rCOS: Defining Meanings of Component-Based Software Architectures

Ruzhen Dong[1,2], Johannes Faber[1], Wei Ke[3], and Zhiming Liu[1]

[1] United Nations University – International Institute for Software Technology, Macau
{ruzhen,jfaber,z.liu}@iist.unu.edu
[2] Dipartmento di Informatica, Università di Pisa, Italy
[3] Macao Polytechnic Institute, Macau
wke@ipm.edu.mo

Abstract. Model-Driven Software Development is nowadays taken as a mainstream methodology. In the software engineering community, it is a synonym of the OMG *Model-Driven Architecture* (MDA). However, in the formal method community, model-driven development is broadly seen as model-based techniques for software design and verification. Because of the difference between the nature of research and practical model-driven software engineering, there is a gap between formal techniques, together with their tools, and their potential support to practical software development. In order to bridge this gap, we define the meanings of component-based software architectures in this chapter, and show how software architectures are formally modeled in the formal model-driven engineering method rCOS. With the semantics of software architecture components, their compositions and refinements, we demonstrate how appropriate formal techniques and their tools can be applied in an MDA development process.

Keywords Component-Based Architecture, Object-Oriented Design, Model, Model Refinement, Model Transformation, Verification.

Table of Contents

1 Introduction . . . 3
1.1 Software Complexity . . . 3
1.2 Model-Driven Development . . . 4
1.3 Formal Methods in Software Development . . . 5
1.4 The Aim and Theme of rCOS . . . 6
1.5 Organization . . . 7
2 Unified Semantics of Sequential Programming . . . 8
2.1 Designs of Sequential Programs . . . 8
2.2 Designs of Object-Oriented Programs . . . 12
2.3 Reactive Systems and Reactive Designs . . . 19
3 Model of Primitive Closed Components . . . 25
3.1 Specification Notation for Primitive Closed Components . . . 25
3.2 Labeled Transition Systems of Primitive Closed Components . . . 27
3.3 Component Contracts and Publications . . . 30
3.4 Refinement between Closed Components . . . 32
3.5 Separation of Concerns . . . 34
4 Primitive Open Components . . . 35
4.1 Specification of Open Components . . . 35
4.2 Semantics and Refinement of Open Components . . . 36
4.3 Transition Systems and Publications of Open Components . . . 38
5 Processes . . . 40
5.1 Specification of Processes . . . 40
5.2 Contracts of Processes . . . 41
5.3 Transition Systems and Publications of Processes . . . 42
6 Architectural Compositions and General Components . . . 43
6.1 Coordination of Components by Processes . . . 43
6.2 Composition of Processes . . . 45
6.3 Parallel Composition of Components . . . 46
6.4 Renaming and Restriction . . . 47
6.5 More Examples . . . 49
7 Interface Model of Components . . . 51
7.1 Component Automata . . . 52
7.2 Non-blockable Provided Events and Traces . . . 54
7.3 Interface Publication Automata . . . 56
7.4 Composition and Refinement . . . 58
8 Conclusions . . . 62

1 Introduction

Software engineering was born and has been growing up with the "software crisis". The root of the crisis is the inherent complexity of software development, and the major cause of the complexity "is that the machines have become several orders of magnitude more powerful" [18] within decades. Furthermore, ICT systems with machines and smart devices that are communicating through heterogeneous Internet and communication networks, considering integrated healthcare information systems and environment monitoring and control systems, are becoming more complex beyond the imagination of the computer scientists and software engineers in the 1980's.

1.1 Software Complexity

Software complexity was characterized before the middle of the 1990s in terms of four fundamental attributes of software [5–7]:

1. the *complexity of the domain application*,
2. the *difficulty of managing the development process*,
3. the *flexibility possible* through software,
4. and the *problem of characterizing the behavior* of software systems [5].

This characterization remains sound, but the extensions of the four attributes are becoming much wider.

The first attribute focuses on the difficulty of understanding the application domain (by the software designer in particular), capturing and handling the ever-changing requirements. It is even more challenging when networked systems support collaborative workflows involving many different kinds of stakeholders and end users across different domains. Typical cases are in healthcare applications, such as telemedicine, where chronic conditions of patients on homecare plans are monitored and tracked by different healthcare providers. In these systems, requirements for safety, privacy assurances and security are profound too.

The second attribute concerns the difficulty to define and manage a development process that has to deal with changing requirements for a software project involving a large team comprising of software engineers and domain experts, possibly in different geographical places. There is a need of a defined development process with tools that support collaboration of the team in working on shared software artifacts.

The third is about the problem of making the right design decisions among a wide range of possibilities that have conflicting features. This includes the design or reuse of the software architecture, algorithms and communication networks and protocols. The design decisions have to deal with changing requirements and aiming to achieve the optimal performance to best support the requirements of different users.

The final attribute of software complexity pinpoints the difficulty in understanding and modeling the semantic behavior of software, for analysis, validation

and verification for correctness, as well as reliability assurance. The semantic behavior of modern *software-intensive systems* [63], which we see in our everyday life, such as in transportation, health, banking and enterprise applications, has a great scale of complexity. These systems provide their users with a large variety of *services* and *features*. They are becoming increasingly *distributed*, *dynamic* and *mobile*. Their components are *deployed* over large networks of *heterogeneous platforms*. In addition to the complexity of functional structures and behaviors, modern software systems have complex aspects concerning *organizational structures* (i.e., *system topology*), *adaptability*, *interactions*, *security*, *real-time* and *fault-tolerance*. Thus, the availability of models for system architecture components, their interfaces, and compositions is crucially important.

Complex systems are open to total breakdowns [53], and consequences of system breakdowns are sometimes catastrophic and very costly, e.g., the famous Therac-25 Accident 1985-1987 [41], the Ariane-5 Explosion in 1996 [56], and the Wenzhou High Speed Train Collision.[4] Also the software complexity attributes are the main source of *unpredictability* of software projects, software projects fail due to our failure to master the complexity [33]. Given that the global economy as well as our everyday life depends on software systems, we cannot give up advancing the theories and the engineering methods to master the increasing complexity of software development.

1.2 Model-Driven Development

The *model-driven architecture* (MDA) [52, 60, 63] approach proposes building of system models in all stages of the system development as the key engineering principle for mastering software complexity and improving dependability and predictability. The notion of software architectures is emphasized in this approach, but it has not been precisely defined. In industrial project development, the architecture of a system at a level of abstraction is often represented by diagrams with "boxes" and "links" to show parts of the systems and their linkages, and sometimes these boxes are organized into a number of "layers" for a "layered architecture". There even used to be little informal semantic meaning for the boxes and links. This situation has improved since the introduction of the Unified Modeling Language (UML) in which boxes are defined as "objects" or "components", and links are defined as "associations" or "interfaces". These architectural models are defined as both *platform independent models* (PIM) and *platform specific models* (PSM), and the mapping from a PIM to a PSM is characterized as a *deployment model*.

MDA promotes in software engineering the principles of *divide and conquer* by which an architectural component is hierarchical and can be divided into subcomponents; *separation of concerns* that allows a component to be described by models of different viewpoints, such as static component and class views, interaction views and dynamic behavioral views; and *information hiding by ab-*

[4] `http://en.wikipedia.org/wiki/Wenzhou_train_collision`

stractions so that software models at different stages of development only focus on the details relevant to the problem being solved at that stage.

All the different architectural models and models of viewpoints are important when defining and managing a development process [43]. However, the semantics of these models is largely left to the user to understand, and the integration and transformation of these models are mostly syntax-based. Hence, the tools developed to support the integration and transformation cannot be integrated with tools for verification of semantic correctness and correctness preserving transformations [46].

For MDA to support a seamless development process of model decomposition, integration, and refinement, there is a need of formal semantic relations between models of different viewpoints of the architecture at a certain level, and the refinement/abstraction relation between models at different levels of abstraction. It is often the case that a software project in MDA only focuses on the grand levels of abstraction — *requirements*, *design*, *implementation* and *deployment*, without effectively supporting refinement of the requirements and design models, except for some model transformations based on design patterns. This is actually the reason why MDA has failed to demonstrate the potential advantages of *separation of concerns*, *divide and conquer* and *incremental development* that it promises. This lack of semantic relations between models as well as the lack of techniques and tools for semantics-preserving model transformations is also an essential barrier for MDA to realize its full potential in improving safety and predictability of software systems.

1.3 Formal Methods in Software Development

Ensuring semantic correctness of computer systems is the main purpose of using formal methods. A *formal method* consists of a body of techniques and tools for the *specification*, *development*, and *verification* of programs of a certain paradigm, such as sequential or object-oriented procedural programs, concurrent and distributed programs and now web-services. Here, a specification can be a description of an abstract model of the program or the specification of desirable properties of the program in a formally defined notation. In the former case, the specification notation is also often called a modeling language, though a modeling language usually includes graphic features. Well-known modeling/specification languages include CSP [28], CCS [50], the Z-Notation [58], the B-Method [1, 2], VDM [34], UNITY [9], and TLA+ [38]. In the latter case, i.e., the specification of program properties, these desirable properties are defined on a computational model of the executions of the system, such as state machines or transition systems. Well-known models of this kind include the *labeled transition systems* and the *linear temporal logic* (LTL) of Manna and Pnueli [49], which are also used in verification tools like Spin [31] and, in an extended form, Uppaal.[5]

The techniques and tools of a formal method are developed based on a mathematical theory of the execution or the behavior of programs. Therefore, we define

[5] `http://www.uppaal.org`

a *formal method* to include a *semantic theory* as well as the *techniques* and *tool support* underpinned by the theory for *modeling*, *design*, *analysis*, and *verification* of programs of a defined programming paradigm. It is important to note that the semantic theory of a formal method is developed based on the fundamental theories of *denotational semantics* [59], *operational semantics* [54], and *axiomatic semantics* (including algebraic semantics) [17,27] of programming. As they are all used to define and reason about behavior of programs, they are closely related [51], and indeed, they can be "unified" [29].

In the past half a century or so, a rich body of formal theories and techniques have been developed. They have made significant contribution to program behavior characterization and understanding, and recently there has been a growing effort in development of tool support for verification and reasoning. However, these techniques and tools, of which each has its community of researchers, have been mostly focusing on models of individual viewpoints. For examples, type systems are used for data structures, Hoare Logic for local functionality, process calculi (e.g., CSP and CSS) and I/O automata [48] for interaction and synchronization protocols. While process calculi and I/O automata are similar from the perspective of describing the interaction behavior of concurrent and distributed components, the former is based on the observation of the global behavior of interaction sequences, and the latter on the observation of local state transitions caused by interaction events. Processes calculi emphasize on support of algebraic reasoning, and automata are primarily used for algorithmic verification techniques, i.e., model checking [15, 55].

All realistic software projects have design concerns on all viewpoints of data structures, local sequential functionalities, and interactions. The experience, e.g., in [32], and investigation reports on software failures, such as those of the Therac-25 Accident in 1985–1987 [41] and the Ariane-5 Explosion in 1996 [56], show that the cause of a simple bug that can lead to catastrophic consequences and that *ad hoc* application of formal specification and verification to programs or to models of programs will not be enough or feasible to detect and remove these causes. Different formal techniques that deal with different concerns more effectively have to be systematically and consistently used in all stages of a development process, along with safety analysis that identifies risks, vulnerabilities, and consequences of possible risk incidents. There are applications that have extra concerns of design and verification, such as real-time and security constraints. Studies show that models with these extra functionalities can be mostly treated by model transformations into models for requirements without these concerns [44].

1.4 The Aim and Theme of rCOS

The aim of the rCOS method is to bridge the gap sketched in the previous sections 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.

1.5 Organization

Following this introduction section, we lay down the semantic foundation in Sect. 2 by developing a general model of *labeled transition systems* that combines the local computation (including structures and objects) in a transition and the dynamic behavior of interactions. We propose a *failure-divergence semantics* and a *failure-divergence refinement* relation between transition systems following the techniques of CSP [57]. Then in Sect. 3-5, we introduce the specification of *primitive closed components*, *primitive open components* and *processes* that are the basic architectural components in rCOS. Each of the different kinds of components is defined by their corresponding label transition systems. Models at different levels of abstraction, including *contracts* and *publications* for different purposes in a model-driven development process, are defined and their relations are studied. Section 6 defines the architectural operators for composing and adapting components. These operations on component specifications show how internal autonomous transitions are introduced and how they cause nondeterminism that we characterized in the general labeled transition systems. These operators extend and generalize the limited plugging, disjoint union and gluing operators we defined in the original components. The model also unifies the semantics and compositions of components and processes. A *general* component can exhibit both passive behavior of receiving service requests and actively invoke services from the environment. This is a major extension, but it preserves the results that we have developed for the original rCOS model. However, this extension still needs more detailed investigation in the future, including their algebraic properties. Section 7 is about a piece of work on an interface model of rCOS components. The aim is to propose an input-deterministic model of component interfaces for better composability checking, and to give a more direct description of provided and required protocols of components. There, we define a partial order, called *alternative refinement*, among component interface models. The results are still preliminary, and interesting further research topics are thus pointed out. Concluding remarks are given and future work is discussed in Sect. 8.

2 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*. Refinement of components is also built on a refinement calculus of object-oriented programs.

2.1 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,

$$c ::= x := e \mid c; c \mid c \lhd b \rhd c \mid c \sqcap c \mid b * c \tag{1}$$

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. We follow UTP [29] to define the semantics of programs in the above simple syntax as relations between the initial and final states.

States We assume an infinite set of names $\mathcal{X}$ representing *state variables* with an associated value space $\mathcal{V}$. We define a *state* of $\mathcal{X}$ as a function $s : \mathcal{X} \to \mathcal{V}$ and use Σ 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

$$R_1; R_2 \mathrel{\widehat{=}} \exists x_0 \bullet R_1(x_0/x') \wedge R_2(x_0/x), \tag{2}$$

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 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 (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.

$$\begin{aligned} x := e &\mathrel{\widehat{=}} \mathit{true} \vdash x' = e \wedge \textstyle\bigwedge_{y \in \alpha, y \neq x} y' = y, \\ c_1; c_2 &\mathrel{\widehat{=}} c_1; c_2 \\ c_1 \lhd b \rhd c_2 &\mathrel{\widehat{=}} b \wedge c_1 \vee \neg b \wedge c_2, \\ c_1 \sqcap c_2 &\mathrel{\widehat{=}} c_1 \vee c_2, \\ b * c &\mathrel{\widehat{=}} (c; b * c) \lhd b \rhd \mathbf{skip}, \end{aligned} \tag{3}$$

where we have $\mathbf{skip} \mathrel{\widehat{=}} \mathit{true} \vdash \bigwedge_{x \in \alpha}(x' = x)$. We also define $\mathbf{chaos} \mathrel{\widehat{=}} \mathit{false} \vdash \mathit{true}$. 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 (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 (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 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. *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 (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

$$D; D_h \sqsubseteq D; D_l, \qquad D_h; D \sqsubseteq D_l; D,$$
$$D_h \lhd b \rhd D \sqsubseteq D_l \lhd b \rhd D, \quad D_h \sqcap D \sqsubseteq D_l \sqcap D,$$

4. $(\mathcal{D}, \sqsubseteq)$ *forms a CPO and the recursive equation* $X = (D; X) \lhd b \rhd \mathbf{skip}$ *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 [29]. 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 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 1 and 2 both hold. For details of UTP, we refer to the book [29]. 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 [27] and the predicate transformer semantics of Dijkstra [17]. 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.

2.2 Designs of Object-Oriented Programs

We emphasize the importance of a semantic theory for concept clarification, development of techniques and tool support for *correct by design* and verification. The semantic theory presented in the previous section needs to be extended to define the concepts of classes, objects, methods, and OO program execution. The execution of an OO program is more complex than that of a traditional sequential program because the execution states have complex structures and properties. The semantics of OO programs has to cohesively define and treat

- the concepts of object heaps, stacks and stores,
- the problems of *aliasing*,
- subtyping and polymorphism introduced through the class inheritance mechanism, and
- dynamic typing of expression evaluation and dynamic binding of method invocation.

Without an appropriate definition of the execution state, the classic Hoare-logic cannot be used to specify OO program executions. Consider two classes C_1 and C_2 such that C_1 is a subclass of C_2 (denoted by $C_1 \preceq C_2$), and variables $C_1\ x_1$ and $C_2\ x_2$ of the classes, respectively. Assume a is an attribute of C_2 and thus also an attribute of C_1, the following Hoare-triple (confer previous section for representing Hoare-triples as designs) holds when x_1 and x_2 do not refer to the same object, i.e., they are not aliases of the same object, but does not necessarily hold if they refer to the same object:

$$\{x_2.a = 4\}\ x_1.a := 3\ \{x_2.a = 4\}.$$

If inheritance allows *attribute hiding* in the sense that the attribute a of C_2 can be redeclared in its subclass C_1, even the following Hoare-triple does not generally hold:

$$\{x_1.a = 3\}\ x_2 := x_1\ \{x_2.a = 3\}.$$

Therefore, the following fundamental *backward substitution rule* does not generally hold for OO programs:

$$\{Q[e/le]\}\ le := e\ \{Q\}.$$

In order to allow the use of OO design and programming for component-based software development, rCOS extends the theory of designs in UTP to a theory of OO designs. The theory includes an UTP-based denotational semantics [26, 66], a graph-based operational semantics of OO programs [36] and a refinement calculus [66] of OO designs. We only give a summary of the main ideas and we refer to the publications for technical details, which are of less interest for general readers.

OO specification The rCOS OO specification language is defined in [26]. Similar to Java, an OO program P consists of a list *ClassDecls* of class declarations and a main program body *Main*. Each class in *ClassDecls* is of the form:

$$\begin{array}{ll}\textbf{class}\ M\ [\textbf{extends}\ N] & \\ \quad \textbf{private} & T_{11}\ a_{11} = d_{11}, \ldots, T_{1n_1}\ a_{1n_1} = d_{1n_1}; \\ \quad \textbf{protected} & T_{21}\ a_{21} = d_{21}, \ldots, T_{2n_2}\ a_{2n_2} = d_{2n_2}; \\ \quad \textbf{public} & T_{31}\ a_{31} = d_{31}, \ldots, T_{3n_3}\ a_{3n_3} = d_{3n_3}; \\ \quad \textbf{method} & m_1\ (T_{11}\ x_1; T_{12}\ y_1)\ \{c_1\} \\ & \cdots \\ & m_\ell\ (T_{\ell 1}\ x_\ell; T_{\ell 2}\ y_\ell)\ \{c_\ell\}\end{array}$$

Therefore, a class can declare at most one direct superclass using **extends**, some attributes with their types and initial values, and methods with their signatures and body commands. Types include classes and a set of assumed primitive data types such as integers, booleans, characters and strings. The scopes of *visibility* of the attributes are defined by the **private**, **protected** and **public** keywords. We could also have different scopes of visibility for the methods, but we assume all methods are public for simplicity. A method can have a list of input parameters and return parameters with their types. We use return parameters, instead of return types of methods to a) avoid the use of method invocations in expressions so that evaluation of expressions have no side effect, and b) give us the flexibility in specifications that a method can have a number of return values.

The main program body *Main* declares a list *vars* of variables, called the *global variables* with their types and initial values, and a command c. We can thus denote the main program body as a pair $(vars, c)$ in our discussion. One can view the main program body as a class *Main*:

$$\textbf{class}\ Main\ \{\ \textbf{private}\ vars;\ \textbf{method}\ main()\{c\}\ \}$$

A command in a method, including the *main* method, is written in the following syntax:

$$
\begin{array}{ll}
\text{expressions:} & e ::= x \mid \mathit{null} \mid \mathit{this} \mid e.a \mid (C)e \mid f(e) \\
\text{assignable expressions:} & \mathit{le} ::= x \mid e.a \\
\text{commands:} & c ::= \mathbf{skip} \mid \mathbf{chaos} \mid \mathbf{var}\ T\ x = e;\ c;\ \mathbf{end}\ x \mid \\
 & \qquad c; c \mid c \lhd b \rhd c \mid c \sqcap c \mid b * c \mid \\
 & \qquad e.m(e^*; \mathit{le}) \mid \mathit{le} := e \mid C.\mathit{new}(\mathit{le})
\end{array}
$$

Here, x is a basic type or an object variable and $e.a$ an attribute of e. For the sake of clarity, a simplified presentation for method parameters and variable scope is used; we generally allow lists of expressions as method parameters and lists of variable declarations for the scope operator **var**. Notice that the creation of a new object $C.\mathit{new}(\mathit{le})$ is a command not an expression. It returns in le the object newly created and plays the same role as $\mathit{le} = \mathit{new}\ C()$ in Java or C++.

Objects, Types and States An *object* has an identity, a state and a behavior. We use a designated set *REF* to represent object identities. An object also has a runtime type. Thus, we define an object by a triple $o = (r, C, s)$ of its identity r, runtime type C and state s. The state s is a *typed function*

$$s : \mathcal{A}(C) \to \mathcal{O} \cup \mathcal{V},$$

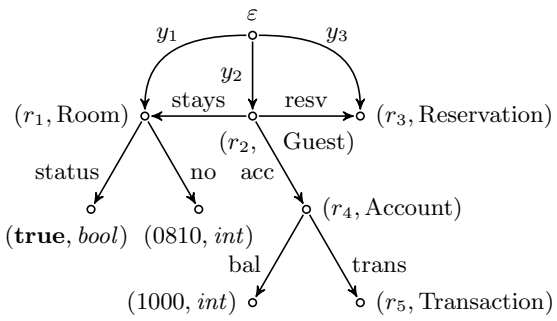
where

- $\mathcal{O}$ is the set of all objects of all classes,
- $\mathcal{V}$ the value space of all the primitive data types,
- $\mathcal{A}(C)$ is the set of names of the attributes of C, including those inherited from all its superclasses, and
- s maps an attribute a to an object or value of the type of a declared in C.

Therefore, an object o has a recursive structure, and can be represented by a rooted-labeled-directed graph, called an *object graph* [36, 66], in which

- the root represents the object labeled by its runtime type,
- each outgoing edge is labeled by an attribute of the object and leads to a node that is either an object or a value, and
- each object node is the root of a subgraph representing that object.

In an object graph, all value nodes are leaves. An object graph can also be represented by a UML object diagram [66], but UML object diagrams do not have the properties of the mathematical structure of rooted-labeled-directed graphs needed for formal reasoning and analysis. Furthermore, the types in an object graph together with the labels for attributes form a *class graph* that is called the *type graph* of the object that the object graph represents [36, 66].




```
1  program Hotel {
2    class Person {
3      public Account acc = null;
4      method m1(){...}
5    }
6    class Guest extends Person {
7      public
8        Room stays = null,
9        Reservation resv = null;
10     method
11       makeReservation() {...}
12       checkOut() {...}
13   }
14   class Account {
15     public
16       int bal = 0;
17       Transaction trans = null;
18     method checkBalance() {...}
19   }
20   class Room {
21     public
22       int no = 0,
23       bool status = false;
24     method changeStatus() {...}
25   }
26   class Transaction {
27     ...
28   }
29   class Main {
30     private
31       Room y1,
32       Guest y2,
33       Reservation y3;
34     method main() {...}
35   }
36 } // end of program
```

Fig. 1. An example of object graph

States of Programs Given an OO program $P = \mathit{ClassDecls} \bullet \mathit{Main}$, a *global state* of P is defined as a mapping $s : \mathit{vars} \to \mathcal{O} \cup \mathcal{V}$ that assigns each variable $x \in \mathit{vars}$ an object or a value depending on the type of x. Taking *Main* as a class, a global state of P is thus an object of *Main* and can be represented as an object graph, called a *global state graph*. During the execution of the main method, the identity of the object representing the state will never be changed, but its state will be modified in each step of the execution. All the global state graphs have the same type graph. The type graph of the program can be statically defined from the class declarations *ClassDecls*. Its UML counterpart is the UML class diagram of the program in which classes have no methods. For example, Fig. 1 is a global state of the accompanied program outline, and its type graph (and the corresponding UML class diagram) is given in Fig. 2.

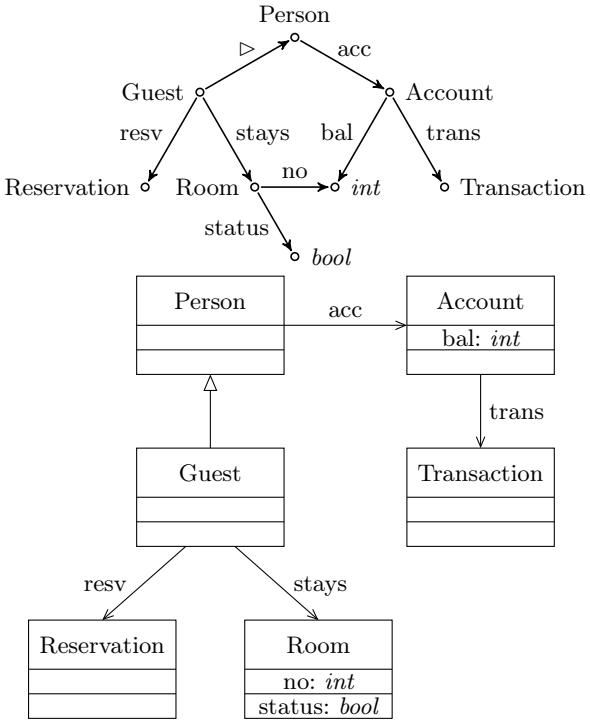


Fig. 2. An example of class graph and diagram

Global states are enough for defining a UTP-based denotational semantics [26] and a "big step semantics" of the program in which executions of intermediate execution steps and the change of locally declared variables are hidden. To define a small step operational semantics, we need to represent the *stack* of local variable declarations to characterize the execution of **var** $T\ x = x_0$, where T can either be a class or a data type, and x_0 is the declared initial value of x. For this, we extend the notation of global state graphs by introducing edges labeled by a designated symbol `$`. The execution of **var** $T\ x = x_0$ from a state graph G adds a new root node n' to G that has an outgoing edge to the root n of G, labeled by `$`, and another outgoing edge to x_0, labeled by x. We can understand this as *pushing* a new node on top of G with one outgoing edge labeled by `$` to the root of G and another labeled by x to its initial value. Such a *state graph* contains a `$`-path of *scope nodes*, called the *stack*. Executing the command **end** x from such a state graph pops out the root together with its outgoing edges. Figure 3 shows an example of a state graph that characterizes the local scopes below:

$$\textbf{var}\ C_2\ z = o_2, C_3\ x = o_3; \textbf{var}\ C_2\ x = o_2; \textbf{var}\ int\ z = 3, C_1\ y = o_1$$

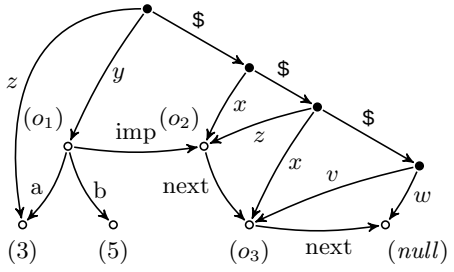


Fig. 3. An example of state graph with local scopes

where o_1, o_2 and o_3 are objects of type C_1, C_2, and C_3 referred to by the variables y, z and x in their scopes, respectively.

Semantics We explain the semantics informally. Both a denotational semantics and an operational semantics can be defined by specifying which changes the execution of a command makes on a given state graph. This can be understood with our graph representation of states. Given an initial state graph G

- *assignment*: $le.a := e$ first evaluates e on G as a node n' of G and then swings the a-edge of the target node of le in G to the node n';
- *object-creation*: $C.new(le.a)$ makes an object graph of C according to the initial values of its attributes, such that the root n' is not in G, and then swings the a-edge of the target node of le in G to the node n';
- *method invocation*: $e.m(e_1; le.a)$ first records e, e_1 and le to *this*, the formal value parameter of $m()$ and y^+, respectively, then executes the body of $m()$, returns the value of the formal return parameter of $m()$ to the actual return parameter $y^+.a$ which is the initial $le.a$ that might be changed by the execution, roughly that is

  var $this = e, in = e_1, y^+ = le, return$;
  $c; y^+.a := return$;
  end $this, in, y^+, return$

  where *in* and *return* are the formal parameters of $m()$.

Then conditional choice, non-deterministic choice and iterative statements can be defined inductively. For a denotational semantics, a partial order has to be defined with that a unique fixed-point of the iterative statements and recursive method invocations can be defined. The theory of denotational semantics is presented in [26] and the graph-based operational semantics is given in [36].

OO Refinement OO refinement is studied at three levels in rCOS: refinement between whole programs, refinement between class declarations (called *structure refinement*), and refinement between commands. The refinement relation

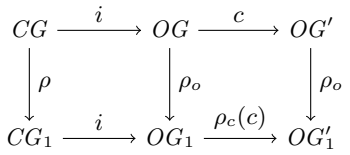


Fig. 4. Class graph transformations and command transformations

between commands takes exactly the same view as in the previous section about traditional sequential programs, where the execution of a program command is a relation between states. A command c_l is a refinement of a command c_h, denoted by $c_h \sqsubseteq c_l$, if for any given initial state graph G, any possible final state G' of c_l is also a possible final state of c_h. This definition takes care of non-determinism and a refined command is not more non-deterministic than the original command. However, refinement between commands in OO programming only makes sense under the context of a given list of class declarations *ClassDecls*. Under such a given context, some variables and method invocations in a command c might not be defined. In this case, we treat the command to be equivalent to **chaos**, which can be refined by any command under the same context. To compare two commands under different class contexts, we use the extended notation of data refinement and relate the context of c_l to that of c_h by a *class (graph) transformation*.

The combination of class graph transformations and command transformations is illustrated in Fig. 4. It shows that given a class graph transformation ρ from CG to CG_1, we can derive a transformation ρ_o from an instance object graph OG of CG to an instance object graph OG_1 of CG_1, as well as a transformation ρ_c on commands. Then ρ is a class refinement if the diagram commutes for all OG of CG and all commands c.

Definition 4 (OO program refinement). *A program P_l = $ClassDecls_l \bullet Main_l$ is a refinement of a program P_h = $ClassDecls_h \bullet Main_h$, if there is a class graph transformation from the class graph of P_l to that of P_h such that the command of $Main_l$ is a refinement of the command of $Main_h$.*

In the paper [66], we give a systematic study of the combination of class refinement and command refinement, and develop a graph-based OO refinement calculus. It gives a full formalization of OO program refactoring [23] by a group of simple rules of class graph transformations, including adding classes, attributes, methods, decomposition and composition of classes, promoting methods from subclasses to super classes, from private to protected and then to public. The combination of class graph transformations with command transformations formalizes the design patterns for class responsibility assignments for object-oriented design, including in particular the *expert patterns*, *low coupling* and *high cohesion* patterns [39]. The use of these patterns is an essential practice in OO program design [12].

An essential advantage of OO programming is that classes can be reused in different applications that are implemented in different main methods. Classes can also be extended for application evolution. The classes of an application program are in fact the metamodel of the structure or organization of the application domain in terms of concepts, objects, their relations, and behavior. On the other hand, the main method of the program is the automation of the application business processes, i.e., *use cases*, via the control of the objects' behavior. Of course, different structures provide different functionalities and thus different use cases, the same use case can also be supported by different structures. The structure refinement in rCOS characterizes this fundamental feature of OO programming.

Definition 5 (OO structure refinement). *A list $ClassDecls_l$ of class declarations is a refinement of a list $ClassDecls_h$ of class declarations if for any main method Main, $ClassDecls_h \bullet Main \sqsubseteq ClassDecls_l \bullet Main$.*

This means that a refined list of class declarations has more capacity in providing more and "better" services in the sense that the lower level class declarations may provide additional functionality or may provide more defined functionality with less non-determinism following the classical notion of refinement.

The refinement calculus is proved to be sound and relatively complete in the sense that the rules allow us to transform the class graph of a program to a tree of inheritance, and with the derived transformation rules on the main method, the program can be refined to an equivalent program that only has the main class. Thus each OO program can be transformed to an equivalent procedural program [66].

2.3 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

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

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 $\mathcal{H}(\mathcal{H}(D)) = \mathcal{H}(D)$.

Theorem 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

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

Theorem 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 $\mathcal{H}(p \vdash R) = (wait \vee p) \vdash (wait' \lhd wait \rhd R)$. 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 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:

$$c ::= x := e \mid c; c \mid c \lhd b \rhd c \mid c \Box c \mid g \;\&\; c \mid b * c \tag{4}$$

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

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

and for all other cases as defined in equation (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* [3], a UNITY program [9] and a TLA specification [37].

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

$$S = \langle var, init, \Omega, \Lambda \rangle,$$

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 $var \cup \{ok, wait\}$,*
- *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 (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 7 (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 8 (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*

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

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) $\sigma = s_0 \xrightarrow{a_1} \cdots \xrightarrow{a_k} s_k$ *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

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

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 9 (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 [37]).

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.

Theorem 6 (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 $guard_l(a) \iff guard_h(a)$,*
3. *for each $a \in \Omega_l$, $a \in \Omega_h$ and $next_h(a) \sqsubseteq next_l(a)$, and*
4. *$\neg g(\Omega_h) \implies (g(\Lambda_h) \iff g(\Lambda_l)) \wedge (inext(S_l) \implies inext(S_h))$, 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

$$\neg g(\Omega_h) \implies (g(\Lambda_h) \iff g(\Lambda_l))$$

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 [3, 9, 37, 44] and failure-divergence refinement of CSP [57]. 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.

3 Model of Primitive Closed Components

The aim of this chapter 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.

3.1 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 nota-

tion implemented in a tool, called *the rCOS modeler*.[6] 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[7] 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. 5 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.

Based on the theory of guarded designs presented in Sect. 2, 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.

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

[7] 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
}
```

Fig. 5. Format of rCOS closed components

3.2 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 defined in Sect. 2.3. 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 10 (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 11 (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 $\mathcal{T}(K)$ of K are also defined by the traces of the corresponding transition system: $\mathcal{T}(K) \mathrel{\widehat{=}} \mathcal{T}(\underline{K})$.

Example 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) }
   }
}
```

Relation to Traditional Theories of Programming 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 [38]. 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* [3] that extends Dijkstra's *guarded commands language* [17] to concurrent programming.
3. The model of components here is similar to I/O automata [48]. However, the guards of methods in general may depend on input parameters, as well as state variables of the component. This implies that a component may be an I/O automata with infinite number of states. The I/O automata used for verification, model checking in particular, are finite state automata and the states are abstract symbolic states. The guards of transitions are also encoded in the symbolic states such that in some states of an automaton, transitions are enabled or disabled.
4. 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 [28, 57]. 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 [61].
5. Other formalisms like, e.g. CSP-OZ [22, 30], also combine state and event-based interaction models in a similar way. These approaches and also similar combinations like Circus [64] 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* [8, 29] for *Separation of Concerns*, instead of extending a notation to increase expressive power.

3.3 Component Contracts and Publications

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

Definition 12 (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 the component M of Example 1 on page 28 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 1 and its contract, the state variable *start* is a protocol control variable.

Clearly, for each component 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, $\mathcal{T}(K)$ 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 13 (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* $pref = a_1 \cdots a_k$*, there does not exist a set* X *of provided events of* K *such that* $a_{k+1} \in X$ *and* $(pref, X) \in \mathcal{IF}(K)$*. 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* $\mathcal{PP}(\underline{K})$*).*

The notion of contract is a rather "operational" model in the sense that the behavior is defined through the enabledness of a method at a state and the transition to the next possible state. This model has its advantage in supporting model checking verification techniques. However, for such an operational model with its non-determinism it is not easy to support third party usage and composition. A more denotational or global behavioral model would be more

appropriate. Hence, we define the notion of protocols of components and publications of components below. From the behavioral semantics of a contract C defined by Definition 11, we obtain the following model interface behavior.

Definition 14 (Publication). *A* **component publication** B *is similar to a contract and consists of the following sections of declarations,*

- *its provided interface* $B.pIF$*,*
- *variables* $B.var$ *and initial states* $B.init$*,*
- *the data functionality specification* $m(T_1\ x; T_2\ y)\{p \vdash R\}$*, and*
- *the provided protocol* $B.pProt$ *that is a set of traces.*

A contract C can be transformed into a component publication by embedding the guards of methods into the protocol. That is, the component publication for C is obtained by using the set of non-blockable traces of $\underline{C}$ as provided protocol $\mathcal{PP}(C)$ and by removing the guards of interface methods. For the same reason, the state variables that are only used in the flow of interaction control, such as *start* in the memory component M in Example 1, can also be abstracted away from the publication. The protocol can be specified in a formal notation. This includes formal languages, such as regular expressions or a restricted version of process algebra such as CSP without hiding and internal choice. Publications are declarative, while contracts are operational. Thus, publications are suitable for specifying components in system synthesis.

Example 2. Both the methods W and R of the interface of M in Example 1 are deterministic. Thus, M is input-deterministic and we have

$$\mathcal{PP}(M) = \mathcal{T}(M) = ?W(\{?W, ?R\}^*)$$

Here we adopt the convention to use a question mark to prefix an input service request event, i.e., a method invocation, in order to differentiate it from a calling out event in the required interface of an *open component*, which we will define later. Also, we have omitted the actual parameters in the method invocations, and $?W$ for example represents all possible $?W(a)$ for $a \in Z$. Thus, the following specification is a publication of the memory component M of Example 1.

```
component M {
   provided interface MIF {
      Z d;
      W(Z v) { {d}: true ⊢ d' = v }
      R(; Z v) { {v}: true ⊢ v' = d }
      protocol { ?W({?W,?R})* }
   }
}
```

In the example, we used a regular expression to describe the provided protocol. However, regular expressions have limited expressive power and can only express

languages of finite state automata. Languages like CSP can be used for more general protocols.

For the rest of the chapter, we use the programming notation defined in Sect. 2 in place of the designs that define its semantics. We use the notion "component" also for a "contract" and a "publication", as they are specifications of components at different levels of abstractions and for different purposes.

3.4 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 6, we have the following theorem for the refinement relation between two primitive closed components.

Theorem 7. *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 $tr = e_1 \dots e_k e_{k+1}$ 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 $(e_1 \dots e_k, X) \in \mathcal{IF}(K_l)$. Because $K_h \sqsubseteq K_l$, $\mathcal{IF}(K_l) \subseteq \mathcal{IF}(K_h)$, thus $(e_1 \dots e_k, X) \in \mathcal{IF}(K_h)$. 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 7 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. For this reason, we use an extended version of the stable-failures semantics in Sect. 7

The semantic definition of refinement of components (or contracts) by Definition 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 8. *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 9. *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 10. *$C_h \sqsubseteq C_l$ 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 [10].

3.5 Separation of Concerns

A component can be modeled in the specification notation defined in Fig. 5 at different levels of abstraction, and the relation between different levels of abstraction is expressed by refinement. The semantics theory of guarded commands leads to the model of contracts, and a *failure-divergence* semantic model of components. Contracts serve as a requirement model of components that can directly be represented as automata or labeled transition systems. Contracts are also used for correct by construction of components with the refinement calculus.

From the model of contracts and their failure-divergence semantics, we derive the model of publications. The publication of a component eases the usage of a component, including the usage through the interface and the composition with other components to form new software components. For this, a publication has to be faithful with respect to the specification of the component by its contract.

Definition 15. *Let B be a publication and C be a contract such that $B.pIF = C.pIF$, $B.var = C.var$ and $B.init \subseteq C.init$. B is* **faithful** *to C if $B.pProt \subseteq \mathcal{PP}(C)$ and $body_C(m) \sqsubseteq body_B(m)$.*

The faithfulness of a publications states that each method provides the functionality as specified in the contract and all traces published are acceptable traces of the component specified by the contract. Now we have the following *theorem of separation of concerns*.

Theorem 11. *We can separate the analysis, design and verification of the data functionality and the interaction protocol of a component as follows.*

1. *If $B_1 = (pIF, X, X_0, \Psi, pProt_1)$ and $B_2 = (pIF, X, X_0, \Psi, pProt_2)$ are faithful to $C = (pIF, X, X_0, \Phi, \Gamma)$ then*

$$B = (pIF, X, X_0, \Psi, pProt_1 \cup pProt_2) \text{ is also faithful to } C.$$

2. *If $B_1 = (pIF, X, X_0, \Psi, pProt_1)$ is faithful to C and $pProt_2 \subseteq pProt_1$, then*

$$B_2 = (pIF, X, X_0, \Psi, pProt_2) \text{ is also faithful to } C.$$

3. *If $B_1 = (pIF, X, X_0, \Psi_1, pProt)$ is faithful to C and $\Psi_1 \sqsubseteq \Psi_2$, then*

$$B_2 = (pIF, X, X_0, \Psi_2, pProt) \text{ is also faithful to } C.$$

4. *If a publication B is faithful to C_1 and $C_1 \sqsubseteq C_2$, then B is faithful to C_2.*

Proof. The first three properties are easy to check, and the last property is a direct corollary of Theorem 7.

For a contract (or a component) $K = (pIF, X, X_0, \Psi, \Gamma)$, the *largest faithful publication* of K with respect to the refinement relation is $(pIF, X, X_0, \Psi, \mathcal{PP}(K))$.

```
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
}
```

Fig. 6. Format of rCOS primitive open components

4 Primitive 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.

4.1 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. 6.

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 3. If we "plug" the provided interface of the memory component M of Example 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)
  }
}
```

4.2 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 16 (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.*

To illustrate the semantics definition, we give the following example.

Example 4. With the memory component in Example 1, *Buff*(M) for *Buff* in Example 3 is equivalent to the contract of a one-place buffer whose publication can be specified as

```
component B {
  provided interface BuffIF {
    Z d;
    put(Z v) { d := v }
    get(; Z v) { v := d }
  }
  provided protocol {
    (?put ?get)*+(?put ?get)*?put // data parameters are omitted
  }
}
```

Definition 17. *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 12. *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 18 (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,*

- *the definition of an assignment is the same as it is defined in Sect. 2: $x := e = \{x\} : true \vdash x' = e$, 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 of Sect. 2.3.

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 19 (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 18.*

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 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 $\overline{K(wtc(K.rIF))}$.*

4.3 Transition Systems and Publications of Open Components

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 20 (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.

```
publication K {
   T x = c; // initial state of component
   provided interface I { // provided methods
      m1(parameters) { c1 /* unguarded design */ };
      m2(parameters) { c2 /* unguarded design */ };
      ...
      m(parameters) { c /* unguarded design */ };
   };
   internal interface {
      n1(parameters) { d1 /* unguarded design */ };
      n2(parameters) { d2 /* unguarded design */ };
      ...
      n(parameters) { d /* unguarded design */ };
   };
   required interface J { // required services
      T y = d;
      n1(parameters), n2(parameters), n3(parameters)
   };
   provided protocol {
      L1 // a regular language expression over provided method invocations
   }
   required protocol {
      L2 // a regular language expression over required method invocations
   }
   class C1{...}; class C2{...}; ... // used in the above specification
}
```

Fig. 7. Format of rCOS open publication

- For each trace $tr = a_1/E_1 \dots a_k/E_k$, we have a provided trace $tr^{>} = a_1 \dots a_k$ 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.

As shown in Fig. 7, the specification of a publication of an open component is similar to that of a closed component, except that the bodies of the methods

of the provided and internal interfaces are defined as commands without guards, and the specification is extended with provided and required protocols.

Example 5. The publication of the open component *Buff* of Example 3 can be specified as follows.

```
publication BuffP {
   provided interface BuffIF {
      put(Z v) { W(v) }
      get(; Z v) { R(; v) }
   };
   required interface BuffrIF { W(Z v), R(; Z v) };
   provided protocol { (?put ?get)*+(?put ?get)*?put };
   required protocol { (!W !R)*+(!W !R)*!W }
}
```

Note, unlike in a contract of a component where each transition step is represented atomically, in a publication an action a/E is executed as an atomic step of state transition, though E represents a set of traces of method invocations. However, the composability can be checked in the following way: If the provided protocol $K.pProt$ of a component K contains (accepts) all the invocation traces of the required protocol $\mathcal{RP}(J)$ of an open component J, then K can be plugged to provide the services that J requires.

5 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 3, 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 $get_1(; a); put_2(a)$ 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*.

5.1 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. 8. 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.

```
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
}
```

Fig. 8. Format of rCOS process specifications

5.2 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 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.

However, we apply the same trick as we did when defining the semantics in Definition 18 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.

```
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
}
```

Fig. 9. Format of rCOS process contracts

Definition 21 (Contract of process). *Given a specification of a process P in the form shown in Fig. 8, its* **contract** $\overline{P}$ *is defined analogously to Definition 19 by enrichment with history variables, i.e., it is specified as shown in Fig. 9.*

Example 6. 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' = {get1(;a) · put2(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, and from that, the publication of a process.

5.3 Transition Systems and Publications of Processes

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 22 (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 nondeterministically invoke methods of components, and its whole trace set is taken as the **required protocol**

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

The definition of the protocol of a process allows us to define publications of processes in the same way as we defined publications for open components.

6 Architectural Compositions and General Components

We have defined the models of primitive closed and open components as well as processes. These models do not show any entities resulting from compositions, but software components and processes that are about constructing components by compositions of these primitive forms. In this section, we go beyond the primitive component model and define architectural composition operations that allow us to build hierarchical software architectures. We will also define a general model of components in this way and discuss the special classes of components that are useful in building software components. For this, we reiterate the notations for the different sections in a component specification (or process) K, that are $K.var$, $K.init$, $K.pIF$, $K.rIF$, $K.iIF$ and $K.Act$. Component operations are syntactically defined as operations on these sections, and then their semantics definitions are derived. We provide examples to illustrate their meanings and uses, too.

6.1 Coordination of Components by Processes

One way of building larger components from existing components is to coordinate their behavior by active processes. We start with this composition as it introduces internal autonomous actions to components, by which the primitive component model defined in the previous section is extended.

Definition 23 (Coordination of components). *Let K be a component and P a process. The coordination of K by P, denoted by $K \parallel P$, is defined if K and P do not have common variables. When $K \parallel P$ is defined, it is the following* **general component***, denoted by J,*

$$\begin{aligned} &J.var = K.var \cup P.var, \quad J.init = K.init \times P.init,\\ &J.pIF = K.pIF, \qquad\quad J.rIF = K.rIF \cup P.rIF,\\ &J.iIF = K.iIF \cup P.iIF. \end{aligned}$$

Additionally, we extend the specification of components with the section

$$J.inva = P.Act$$

containing the set of actions that have the triggering events invisible to the environment and can be executed autonomously when enabled. A subset of P.Act contains those which have no external required events or external triggering events. These are the actions entirely internal in the component.

It is necessary to introduce the internal autonomous actions in the above definition, because internal actions emerge when a process is used to coordinate the behavior of a component.

Definition 24 (Transition system of coordination). *When $J = K \parallel P$ is defined, we define the general transition system $\underline{J} = \langle J.var, J.init, \Omega, \Lambda\rangle$ such that*

1. *$\Omega = K.ifa$, that is $\omega(K.pIF) \times 2^{\omega(K.rIF)^*}$,*
2. *$\Lambda = P.Act$,*
3. *$(s_1, s_2) \xrightarrow{e/E} (s_1', s_2')$ is a transition in $\underline{J}$ if*
   - *(component step) $s_2 = s_2'$, $e \in \omega(K.pIF)$ and $s_1 \xrightarrow{e/E} s_1'$ is a transition of $\underline{K}$, or*
   - *(process step) $e \in \Lambda$ and there exists $s_2 \xrightarrow{e/F} s_2'$ in $\underline{P}$ such that for every*

$$tr_0 \cdot m_1 \cdot tr_1 \cdots m_k \cdot tr_k \in F,$$

   where $m_i \in \omega(K.pIF)$ and $tr_j \in \omega(J.rIF)^*$, it holds
   - *there exist $E_1, \ldots, E_k$ such that $s_1 \xrightarrow{m_1/E_1,\ldots,m_k/E_k} s_1'$,*
   - *$tr_0 \cdot E_1 \cdots E_k \cdot tr_k \subseteq E$, and*
   - *E is the smallest set that satisfies these two properties.*

In rCOS, we specify a general component in the format shown in Fig. 10, that extends the specification of an open component in Fig. 6 with a section of invisible actions. Thus, K is a primitive open component if $K.inva$ is empty, a closed component when $K.rIF$ is empty, and a process when $K.pIF$ is empty. General components thus contain both active and passive behavior. From now on, a process is also treated as a component.

As shown in the definition of $K \parallel P$, actions defined in $J.inva$ may also require methods given in the required interface $J.rIF$. When an invisible action does not require any methods outside the component, it is then totally invisible.

```
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 */ };
   };
   actions { // invisible autonomous action
      a1() { f1 & e1 }; // no parameters
      a2() { f2 & e2 }; // no parameters
      ...
      a() { f & e }
   };
   required interface J { // required services
      T y = d;
      n1(parameters), n2(parameters), n3(parameters)
   };
   class C1{...}; class C2{...}; ... // used in the above specification
}
```

Fig. 10. Format of rCOS general components

6.2 Composition of Processes

Now we define the parallel composition of processes. Since processes only actively call external methods, but do not provide methods to be called, two processes do not communicate directly. Thus, the execution of the parallel composition of two processes is simply the interleaving execution of the actions of the individual processes.

Definition 25 (Parallel composition of processes). *For two processes P_1 and P_2, the parallel composition $P_1 \parallel P_2$ is defined if they have neither common variables nor common action names. When $P_1 \parallel P_2$ is defined, the composition, denoted by P, is defined as follows,*

$$P.var = P_1.var \cup P_2.var, \quad P.init = P_1.init \times P_2.init,$$
$$P.Act = P_1.Act \cup P_2.Act, \quad P.rIF = P_1.rIF \cup P_2.rIF.$$

The following theorem ensures that the above syntactic definition is consistent with the semantic definition.

Theorem 13 (Semantics of process parallel composition). *If $P = P_1 \parallel P_2$ is defined, the transition system $\underline{P}$ of the composition P is the product of $\underline{P_1}$ and $\underline{P_2}$, that is,*

1. *the states $\Sigma_{P.var} = \Sigma_{P_1.var} \times \Sigma_{P_2.var}$,*
2. *the initial states $P.init = P_1.init \times P_2.init$,*
3. *the transition labels $\Omega = 2^{\omega(P_1.rIF)^*} \cup 2^{\omega(P_2.rIF)^*}$, and*
4. *$(s_1, s_2) \xrightarrow{e/E} (s_1', s_2')$ is a transition of $\underline{P}$ if either*
   - *$s_2 = s_2'$ and $s_1 \xrightarrow{e/E} s_1'$ is a transition of $\underline{P_1}$, or*
   - *$s_1 = s_1'$ and $s_2 \xrightarrow{e/E} s_2'$ is a transition of $\underline{P_2}$.*

It can be shown that a parallel composition of processes preserves the refinement relation between processes.

6.3 Parallel Composition of Components

We continue with introducing composition operators to build larger components.

Definition 26 (Parallel composition of components). *Given two components K_1 and K_2, either closed or open, the parallel composition $K_1 \parallel K_2$ is defined, provided the following conditions hold,*

1. *they do not have common variables, $K_1.var \cap K_2.var = \emptyset$,*
2. *they do not have common provided methods, $K_1.pIF \cap K_2.pIF = \emptyset$,*
3. *they do not have common autonomous actions, $K_1.Act \cap K_2.Act = \emptyset$, and*
4. *they do not have common internal methods, $K_1.iIF \cap K_2.iIF = \emptyset$.*

When the composition is defined, the composed component, denoted by K, is defined as

$$
\begin{aligned}
&K.var = K_1.var \cup K_2.var, \qquad && K.init = K_1.init \times K_2.init,\\
&K.iIF = K_1.iIF \cup K_2.iIF, \qquad && K.pIF = K_1.pIF \cup K_2.pIF,\\
&K.rIF = (K_1.rIF \cup K_2.rIF) \setminus (K_1.pIF \cup K_2.pIF),\\
&K.Act = K_1.Act \cup K_2.Act.
\end{aligned}
$$

It is important to note that this syntactic definition is actually consistent with the semantic definition by the transition systems.

Definition 27 (Parallel composition of component transition systems). *The labeled transition system $\underline{K} = \langle K.var, K.init, \Omega, K.Act\rangle$ of the parallel composition $K = K_1 \parallel K_2$ is defined by*

1. *Ω is defined from the interface K as for a primitive component,*
2. *$(s_1, s_2) \xrightarrow{a/E^-} (s_1', s_2')$ is a transition of $\underline{K}$ if one of the following conditions holds.*
   (K_1 step) When $a \in \omega(K_1.pIF)$

(a) there exists a transition $s_1 \xrightarrow{a/E} s_1'$ *of* $\underline{K_1}$*, with*

$$tr_0 \cdot m_1 \cdot tr_1 \cdots m_k \cdot tr_k \in E,$$

where $m_i \in \omega(K_2.pIF)$ *and* $tr_i \in \omega(K.rIF)^*$ *for* $0 \leq i \leq k$*, then*

(b) if $k = 0$*,* $s_2' = s_2$

(c) for each $s_2 \xRightarrow{m_1/E_1,\ldots,m_k/E_k} s$ *in* K_2 *(i.e.,* $k > 0$*)*

- $s_2' =$ **error** *if* $(E_1 \cup \cdots \cup E_k) \cap \omega(K_1.pIF) \neq \emptyset$*, and*
- $s_2' = s$ *otherwise,*

(d) $tr_0 \cdot E_1 \cdot tr_1 \cdots E_k \cdot tr_k \subseteq E^-$*, and*

(e) E^- *is the smallest set that satisfies above conditions.*

(K_2 step) When $a \in \omega(K_2.pIF)$*, the transition is defined symmetrically.*

(action step) When $a \in K.Act$*, the transition is defined like the process step in Definition 24 for the coordination of a component by a process. Here, the autonomous actions* K *can be seen as a process step and the rest can be seen as a component step.*

The **error** state is a designated deadlock state used to explicitly mark failures from cyclic method calls in compositions violating the *run to complete semantics*. We define a composed state (s_1, s_2) an **error** state if either s_1 or s_2 is **error**. We will discuss the nature of the **error** state in more detail in the next section.

Notice that the required interfaces of K_1 and K_2 do not have to be disjoint, they may require common services. A special case for the parallel composition is that when K_1 and K_2 are totally disjoint, i.e., there is no overlapping in the required interfaces and no component provides methods that the other requires. In this case, we call $K_1 \parallel K_2$ a *disjoint union*, and denote it by $K_1 \otimes K_2$. Even more specifically, when K_1 and K_2 are closed components, $K_1 \parallel K_2$ is always a disjoint union and $K_1 \otimes K_2$ is also a closed component.

For the refinement of components, we have that $\parallel$ is monotonic.

Theorem 14 (Parallel composition preserves refinement). *If* $K_1 \sqsubseteq J_1$ *and* $K_2 \sqsubseteq J_2$*, then* $K_1 \parallel K_2 \sqsubseteq J_1 \parallel J_2$*.*

The parallel composition of components is commutative. Since methods from the provided interface are never hidden from a composition, it is also associative.

Theorem 15. $K_1 \parallel K_2 = K_2 \parallel K_1$ *and* $J \parallel (K_1 \parallel K_2) = (J \parallel K_1) \parallel K_2$*.*

6.4 Renaming and Restriction

When we compose components (including processes), sometimes the provided method names and required method names do not match. We often need to rename some methods. A rename function for a component is a one-one function on the set of methods.

Definition 28 (Renaming). *Let* K *be a component and* f *a renaming function, we use* $K[f]$ *to denote the component for which all the specifications are the same as those of* K*, except for the provided and required interfaces, which are defined by*

1. $K[f].pIF = \{f(m) \mid m \in K.pIF\}$,
2. $K[f].rIF = \{f(m) \mid m \in K.rIF\}$, *and*
3. *any occurrence of* m *in* K *is replaced by* $f(m)$.

Notice that we do not allow renaming internal interface methods, thus an implicit assumption is that a provided or required interface method is not renamed to an internal interface method. This is equivalent to require that $f(m) = m$ for all $m \in K.iIF$.

As a component only involves a finite number of methods, thus a renaming function is only effective on these methods. Therefore, for any renaming functions f and g, if $f(m) = g(m)$ for any method name m of K, then $K[f] = K[g]$. In particular, $K[n/m]$ is the component obtained from K by renaming its method m to n. This is extended to the case when a number of methods are renamed, i.e., $K[n_1/m_1, \ldots, n_k/m_k]$, which is similar to the renaming function in process algebras.

Example 7. For the memory component M in Example 1, $M[put/W, get/R]$ is the same as M except that any occurrence of the method name W is replaced by *put* and any occurrence of the method name R is replaced by *get*.

It is often the case when using a component in a context that some provided methods are restricted from being accessed by the environment. However, these methods cannot be simply removed. Instead, they should be moved to the internal interface so that they can be still called by the other provided and internal methods of the same component.

Definition 29 (Restriction). *Let* K *be a component and* β *a subset of the names of the provided methods of* K*, the component* $K \setminus \beta$ *is obtained from* K *by moving the declarations of the methods in* β *from the provided interface section to the internal interface section, that is,*

$$(K \setminus \beta).pIF = K.pIF \setminus \{m(u;v)\{c\} \mid m \in \beta \wedge m(u;v)\{c\} \in K.pIF\},$$
$$(K \setminus \beta).iIF = K.iIF \cup \{m(u;v)\{c\} \mid m \in \beta \wedge m(u;v)\{c\} \in K.pIF\}.$$

Example 8. Let M be the memory component given in Example 1, *Buff* the open component in Example 3, and $B = (M \parallel Buff) \setminus \{W, R\}$. Thus, B is a one-place buffer component. Further, let $B_i = B[get_i/get, put_i/put]$, for $i = 1, 2$. We now use process *Shift* to coordinate $B_1 \otimes B_2$ and define

$$Buff_1 = ((B_1 \otimes B_2) \parallel Shift) \setminus \{get_1, put_2\},$$

$Buff_1$ is a two-place buffer.

Notice that in the above example, the closed component M provides all the methods required by the open component *Buff*. The restriction of $\{W, R\}$ from the composition $M \parallel Buff$ makes W and R only accessible to *Buff*. We call such a restricted composition *plugging*, and denote it by $M \gg Buff$. In general, we have the following definition.

Definition 30 (Plugging). *Let K_1 and K_2 be components such that $K_1 \parallel K_2$ is defined. If $K_2.rIF \subseteq K_1.pIF$, then we define the* plugging of K_1 with K_2 *by*

$$K_1 \gg K_2 = (K_1 \parallel K_2) \setminus K_2.rIF.$$

In the following example, we build the two-place buffer in a different way.

Example 9. We first define the following open component.

```
1  component Connector {
2     int z;
3     provided interface { shift() { get1(; z); put2(z) } };
4     required interface { get1(; int z); put2(int z) }
5  }
6  process P {
7     required interface { shift() }
8  }
```

Then, we can define the component

$$Buff_2 = ((B_1 \otimes B_2) \gg Connector \parallel P) \setminus \{shift\}.$$

This can be simply written as $Buff_2 = ((B_1 \otimes B_2) \gg Connector) \gg P$. One can prove that $Buff_2$ is equivalent to the component $Buff_1$ in Example 8.

When a number of components are coordinated by a process, the components are not aware of which other components they are working with or exchange data with. Another important aspect is the separation of local data functionality of a component from the control of its interaction protocols. We can design components in which the provided methods are not guarded and thus have no access control. Then, using connectors and coordinating processes the desired interface protocols can be designed. In terms of practicability, most connectors and coordinating processes in applications are data-less, thus having a purely event-based interface behavior. This allows rCOS to enable the separation of design concerns of data functionality from interaction protocols.

6.5 More Examples

The memory component M given in Example 1 models a perfect memory cell in the sense that its content will not be corrupted. As in Liu's and Joseph's work on fault-tolerance [44], a fault can be modeled as an internal autonomous action. We model a faulty memory, where an occurrence of a fault corrupts the content of the memory.

```
component fM { // faulty memory
   provided interface MIF {
      Z d; bool start = false;
      W(Z v) { true & (d := v; start := true) }
      R(; Z v) { start & v := d }
   };
   actions {
      fault() { true & true ⊢ d' ≠ d } // corrupting the memory
   }
}
```

Now we show how to use three faulty memories to implement a perfect memory. First, for $i = 1, 2, 3$, let $fM_i = fM[W_i/W, R_i/R]$. We define the following open component.

```
component V { // majority voting
   provided interface VIF {
      W(Z v) { W1(v); W2(v); W3(v) }
      R(; Z v) {
         var Z v1, Z v2, Z v3;
         R1(; v1); R2(; v2); R3(; v3);
         vote(v1, v2, v3; v);
         end v1, v2, v3
      }
   }
   required protocol { // interleaving of all fM_i's provided protocols
      ...
   }
}
```

Then we have the composite component $(fM_1 \parallel fM_2 \parallel fM_3) \gg V$, that can be proven to be equivalent to the memory component M in Example 1. The proof requires an assumption that at any time at most one memory is faulty. This involves the use of auxiliary variables to record the occurrence of the fault [44]. The architecture of this fault-tolerant memory is shown in the component diagram in Fig. 11, which is a screen-shot from the rCOS Modeler.

In this section, we have defined the general parallel composition for components (including closed components, open components and processes). However, it is important to develop a theory of compositions, techniques for checking composabilities among components, and refinement calculi for the different models with respect to parallel composition and restriction. These have been partly studied in the rCOS literature [12, 19, 20, 25, 26, 45, 47, 62, 66]. However, the semantic models defined in this chapter extend the models in those papers. Thus, the theory and techniques of refinement and composability need a reinvestigation. In the following section, we present preliminary work on how an interface model of components supports composability checking, focusing on the interaction between components.

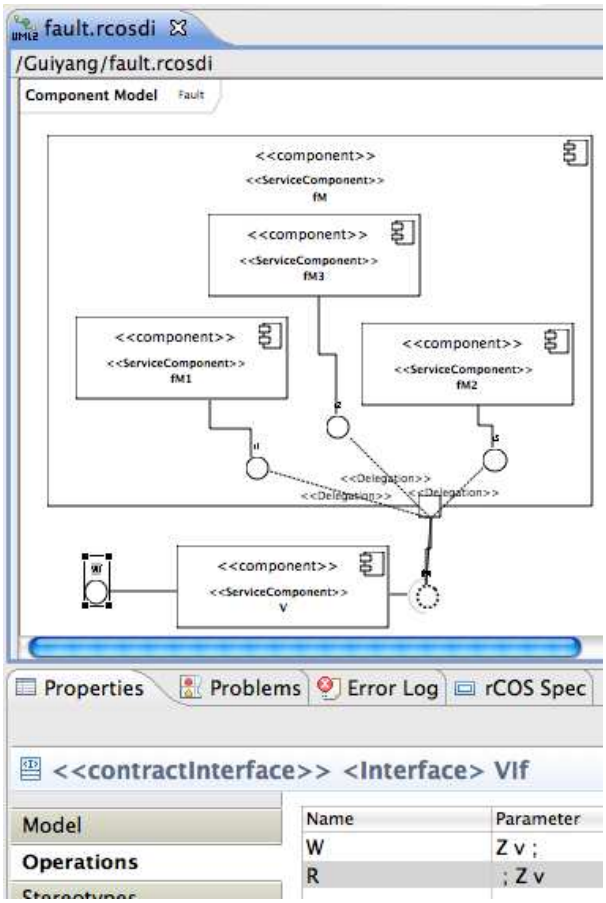


Fig. 11. Component-based design of a fault-tolerant component

7 Interface Model of Components

In rCOS, the refinement of data functionality is dealt within the unified semantic theory for sequential programming presented in Sect. 2. Interactions are handled with the failure-divergence semantics of components. In this section, we first present a model of components that abstracts the data states away, thus focusing only on interactions. We call this model *component automata*. This model still exhibits non-determinism caused by autonomous actions and encounters difficulties in checking composability, for third party composition in particular. Therefore, we will define an interface model for components, called *interface publication automata*. An interface publication automaton is *input-deterministic*, that is, at each step the choice among provided method invocations is controlled by the environment. Both models are simplified labeled transition systems, but

the states are only symbolic states for control of dynamic flows. We also focus on finite state components.

7.1 Component Automata

First, some preliminary notations are defined that we are going to use in the discussion in this section. For a pair $\ell = (e_1, e_2)$ in a product of sets $A_1 \times A_2$, we define the projection functions π_i, for $i = 1, 2$, that is, $\pi_1(\ell) = e_1$ and $\pi_2(\ell) = e_2$. The projection functions are naturally extended to sequences of pairs and sets of sequences of pairs as follows. Given a sequence of pairs $tr = \langle \ell_1, \ldots, \ell_k \rangle$ and a set T of sequences of pairs, we define $\pi_i(tr) = \langle \pi_i(\ell_1), \ldots, \pi_i(\ell_k) \rangle$, and $\pi_i(T) = \{\pi_i(tr) \mid tr \in T\}$, for $i = 1, 2$.

Given $\Gamma \subseteq \Omega$ and a sequence $tr \in \Omega^*$, $tr\lfloor_\Lambda$ is the restriction of tr to element in Λ, returning the sequence obtained from tr by keeping only those elements in Λ. We also extend the restriction function to sets of sequences and define $T\lfloor_\Gamma = \{tr\lfloor_\Gamma \mid tr \in T\}$.

We now introduce a symbolic version of the labeled transition systems, called component automata, by replacing the variables with a set of states and abstract the data parameters from the interface methods. We only consider finite state automata.

Definition 31 (Component automaton). *A* **component automaton** *is a tuple $K = \langle \Sigma, s_0, pIF, rIF, Act, \delta \rangle$, where*

- *Σ is a finite set of states and $s_0 \in \Sigma$ is the initial state;*
- *pIF, rIF, and Act are disjoint finite sets of* **provided**, **required** *and* **internal** *events, respectively;*
- *$\delta \subseteq \Sigma \times \Omega(pIF, rIF, Act) \times \Sigma$ is the transition relation, where the set of* **labels** *is defined as $\Omega(pIF, pIF, Act) = (pIF \cup Act) \times (2^{rIF^*} \setminus \{\emptyset\})$, simply denoted by Ω when there is no confusion.*

As before, we use e/E to denote a pair (e, E) in Ω and a transition $(s, e/E, s') \in \delta$ by $s \xrightarrow{e/E} s'$. This transition is called a step of *provided transition* if $e \in pIF$, otherwise an *autonomous transition*. We use $s \xrightarrow{e/} s'$ for $s \xrightarrow{e/\{\epsilon\}} s'$. Notice that $e/\emptyset$ is not a label in an automaton, and a transition without required events is a transition by a label of the form $e/\{\epsilon\}$. The internal events are prefixed with a semicolon, e.g., $s \xrightarrow{;e/E} s'$, to differentiate them from a transition step by a provided event. We use τ to represent an internal event when it causes no confusion. For a state s, we define the set of events with outgoing transitions by

$$out(s) = \{e \in pIF \cup Act \mid \exists s', E \bullet s \xrightarrow{e/E} s'\}.$$

Further, let $out^\circ(s) = out(s) \cap Act$ and $out^\bullet(s) = out(s) \cap pIF$. We write $s \xrightarrow{e/} s'$ for $s \xrightarrow{e/E} s'$, when E is not significant.

Notice that there are no guards for transitions. Instead, the guards of actions from the general component transition systems are encoded in the states of the automata. A composite event e/E in Ω is *enabled* in a state s if there exists a transition $s \xrightarrow{e/E} s'$, and an action $e \in pIF \cup Act$ is enabled in a state $s \in \Sigma$, if there is a set of sequences $E \in 2^{rIF^*}$ such that $s \xrightarrow{e/E} s' \in \delta$. Then, the executions of an automaton C can be defined in the same way as for a labeled transition system. However, the execution traces and the interaction traces are of no significant difference. Formally, we have the following definitions and notations,

- a sequence of transitions $s \xrightarrow{\ell_1} s_1 \cdots \xrightarrow{\ell_k} s'$ is called an execution sequence, and $\langle \ell_1, \ldots, \ell_k \rangle$ is called a trace from s to s',
- we write $s \xRightarrow{\ell_1,\ldots,\ell_k} s'$ if there exists an execution sequence $s \xrightarrow{\ell_1} s_1 \cdots \xrightarrow{\ell_k} s'$,
- for a trace tr over Ω and a state s, $target(tr, s) = \{s' \mid s \xRightarrow{tr} s'\}$, and $target(tr) = target(tr, s_0)$,
- for a sequence sq over $pIF \cup Act$, we write $s \xRightarrow{sq} s'$ if there is a trace tr such that $s \xRightarrow{tr} s'$ and $\pi_1(tr) = sq$,
- $\mathcal{T}(s) = \{\langle \ell_1, \ldots, \ell_k \rangle \mid \exists s' \bullet s \xRightarrow{\ell_1,\ldots,\ell_k} s'\}$, and it is called the traces of s,
- $\mathcal{T}(s_0)$ is the set of traces of the component automaton C, it is also denoted by $\mathcal{T}(C)$,
- for a state s, the *provided traces* for s are given by

$$\mathcal{PT}(s) = \{\pi_1(tr) \lfloor pIF \mid tr \in \mathcal{T}(s)\},$$

- the set $\mathcal{PT}(s_0)$ is called the set of *provided traces* of C, and it is also written as $\mathcal{PT}(C)$.

Example 10. Consider the internet-connection component presented in Fig. 12. It provides the services *login*, *print*, and *read* to the environment and there is an internal service ; *wifi*. The services model the login into the system, invocation of printing a document, an email service, and automatically connecting the wifi, respectively. The component calls the services unu_1, unu_2, *cserv*, *cprint* and *senddoc*. The first three of them model the searching for a wifi router nearby, connecting to either the unu_1 or unu_2 wireless network, and then to an application server, respectively. The services *cprint* and *senddoc* connect to the printer, send the document to print and start the printing job. The *print* service is only available for the wifi network unu_1, while *read* can be accessed from both networks.

The component automaton in Fig. 12 can perform, for example, the following execution,

$$\langle 0, (login/\{\epsilon\}), 1, (; wifi/\{unu_1\}), 2, (print/\{cprint \cdot senddoc\}), 2\rangle.$$

Now $pt = \langle login, print\rangle$ is a provided trace of the execution and the set of required traces of pt is $\{\langle unu_1 \cdot cprint \cdot senddoc\rangle\}$.

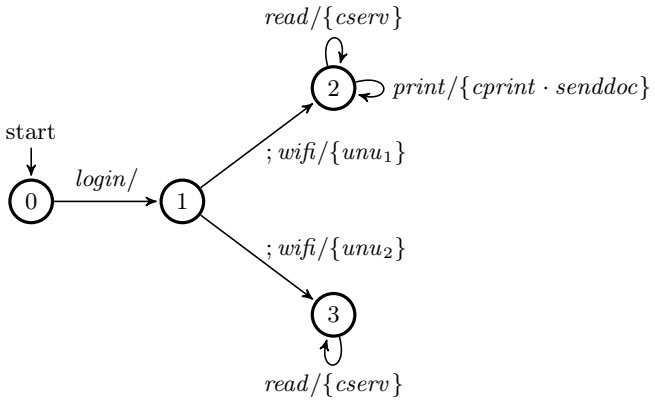


Fig. 12. Automaton of internet connection component C_{ic}

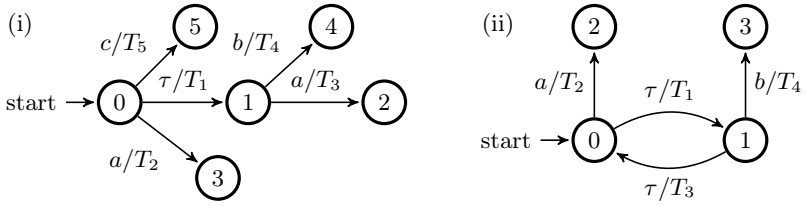


Fig. 13. Examples of enabled actions

7.2 Non-blockable Provided Events and Traces

The model of component automata describes how a component interacts with the environment by providing and requiring services. However, some provided transitions or executions may be blocked due to the non-determinism caused by autonomous actions. In this section, we will discuss about the non-refusal of provided events and traces.

Figure 13 shows what kinds of provided events can be refused (or blocked). The states 0 and 1 in either automaton are indistinguishable, because every internal autonomous transition (a τ transition) will be taken eventually. Therefore, a state, at which an internal autonomous transition may happen, like state 0, is called an *unstable state*. In general, a state s of a component automaton C is *stable* if $out^{\circ}(s) = \emptyset$, that is, no internal actions are enabled, otherwise s is unstable. A state s is a *deadlock state* if there is no action enabled at all. A deadlock state is always a stable state.

Now consider state 1 in the automaton in Fig. 13(i), it is a stable state. The automaton will eventually leave the initial state 0, because if a or c are never tried, the autonomous τ transition will be eventually performed. Thus the provided action b cannot be refused (blocked) in state 0 or 1, because it is enabled in state 1. We call a state s' *internally reachable* from state s, denoted

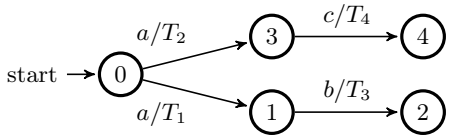


Fig. 14. An example of a refusal trace

by $autoR(s, s')$, if there is a sequence (possibly empty and in that case $s' = s$) $s \xrightarrow{\tau_1,\ldots,\tau_k} s'$ of internal transitions from s to s'. We can see that in the automaton in Fig. 13(i) state 1 is internally reachable from state 0 and all internal executions (only one in this case) reach state 1. For this reason, we say provided action b cannot be refused from state 0. Notice that in the automaton in Fig. 13(ii), there is no internally reachable stable state from 0.

We define $autoR(s) = \{s' \mid autoR(s, s')\}$ to be the set of internally reachable states from s, and $autoR^\bullet(s)$ the set of internally reachable stable states, i.e., $\{s' \mid s' \in autoR(s)$ *and* s' is stable$\}$. Notice that $autoR(s, s')$ is a transitive relation and $autoR(s)$ is closed under this relation. Thus, $autoR^\bullet(s) = \emptyset$ if there exists a livelock state in $autoR(s)$, i.e., an infinite sequence of internal transitions is possible.

With the above discussion, we are ready to define the refusal provided events of a component automaton. Informally, a provided event in *pIF* is a *refusal* of state s if it is not enabled at one of the internally reachable stable states s' of s.

Definition 32 (Refusal events). *Let s be a state of a component automaton C, the set of* **local refusal events** *at s is*

$$\mathcal{R}(s) = \{e \mid e \in pIF \wedge \exists s' \in autoR^\bullet(s) \bullet e \text{ is disabled at } s'\}.$$

We use $\overline{\mathcal{R}}(s)$ to denote the set of **local non-refusal** *(non-blockable) events at s.*

The important concept we are developing in this subsection is the notion of *non-refusal traces*, that is going to be used for the publication of a component.

Consider the component automaton shown in Fig. 14. The provided event a is enabled at state 0, however, after the invocation of a, the component determines internally whether to move to state 1 or 3. So, both of b and c may be refused after a.

Definition 33 (Non-blockable provided trace). *Let $\langle a_1, \cdots, a_k\rangle$, $k \geq 0$, be a sequence of provided events of a component automaton C. It is called a* **non-blockable provided trace** *at state s if for $0 \leq i \leq k-1$ and any state s' such that $s \stackrel{tr}{\Longrightarrow} s'$ and $\pi_1(tr)\lfloor pIF = \langle a_1, \ldots, a_i\rangle$, a_{i+1} is not a refusal at s', i.e., $a_{i+1} \in \overline{\mathcal{R}}(s')$.*

A trace tr of a component automaton C is *non-blockable* at a state s, if the provided trace $\pi_1(tr)\lfloor_{pIF}$ is non-blockable at s. We use $\mathcal{PP}(s)$ and $\mathcal{UT}(s)$ to denote

the set of all non-blockable provided traces (also denoted as provided protocols like in the previous sections) and non-blockable traces at state s, respectively. When s is the initial state of C, we also write $\mathcal{PP}(s)$ and $\mathcal{UT}(s)$ as $\mathcal{PP}(C)$ and $\mathcal{UT}(C)$, respectively.

7.3 Interface Publication Automata

We now define a model of input-deterministic component automata that have non-blockable traces only. We use this model for *publications of components* as they give better composability checking. The main result of this subsection is the design of an algorithm that transforms a general component automaton to such an *interface publication automaton*.

Definition 34 (Input-determinism). *A component automaton*

$$C = \langle S, s_0, P, R, A, \delta \rangle$$

is input-deterministic *if for any* $s_0 \xRightarrow{tr_1} s_1$ *and* $s_0 \xRightarrow{tr_2} s_2$ *such that*

$$\pi_1(tr_1)\lfloor_{pIF} = \pi_1(tr_2)\lfloor_{pIF},$$

the sets of non-blockable events are identical, i.e., $\overline{R}(s_1) = \overline{R}(s_2)$.

The definition says that any provided event e is either a refusal or a non-refusal at both of any two states s_1 and s_2 that are reachable from the initial state through the same provided trace. Therefore, any provided trace of an input-deterministic component automaton is not blocked, provided the required events are acceptable by the environment. Thus, we call an input-deterministic automaton an *interface publication automaton*.

The following theorem states that all the traces of an input-deterministic component automaton are non-blockable.

Theorem 16. *A component automaton C is input-deterministic iff* $\mathcal{PT}(C) = \mathcal{PP}(C)$.

Proof. $\mathcal{PT}(C) = \mathcal{PP}(C)$ means every provided trace of C is non-blockable actually.

First, we prove the direction from left to right. From the input-determinism of C follows that for each provided trace $pt = (a_0, \ldots, a_k)$ and each state s with $s_0 \xRightarrow{tr} s$ and $\pi_1(tr) = \langle a_0, \ldots, a_i \rangle$ for $0 \leq i \leq k-1$, the set $\overline{\mathcal{R}}(s)$ is the same. Since pt is a provided trace (i.e., there exists at least one such s, where a_{i+1} is enabled), so $a_{i+1} \in \overline{\mathcal{R}}(s)$ for all such s. This shows that all the provided traces are non-blockable, so all the 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 two traces tr_1 and tr_2 with $\pi_1(tr_1)\lfloor_P = \pi_1(tr_2)\lfloor_P$ and $s_0 \xRightarrow{tr_1} s_1$, $s_0 \xRightarrow{tr_2} s_2$ such that $\overline{\mathcal{R}}(s_1) \neq \overline{\mathcal{R}}(s_2)$.

Without loss of generality, we assume that there is a provided event a such that $a \in \overline{\mathcal{R}}(s_1)$ and $a \notin \overline{\mathcal{R}}(s_2)$. Now $\pi_1(tr_1) \cdot \langle a \rangle$ is a provided trace of C that is blockable, which contradicts the assumption. □

We now present a procedure in Algorithm 1 that, given a component automaton C, constructs the interface publication automaton $\mathcal{I}(C)$. Each state of $\mathcal{I}(C)$ is a pair (Q, r) of a subset Q of states and a single state r of C. A pair (Q, r) is a state of $\mathcal{I}(C)$ if for some provided trace pt of C

- $s_0 \stackrel{pt}{\Longrightarrow} r$, and
- $Q = \{s \mid s_0 \stackrel{pt}{\Longrightarrow} s\}$.

Thus, in a tuple (Q, r), $r \in Q$ and the first element represents the set of all potentially reachable states for a given provided trace, whereas the second element is a specific reachable state for this trace. Notice that for the same r but a provided trace pt_1 different from pt such that $s_0 \stackrel{pt_1}{\Longrightarrow} r$, there may be a different state (Q', r) for $\mathcal{I}(C)$. Then, the transition relation of $\mathcal{I}(C)$ is defined as $(Q, r) \stackrel{e/E}{\longrightarrow} (Q', r')$ in $\mathcal{I}(C)$ if

- e is not a refusal at any state in Q, that is, $e \in \bigcap_{s \in Q} \overline{\mathcal{R}}(s)$,
- $r \stackrel{e/E}{\longrightarrow} r'$, and
- $Q' = \{s' \mid \exists s \in Q \bullet s \stackrel{e}{\Longrightarrow} s'\}$.

From this it becomes clear that the first element Q of a compound state (Q, r) is necessary to compute if a transition step for an event e is possible at all, which is not the case if e can be refused in any state reachable by the same provided trace. The second element r is needed to identify the full transition step enabled in the state r; note that the required traces E in a compound event e/E can differ for all states in the set Q.

Algorithm 1 computes the pairs (Q, r) defined above and the transition relation to simulate the non-blockable executions of C. In the algorithm, the first elements of these state pairs, i.e., the sets $\{Q \mid \exists r \bullet (Q, r) \text{ is a state of } \mathcal{I}(C)\}$, result from a power-set construction similar to the construction of a deterministic automaton from a non-deterministic automaton. The variables *todo* and *done* are used to collect new reachable states that still have to be processed and states that have already been processed, respectively.

Example 11. In the internet connection component automaton given in Fig. 12, the provided trace $\langle login, read \rangle$ is non-blockable. However, $\langle login, print \rangle$ may be blocked during execution, because after *login* is called, the component may transit to state 3 at which *print* is not available. We use Algorithm 1 to generate the interface publication automaton in Fig. 15.

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

Theorem 17 (Correctness of Algorithm 1). *The following properties hold for Algorithm 1.*

1. *For any given component automaton, the algorithm always terminates.*
2. *For any component C, $\mathcal{I}(C)$ is an input-deterministic automaton.*

Algorithm 1: Construction of interface automaton $\mathcal{I}(C)$

Require: $C = (S, s_0, P, R, A, \delta)$
Ensure: $\mathcal{I}(C) = (S_I, (\{s_0\}, s_0), P, R, A, \delta_I)$, where $S_I \subseteq 2^S \times S$
initialization
  $S_I := \{(\{s_0\}, s_0)\}$; $\delta_I := \emptyset$; $todo := \{(\{s_0\}, s_0)\}$; $done := \emptyset$
end initialization
while $todo \neq \emptyset$ **do**
  choose $(Q, r) \in todo$; $todo := todo \setminus \{(Q, r)\}$; $done := done \cup \{(Q, r)\}$
  foreach $a \in \bigcap_{s \in Q} \overline{\mathcal{R}}(s)$ **do**
    $Q' := \bigcup_{s \in Q} \{s' \mid s \stackrel{a}{\Longrightarrow} s'\}$
    foreach $(r \xrightarrow{e/E} r') \in \delta$ **do**
      $\delta_I := \delta_I \cup \{(Q, r) \xrightarrow{e/E} (Q', r')\}$
      if $(Q', r') \notin (todo \cup done)$ **then**
        $todo := todo \cup \{(Q', r')\}$
        $S_I := S_I \cup \{(Q', r')\}$
  foreach $r \xrightarrow{\tau} r'$ **with** $r' \in Q$ **do**
    $\delta_I := \delta_I \cup \{(Q, r) \xrightarrow{\tau} (Q, r')\}$

3. $\mathcal{PP}(C) = \mathcal{PP}(\mathcal{I}(C))$ *and* $\mathcal{UT}(C) = \mathcal{UT}(\mathcal{I}(C))$.

The termination of the algorithm is obtained, because *todo* will eventually be empty: the set *done* increases for each iteration of the loop in the algorithm, and the union of *done* and *todo* is bounded. The proofs of the other two properties are given in [20].

It is interesting to study the states of $\mathcal{I}(C)$. Each of them is a pair (Q, r) of a set of a states of C and a state r. The state r and all states $s \in Q$ are target states of the same provided trace pt in C. Each transition from (Q, r) in $\mathcal{I}(C)$ adds a non-blockable event e to pt. Therefore, in the automaton $\mathcal{I}(C)$, $\overline{\mathcal{R}}((Q, r)) = \bigcap_{s \in Q} \overline{\mathcal{R}}(s)$, where $\overline{\mathcal{R}}(s)$ are non-blockable events of s in C. We simply write $\overline{\mathcal{R}}(Q, r)$. This means that each (Q, r) in $\mathcal{I}(C)$ actually encodes non-blockable events in an execution (called *global non-blockable events*) up to the state r in the execution of C. We also define the set of events that are enabled locally at a state s but globally refused (or blocked) as $\mathcal{B}(s) = \overline{\mathcal{R}}(s) \setminus \bigcup_{(Q,s) \in S_I} \overline{\mathcal{R}}(Q, s)$. This will be used in the definition of *alternative simulation* in the following subsection.

7.4 Composition and Refinement

The composition operations for component automata are derived from those for the component labeled transitions systems defined in Sect. 6. Further, component automata are special labeled transition systems and there is no data functionality (pre- and post-condition for the actions), and we do not have recursively

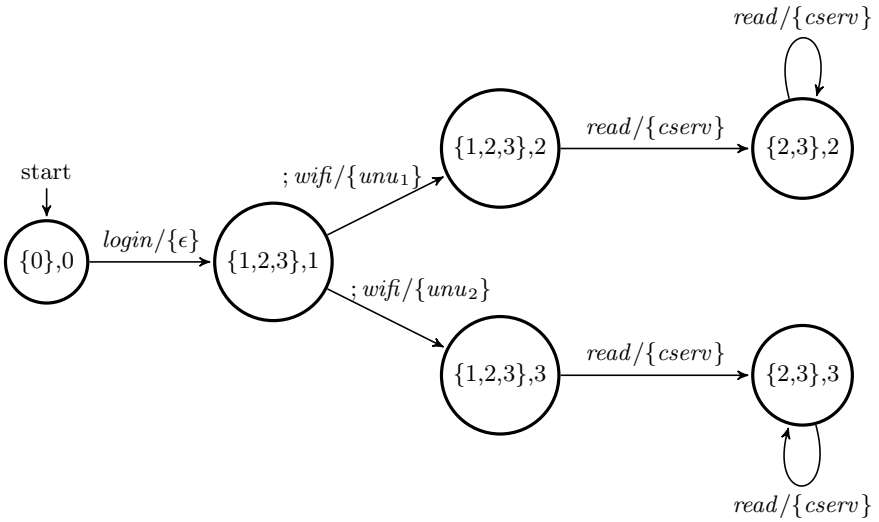


Fig. 15. Interface publication automaton of the internet connection component

defined component automata. Thus, we do not have to deal with divergent behavior caused by recursion. However, there are still possible livelock states in the dynamic behavior, but this can be characterized by interaction failures (similar to the CSP stable failure semantics [57]) with explicit consideration of livelocks. Another kind of execution failure is caused by cyclic method invocations of two interacting components. As discussed in Sect. 6, we use one dedicated state **error**. A transition that encounters cyclic method invocations will take the composite component to this state. In fact, **error** is a deadlock state in which no actions, including internal autonomous events are enabled. However, the difference of an **error** state from a normal deadlock state or a termination state is that the event e that leads from a state s to the **error** state, $s \xrightarrow{e} \textbf{error}$, is disabled in state s too, and it should be eliminated from the traces of the system. Similarly, in a composition $K_1 \parallel K_2$, if a provided action in K_1 invokes a method provided by K_2 that is disabled, the transition also enters the **error** state. An automaton with the **error** state is depicted in Fig. 16. A general component automaton may also have the designated **error** state; an automaton with the **error** state that is not reachable from the initial state is equivalent to one without the **error** state.

More precisely, for an event $e \in \Omega$ and a state s of a component automaton C, if $s \xrightarrow{e} \textbf{error}$, e is disabled in state s. Thus, for a general automaton with **error** state, the definition of the refusal events $\mathcal{R}(s)$ should take such an e into account. Then, Algorithm 1 also removes the transitions to **error**.

We now define two notions of failure sets for a component automaton.

Definition 35 (Failures sets of component automata). *Let C be a component automaton with the* **error** *state. A* **failure** *of C is a pair (tr, X) of a*

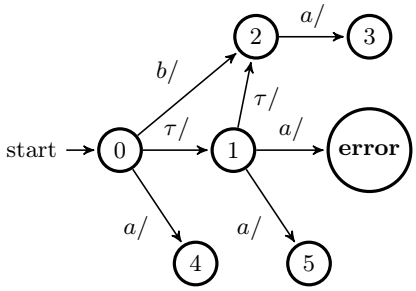


Fig. 16. An automaton with **error** state

trace and a set of the events of Ω of C such that for every $e \in X$ there exists $s \in target(tr)$ such that $e \in \mathcal{R}(s)$, i.e., e is blocked in s, or s is a livelock state (i.e., an infinite sequence of internal steps is possible) and $X \subseteq \Omega$. We use $\mathcal{F}(C)$ to denote the set of failures of C.

It can be shown that the set of traces $\mathcal{T}(C)$ and the set of provided traces $\mathcal{PT}(C)$ are given by

$$\mathcal{T}(C) = \{tr \mid \exists X \bullet (tr, X) \in \mathcal{F}(C)\}$$
$$\mathcal{PT}(C) = \{\pi_1(tr) \mid tr \in \mathcal{T}(C)\}$$

Analogously to the traces of open components defined in Sect. 4, for each provided trace pt, there is an associated set of sequences of required events,

$$\mathcal{RP}(pt) = \bigcup \{E_1 \cdots E_k \mid \exists tr \in \mathcal{T}(C) \bullet \pi_1(tr) = pt \wedge \pi_2(tr) = E_1 \cdots E_k\}.$$

The set of required traces for the non-blockable provided traces is given by $\mathcal{RP}(C) = \bigcup \{\mathcal{RP}(pt) \mid pt \in \mathcal{PP}(C)\}$. Now we formally define the refinement relation between component automata as the following partial order on their failures.

Definition 36 (Failure refinement of component automata). *A component automaton C_2 is a* **refinement** *of a component automaton C_1, denoted as $C_1 \sqsubseteq_f C_2$, if $\mathcal{F}(C_2) \subseteq \mathcal{F}(C_1)$.*

The properties of refinement between component automata, e.g., reflexivity and transitivity, are preserved by composition operations. However, we are interested in a "refinement relation" defined in terms of non-blocking provided and required traces.

Definition 37 (Alternative simulation). *A binary relation R over the set of states of a component automaton C is an* **alternative simulation** *if whenever $s_1 \; R \; s_2$,*

- *for any transition $s_1 \xrightarrow{e/E} s_1'$ with $e \in Act \cup \overline{\mathcal{R}}(s_1) \setminus \mathcal{B}(s_1)$ and* **error** $\notin autoR(s_1)$*, there exist s_2' and E' such that $s_2 \xrightarrow{e/E'} s_2'$, where $E' \subseteq E$ and $s_1' \; R \; s_2'$;*

- *for any transition* $s_2 \xrightarrow{e/E'} s_2'$ *with* $e \in Act \cup \overline{\mathcal{R}}(s_1) \setminus \mathcal{B}(s_1)$ *and* **error** $\notin autoR(s_2)$*, there exist* s_1' *and* E *such that* $s_1 \xrightarrow{e/E} s_1'$*, where* $E' \subseteq E$ *and* $s_1' \ R \ s_2'$*;*
- $\mathcal{B}(s_2) \subseteq \mathcal{B}(s_1)$*;*
- *if* $s_2 \xrightarrow{e/}$ **error** *with* $e \in Act \cup pIF$*, then* $s_1 \xrightarrow{e/}$ **error**.

We say that s_2 **alternative simulates** s_1*, written as* $s_1 \lesssim s_2$*, if there is an alternative simulation relation* R *such that* $(s_1, s_2) \in R$*.* C_2 *is an* **alternative refinement** *of* C_1*, denoted by* $C_1 \sqsubseteq_{alt} C_2$*,* $C_1.init \lesssim C_2.init$*,* $C_1.pIF \subseteq C_2.pIF$ *and* $C_2.rIF \subseteq C_1.rIF$*.*

The above definition is similar to the alternating simulation given in [16] and that is why we use the same term. But they are actually different. The main differences are (1) we only require a pair of states to keep the simulation relation with respect to the provided services that could not result in a deadlock; (2) we also require that a refinement should 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 do not suppose required methods, our definition is stronger than the failure refinement as well as the failure-divergence refinement.

The following theorem indicates that the component publication automaton constructed by Algorithm 1 is a refinement of the considered component automaton with respect to the above definition, which justifies that we can safely use the resulting component interface instead of the component at the interface level.

Theorem 18. *For any component automaton* C*, the refinement relation* $C \sqsubseteq_{alt} \mathcal{I}(C)$ *holds. If* $C_1 \sqsubseteq_{alt} C_2$*, then* $\mathcal{I}(C_1) \sqsubseteq_{alt} \mathcal{I}(C_2)$*.*

Proof. Let $R = \{(s, (Q, s)) \mid s \in S, (Q, s) \in S_I\}$. We show that R is a simulation relation.

For any $s \ R \ (Q, s)$,

- $s \xrightarrow{a/E} s'$ with $a \in \overline{\mathcal{R}}(s)$ and $a \notin \mathcal{B}(s)$. Then $a \in \mathcal{B}(Q, s)$ and $(Q, s) \xrightarrow{a/E} (Q', s')$.
- If $s \xrightarrow{;e/E} s'$ with $;e \in Act$, then $(Q, s) \xrightarrow{;e/E} (Q, s')$.
- For any $(Q, s) \xrightarrow{e/E} (Q', s')$ with $e \in Act \cup \overline{\mathcal{R}}(s) \setminus \mathcal{B}(Q, s)$, then $s \xrightarrow{e/E} s'$ and $s' \ R \ (Q', s')$.
- $\mathcal{B}(Q, s) \subseteq \mathcal{B}(s)$.

Hence, R is a simulation relation.

Now we prove the second part of the theorem. Let R_0 be a simulation for $C_1 \sqsubseteq_{alt} C_2$, then we show

$$R_0' = \left\{ ((Q_1, s_1), (Q_2, s_2)) \;\middle|\; \begin{array}{l} (s_1, s_2) \in R_0, \\ \forall r_1 \in Q_1 \exists r_2 \in Q_2 \bullet (r_1, r_2) \in R_0 \\ \forall r_2 \in Q_2 \exists r_1 \in Q_1 \bullet (r_1, r_2) \in R_0 \end{array} \right\}.$$

For any $(Q_1, s_1)\ R_0'\ (Q_2, s_2)$, $\overline{\mathcal{R}}(Q_1) \subseteq \overline{\mathcal{R}}(Q_2)$ and $\mathcal{B}(Q_1, s_1) = \mathcal{B}(Q_2, s_2) = \emptyset$. Then we can show that this is an alternative simulation between the initial states of $\mathcal{I}(C_1)$ and $\mathcal{I}(C_2)$. □

Theorem 19. *Given two component publication automata C_1 and C_2, if $C_1 \sqsubseteq_{alt} C_2$, then $\mathcal{PP}(C_1) \subseteq \mathcal{PP}(C_2)$, and for any non-blockable provided trace $pt \in \mathcal{PT}(C_1)$, $\mathcal{RP}(pt) \subseteq \mathcal{RP}(pt)$, where $\mathcal{RP}$ on the left is the set of required traces for pt in C_2 and $\mathcal{RP}$ on the right is that defined for C_1.*

This theorem can be proved by induction on the length of pt. The following theorem states that the refinement relation is preserved by the composition operator over component automata. We refer the reader to the paper [20] for its proof.

Theorem 20. *Given a component automaton C and two interface publication automata C_1 and C_2 such that $C_1 \sqsubseteq_{alt} C_2$, then $C_1 \otimes C \sqsubseteq_{alt} C_2 \otimes C$.*

Corollary 1. *Given two component interface automata C_1 and C_2, if $C_1 \sqsubseteq_{alt} C_2$, then $C_1 \parallel C \sqsubseteq_{alt} C_2 \parallel C$.*

The work on interface model in this section is a new development in rCOS. The results are still preliminary. There are still many open problems such as the relation between the failure refinement relation and the alternative refinement between component automata. A thorough study on the relation between CSP failure semantics theory and the automata simulation theory would deserve a Ph.D. thesis.

8 Conclusions

A major research objective of the rCOS method is to improve the scalability of semantic correctness preserving refinement between models in model-driven software engineering. The rCOS method promotes the idea that component-based software design is driven by model transformations in the front end, and verification and analysis techniques are integrated through the model transformations. It attacks the challenges of consistent integration of models for different viewpoints of a software system, for that different theories, techniques and tools can be applied effectively. The final goal of the integration is to support the separation of design concerns, those of the data functionality, interaction protocols and class structures in particular. rCOS provides a seamless combination

of OO design and component-based design. As the semantic foundation presented in Sect. 2 and the models of components show, rCOS enables integration of classical specification and verification techniques, Hoare Logic and Predicate Transformers for data functionality, process algebras, finite state automata and temporal logics for reactive behavior. Refinement calculi for data functionality and reactive behavior are integrated as well.

In this chapter, we presented a model of component-based architecture, which is a generalization of the original rCOS components model presented in our early publications [10,13,25,35]. 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. The presented composition operators for parallel composition and operators for renaming, hiding, and plugging are used for the construction of complex systems out of predefined and refined components. Using refinement of components, this model is particularly suited for model transformations in model-driven software engineering [24,35,47]. Finally, a specific focus of this work was to study the interface behavior of components in order to being able to precisely specify contracts and publications for components, which enable the component's reusability in different contexts. This particularly included the computation of a component's provided protocol, which we identified as necessary to allow correct usage of a component in all desired situations.

Construction of models and model refinements are supported by the rCOS Modeler tool. The method has been tested on enterprise systems [11,12], remote medical systems [65] and service oriented systems [42].

Similar to JML [40], the rCOS method intends to convey the message that the design of a formal modeling notation can and should consider advanced features of architectural constructs in modern programming languages like Java. This will make it easier to use and understand for practical software engineers, who have difficulties to comprehend heavy mathematical constructs and operators.

The link of the rCOS models to classical semantic models is presented in this paper. We have not spent much space on related work as this has been discussed in previous papers, to which we have referred. However, we would like to emphasize the work on CSP and its failure-divergence theory [57] that motivated the input-deterministic interface model. Closely related models are Reo [14] and Circus [64]. The former model is related to the process model in rCOS, and Circus also deals with integration of data state into interaction behavior. In future work, we are interested in dealing with timing issues of components as another dimension of modeling. Also, with the separation of data functionality and flow of interaction control, we would like to investigate how the modeling method can be applied to workflow management, health care workflows in particular [4, 21].

Acknowledgments. In this special occasion of a celebration of He Jifeng's 70th birthday, we would like to express our thanks for his contribution to the development of the semantic foundation of rCOS. Many of our former and current colleagues have made contributions to the development of the rCOS method and its tool support. Xiaoshan Li is another main contributor to the development of the rCOS theory, techniques and tool support. Jing Liu and her group have made significant contributions to the link of rCOS to UML and service oriented architecture; Xin Chen, Zhenbang Chen and Naijun Zhan to the component-based modeling and refinement; Charles Morisset, Xiaojian Liu, Shuling Wang, and Liang Zhao to the object-oriented semantics, refinement calculus and verification; Anders P. Ravn to the design of the tool and the CoCoME case study; Dan Li, Xiaoliang Wang, and Ling Yin to the tool development; Bin Lei and Cristiano Bertolini to testing techniques; and Martin Schäf to the automata-based model of interface behavior. The rCOS methods have been taught in many UNU-IIST training schools, inside and outside Macau, and we are grateful to the very helpful feedback and comments that we have received from the participants.

The work is supported by the projects GAVES, SAFEHR and PEARL funded by the Macau Science and Technology Development Fund, and the Chinese Natural Science Foundation grants NSFC-61103013, 91118007 and 60970031.

References

1. Abrial, J.R.: The B-Book: Assigning Programs to Meanings. Cambridge University Press (1996)
2. Abrial, J.R.: Modeling in Event-B: System and Software Engineering. Cambridge University Press (2010)
3. Back, R.J.R., von Wright, J.: Trace refinement of action systems. In: Proceedings of 5th International Conference on Concurrency Theory, volume 836 of Lecture Notes in Computer Science. pp. 367–384. Springer, Berlin (1994)
4. Bertolini, C., Liu, Z., Schäf, M., Stolz, V.: Towards a formal integrated model of collaborative healthcare workflows. Tech. Rep. 450, IIST, United Nations University, Macao (2011), to appear in Proceedings of 1st International Symposium on Foundations of Health Information Engineering and Systems, Lecture Notes in Computer Science
5. Booch, G.: Object-Oriented Analysis and Design with Applications. Addison-Wesley, Boston (1994)
6. Brooks, F.P.: No silver bullet: Essence and accidents of software engineering. IEEE Computer 20(4), 10–19 (1987)
7. Brooks, F.P.: The mythical man-month: After 20 years. IEEE Software 12(5), 57–60 (1995)
8. Burstall, R., Goguen, J.: Putting theories together to make specifications. In: Reddy, R. (ed.) Proc. 5th Intl. Joint Conf. on Artificial Intelligence. pp. 1045–1058. Department of Computer Science, Carnegie-Mellon University, USA (1977)
9. Chandy, K.M., Misra, J.: Parallel Program Design: A Foundation. Addison-Wesley, Reading (1988)
10. Chen, X., He, J., Liu, Z., Zhan, N.: A model of component-based programming. In: Arbab, F., Sirjani, M. (eds.) International Symposium on Fundamentals of

Software Engineering, volume 4767 of Lecture Notes in Computer Science. pp. 191–206. Springer, Berlin (2007), http://www.iist.unu.edu/www/docs/techreports/reports/report350.pdf
11. Chen, Z., Hannousse, A.H., Hung, D.V., Knoll, I., Li, X., Liu, Y., Liu, Z., Nan, Q., Okika, J.C., Ravn, A.P., Stolz, V., Yang, L., Zhan, N.: Modelling with relational calculus of object and component systems–rCOS. In: Rausch, A., Reussner, R., Mirandola, R., Plasil, F. (eds.) The Common Component Modeling Example, volume 5153 of Lecture Notes in Computer Science, chap. 3, pp. 116–145. Springer, Berlin (2008), http://www.iist.unu.edu/www/docs/techreports/reports/report382.pdf
12. Chen, Z., Liu, Z., Ravn, A.P., Stolz, V., Zhan, N.: Refinement and verification in component-based model driven design. Science of Computer Programming 74(4), 168–196 (Feb 2009), http://www.sciencedirect.com/science/article/B6V17-4T9VP33-1/2/c4b7a123e06d33c2cef504862a5e54d5
13. Chen, Z., Liu, Z., Stolz, V., Yang, L., Ravn, A.P.: A refinement driven component-based design. In: 12th Intl. Conf. on Engineering of Complex Computer Systems (ICECCS 2007). pp. 277–289. IEEE Computer Society (Jul 2007)
14. Clarke, D., Proença, J., Lazovik, A., Arbab, F.: Channel-based coordination via constraint satisfaction. Sci. Comput. Program. 76(8), 681–710 (2011)
15. Clarke, E.M., Emerson, E.A.: Design and synthesis of synchronization skeletons using branching-time temporal logic. In: Kozen, D. (ed.) Logics of Programs. Lecture Notes in Computer Science, vol. 131, pp. 52–71. Springer, Heidelberg (1981)
16. De Alfaro, L., Henzinger, T.: Interface automata. ACM SIGSOFT Software Engineering Notes 26(5), 109–120 (2001)
17. Dijkstra, E.W., Scholten, C.S.: Predicate Calculus and Program Semantics. Springer-Verlag, New York (1990)
18. Dijkstra, E.W.: The humble programmer. Communications of the ACM 15(10), 859–866 (1972), an ACM Turing Award lecture
19. Dong, R., Faber, J., Liu, Z., Srba, J., Zhan, N., Zhu, J.: Unblockable compositions of software components. In: Grassi, V., Mirandola, R., Medvidovic, N., Larsson, M. (eds.) CBSE. pp. 103–108. ACM (2012)
20. Dong, R., Zhan, N., Zhao, L.: An interface model of software components. In: Proceedings of International Colloquium on Theoretical Aspects of Computing (ICTAC 2013), Lecture Notes in Computer Science 8049. Springer (2013)
21. Faber, J.: A timed model for healthcare workflows based on csp. In: Breu, R., Hatcliff, J. (eds.) SEHC 2012. pp. 1 – 7. IEEE (2012), iSBN 978-1-4673-1843-3
22. Fischer, C.: Combination and Implementation of Processes and Data: from CSP-OZ to Java. Ph.D. thesis, University of Oldenburg (2000)
23. Fowler, M.: Refactoring – Improving the Design of Existing Code. Addison-Wesley, Menlo Park (1999)
24. He, J., Li, X., Liu, Z.: Component-based software engineering. In: Hung, D.V., Wirsing, M. (eds.) Theoretical Aspects of Computing - ICTAC 2005, Second International Colloquium. Lecture Notes in Computer Science, vol. 3722, pp. 70–95. Springer, Hanoi, Vietnam (Oct 2005), http://www.iist.unu.edu/www/docs/techreports/reports/report330.pdf, uNU-IIST TR 330
25. He, J., Li, X., Liu, Z.: A theory of reactive components. Electr. Notes Theor. Comput. Sci. 160, 173–195 (2006)
26. He, J., Liu, Z., Li, X.: rCOS: A refinement calculus of object systems. Theoretical computer science 365(1-2), 109–142 (2006), http://rcos.iist.unu.edu/publications/TCSpreprint.pdf

27. Hoare, C.A.R.: An axiomatic basis for computer programming. Communications of the ACM 12(10), 576–580 (1969)
28. Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall, Upper Saddle River (1985)
29. Hoare, C.A.R., He, J.: Unifying Theories of Programming. Prentice-Hall, Upper Saddle River (1998)
30. Hoenicke, J., Olderog, E.R.: Combining specification techniques for processes, data and time. In: Butler, M.J., Petre, L., Sere, K. (eds.) IFM. Lecture Notes in Computer Science, vol. 2335, pp. 245–266. Springer, Heidelberg (2002), `http://link.springer.de/link/service/series/0558/bibs/2335/23350245.htm`
31. Holzmann, G.J.: The SPIN Model Checker: Primer and reference manual. Addison-Wesley (2004)
32. Holzmann, G.J.: Conquering complexity. IEEE Computer 40(12) (2007)
33. Johnson, J.: My Life Is Failure: 100 Things You Should Know to Be a Better Project Leader. Standish Group International, West Yarmouth (2006)
34. Jones, C.B.: Systematic Software Development using VDM. Prentice Hall, Upper Saddle River (1990)
35. Ke, W., Li, X., Liu, Z., Stolz, V.: rCOS: a formal model-driven engineering method for component-based software. Frontiers of Computer Science in China 6(1), 17–39 (2012)
36. Ke, W., Liu, Z., Wang, S., Zhao, L.: 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. pp. 347–366. Springer, Berlin (2009)
37. Lamport, L.: The temporal logic of actions. ACM Transactions on Programming Languages and Systems 16(3), 872–923 (1994)
38. Lamport, L.: Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers. Addison-Wesley (2002)
39. Larman, C.: Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process. Prentice-Hall, 3rd edn. (2005)
40. Leavens, G.T.: JML's rich, inherited specifications for behavioral subtypes. In: Liu, Z., He, J. (eds.) Proceedings of 8th International Conference on Formal Engineering Methods, volume 4260 of Lecture Notes in Computer Science. pp. 2–34. Springer, Berlin (2006)
41. Leveson, N.G., Turner, C.S.: An investigation of the Therac-25 accidents. IEEE Computer 26(7), 18–41 (1993)
42. Liu, J., He, J.: Reactive component based service-oriented design – a case study. In: In: Proceedings of 11th IEEE International Conference on Engineering of Complex Computer Systems. pp. 27–36. IEEE Computer Society (2006)
43. Liu, Z.: Software development with UML. Tech. Rep. 259, IIST, United Nations University, P.O. Box 3058, Macao (2002)
44. Liu, Z., Joseph, M.: Specification and verification of fault-tolerance, timing, and scheduling. ACM Transactions on Programming Languages and Systems 21(1), 46–89 (1999)
45. Liu, Z., Kang, E., Zhan, N.: Composition and refinement of components. In: Butterfield, A. (ed.) Post Event Proceedings of UTP08, volume 5713 of Lecture Notes in Computer Science. Springer, Berlin (2009)
46. Liu, Z., Mencl, V., Ravn, A.P., Yang, L.: Harnessing theories for tool support. In: Proc. of the Second Intl. Symp. on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2006). pp. 371–382. IEEE Computer So-

ciety (Aug 2006), http://www.iist.unu.edu/www/docs/techreports/reports/report343.pdf, full version as UNU-IIST Technical Report 343
47. Liu, Z., Morisset, C., Stolz, V.: rCOS: theory and tools for component-based model driven development. In: Arbab, F., Sirjani, M. (eds.) 3rd Intl. Symp. on Fundamentals of Software Engineering (FSEN 2009). Lecture Notes in Computer Science, vol. 5961, pp. 62–80. Springer (2009), http://www.iist.unu.edu/www/docs/techreports/reports/report406.pdf, keynote, UNU-IIST TR 406
48. Lynch, N.A., Tuttle, M.R.: An introduction to input/output automata. CWI Quarterly 2(3), 219–246 (1989)
49. Manna, Z., Pnueli, A.: The temporal logic of reactive and concurrent systems:specification. Springer-Verlag (1992)
50. Milner, R.: Communication and concurrency. Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1989)
51. Nielson, H., Nielson, F.: Semantics with Applications. A formal Introduction. Wiley (1993)
52. Object Managment Group: Model driven architecture - a technical perspective (2001), document number ORMSC 2001-07-01
53. Peter, L.: The Peter Pyramid. New York, William Morrow (1986)
54. Plotkin, G.D.: The origins of structural operational semantics. Journal of Logic and Algebraic Programming 60(61), 3–15 (2004)
55. Queille, J.P., Sifakis, J.: Specification and verification of concurrent systems in CESAR. In: Dezani-Ciancaglini, M., Montanari, U. (eds.) Symposium on Programming. Lecture Notes in Computer Science, vol. 137, pp. 337–351. Springer, Heidelberg (1982)
56. Robinson, K.: Ariane 5: Flight 501 failure—a case study (2011), http://www.cse.unsw.edu.au/~se4921/PDF/ariane5-article.pdf
57. Roscoe, A.W.: Theory and Practice of Concurrency. Prentice-Hall, Upper Saddle River (1997)
58. Spivey, J.M.: The Z Notation: A Reference Manual. Prentice Hall, Upper Saddle River, 2nd edn. (1992)
59. Stoy, J.E.: Denotational Semantics: The Scott-Strachey Approach to Programming Language Semantics. MIT Press, Cambridge, Massachusetts (1977)
60. Szyperski, C.: Component Software: Beyond Object-Oriented Programming. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edn. (2002)
61. Vaandrager, F.W.: On the relationship between process algebra and input/output automata. In: LICS. pp. 387–398. IEEE Computer Society (1991)
62. Wang, Z., Wang, H., Zhan, N.: Refinement of models of software components. In: Shin, S.Y., Ossowski, S., Schumacher, M., Palakal, M.J., Hung, C.C. (eds.) SAC. pp. 2311–2318. ACM (2010)
63. Wirsing, M., Banâtre, J.P., Hölzl, M.M., Rauschmayer, A. (eds.): Software-Intensive Systems and New Computing Paradigms - Challenges and Visions, Lecture Notes in Computer Science, vol. 5380. Springer, New York (2008)
64. Woodcock, J., Cavalcanti, A.: The semantics of circus. In: Bert, D., Bowen, J.P., Henson, M.C., Robinson, K. (eds.) ZB. Lecture Notes in Computer Science, vol. 2272, pp. 184–203. Springer (2002)
65. Xiong, X., Liu, J., Ding, Z.: Design and verification of a trustable medical system. In: Johnsen, E.B., Stolz, V. (eds.) Proceedings of 3rd International Workshop on Harnessing Theories for Tool Support in Software, volume 266 of Electronic Notes in Theoretical Computer Science. pp. 77–92. Elsevier (2010)
66. Zhao, L., Liu, X., Liu, Z., Qiu, Z.: Graph transformations for object-oriented refinement. Formal Aspects of Computing 21(1–2), 103–131 (Feb 2009)