.

[KC09] Christian Koehler and Dave Clarke. Decomposing port automata. In Sung Y. Shin and Sascha Ossowski, editors, *SAC*, pages 1369–1373. ACM, 2009.

[KLWZ09] Wei Ke, Zhiming Liu, Shuling Wang, and Liang Zhao. A graph-based operational semantics of OO programs. In *Proceedings of 11th International Conference on Formal Engineering Methods,* volume 5885 of *Lecture Notes in Computer Science*, pages 347–366, Berlin, 2009. Springer.

[Kra11] Christian Krause. Distributed port automata. *ECEASST*, 41, 2011.

[Lam94] L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, 1994.

[Lam02] L. Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers.* Addison-Wesley, 2002.

[LH06] Jing Liu and Jifeng He. Reactive component based service-oriented design-a case study. page 10 pp., 2006.

[LJ99] Zhiming Liu and Mathai Joseph. Specification and verification of fault-tolerance, timing, and scheduling. *ACM Transactions on Programming Languages and Systems*, 21(1):46–89, 1999.

[LNW06] Kim Guldstrand Larsen, Ulrik Nyman, and Andrzej Wasowski. Interface input/output automata. In Jayadev Misra, Tobias Nipkow, and Emil Sekerinski, editors, *FM*, volume 4085 of *LNCS*, pages 82–97. Springer, 2006.

[LNW07a] Kim Guldstrand Larsen, Ulrik Nyman, and Andrzej Wasowski. Modal I/O automata for interface and product line theories. In Rocco De Nicola, editor, *ESOP*, volume 4421 of *LNCS*, pages 64–79. Springer, 2007.

[LNW07b] Kim Guldstrand Larsen, Ulrik Nyman, and Andrzej Wasowski. On modal refinement and consistency. In Luís Caires and Vasco Thudichum Vasconcelos, editors, *CONCUR*, volume 4703 of *LNCS*, pages 105–119. Springer, 2007.

[LQL$^+$05] Quan Long, Zongyan Qiu, Zhiming Liu, Lingshuang Shao, and He Jifeng. Post: A case study for an incremental development in rcos. pages 485–500, 2005. UNU-IIST TR 324.

[LT87] Nancy A Lynch and Mark R Tuttle. *Hierarchical Correctness Proofs for Distributed Algorithms.* PhD thesis, PODC, 1987.

[LT89] Nancy A. Lynch and Mark R. Tuttle. An introduction to input/output automata. *CWI Quarterly*, 2(3):219–246, 1989.

[LV12] Gerald Luttgen and Walter Vogler. Modal interface automata. In *TCS'12: Proceedings of the 7th IFIP TC 1/WG 202 international conference on Theoretical Computer Science.* Springer-Verlag, September 2012.

[LW07] Kung-Kiu Lau and Zheng Wang. Software component models. *IEEE Trans. Software Eng*, 33(10):709–724, 2007.

[Mil95] Robin Milner. Communication and concurrency. *Communication and concurrency*, December 1995.

[MT00] Nenad Medvidovic and Richard N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Trans. Softw. Eng.*, 26(1):70–93, January 2000.

[Obj01] Object Managment Group. Model driven architecture - a technical perspective. Document number ORMSC 2001-07-01, 2001.

[Pie02] Benjamin C. Pierce. *Types and Programming Languages.* MIT Press, Cambridge, MA, USA, 2002.

[PV02] Frantisek Plasil and Stanislav Visnovsky. Behavior protocols for software components. *IEEE Trans. Softw. Eng.*, 28(11):1056–1076, November 2002.

[RBB$^+$09] J.B. Raclet, E. Badouel, A. Benveniste, B. Caillaud, A. Legay, and R. Passerone. Modal interfaces: unifying interface automata and modal specifications. In *Proceedings of the seventh ACM international conference on Embedded software*, pages 87–96. ACM, 2009.

[RBB$^+$11] Jean-Baptiste Raclet, Eric Badouel, Albert Benveniste, Benoît Caillaud, Axel Legay, and Roberto Passerone. A modal interface theory for component-based design. *Fundam. Inf.*, 108(1-2):119–149, January 2011.

[Ros98] A.W. Roscoe. *The Theory and Practice of Concurrency.* Prentice Hall, 1998.

[SG96] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.

[Sif05] J Sifakis. A framework for component-based construction. In *Third IEEE International Conference on Software Engineering and Formal Methods (SEFM'05)*, pages 293–299. IEEE, 2005.

[Szy02] Clemens Szyperski. *Component software: beyond object-oriented programming.* Pearson Education, 2002.

[Vaa91] Frits W. Vaandrager. On the relationship between process algebra and input/output automata. In *LICS*, pages 387–398, 1991.

[WBHR08] Martin Wirsing, Jean-Pierre Banâtre, Matthias M. Hölzl, and Axel Rauschmayer, editors. *Software-Intensive Systems and New Computing Paradigms - Challenges and Visions*, volume 5380 of *Lecture Notes in Computer Science*. Springer, New York, 2008.

[WC02] Jim Woodcock and Ana Cavalcanti. The semantics of circus. pages 184–203, 2002.

[XLD10] Xijiao Xiong, Jing Liu, and Zuohua Ding. Design and verification of a trustable medical system. *Electr. Notes Theor. Comput. Sci.*, 266:77–92, 2010.

[YS97] Daniel M Yellin and Robert E Strom. Protocol specifications and component adaptors. *Transactions on Programming Languages and Systems (TOPLAS*, 19(2), March 1997.

[ZLLQ09] Liang Zhao, Xiaojian Liu, Zhiming Liu, and Zongyan Qiu. Graph transformations for object-oriented refinement. *Formal Aspects of Computing*, 21(1–2):103–131, February 2009.