Front. Comput. Sci., 2012, 6(1): 17–39
DOI 10.1007/s11704-012-2901-5

RESEARCH ARTICLE

rCOS: a formal model-driven engineering method for component-based software

Wei KE[1,5], **Xiaoshan LI**[2], **Zhiming LIU**(✉)[3], **Volker STOLZ**[3,4]

1 School of Computer Science and Engineering, Beihang University, Beijing 100191, China
2 Faculty of Science and Technology, University of Macau, Macau, China
3 UNU-IIST, Macau, China
4 Department of Informatics, University of Oslo, Oslo 0316, Norway
5 Macao Polytechnic Institute, Macau, China

Abstract Model-driven architecture (MDA) has become a main stream technology for software-intensive system design. The main engineering principle behind it is that the inherent complexity of software development can only be mastered by building, analyzing and manipulating system models. MDA also deals with system complexity by providing component-based design techniques, allowing independent component design, implementation and deployment, and then system integration and reconfiguration based on *component interfaces*. The model of a system in any stage is an integration of models of different viewpoints. Therefore, for a model-driven method to be applied effectively, it must provide a body of techniques and an integrated suite of tools for model construction, validation, and transformation. This requires a number of modeling notations for the specification of different concerns and viewpoints of the system. These notations should have formally defined syntaxes and a unified theory of semantics. The underlying theory of the method is needed to underpin the development of tools and correct use of tools in software development, as well as to formally verify and reason about properties of systems in mission-critical applications. The modeling notations, techniques, and tools must be designed so that they can be used seamlessly in supporting development activities and documentation of artifacts in software design processes. This article presents such a method, called the rCOS, focusing on the models of a system at different stages in a software development process, their semantic integration, and how they are constructed, analyzed, transformed, validated, and verified.

Keywords component-based design, models, model transformations, verification, tool support

Received July 20, 2011; accepted October 8, 2011

E-mail: z.liu@iist.unu.edu

1 Introduction

Software engineering was born with and has been growing because of the "software crisis". The root of the crisis is the inherent complexity of software, and the major cause of the complexity "is that the machines have become several orders of magnitude more powerful" [1] within decades.

1.1 Software complexity

Software complexity is characterized in terms of four fundamental attributes of software [2–5]: the *complexity of the domain application*, the *difficulty of managing the development process*, the *flexibility possible* to offer through software, and the *problem of characterizing the behavior* of software systems [3]. 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. The second concerns the difficulty to define and

manage a development process that has to deal with changing requirements for a software project that involves a large team composed of software engineers and domain experts possibly in different geographical places. The third is about the problem of making the right design decisions among a wide range of possibilities that have conflicting features. 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 pin-points the difficulty in understanding and modeling the semantic behavior of software for software analysis, validation and verification for correctness, and reliability assurance.

We are now facing an even greater scale of complexity with modern *software-intensive systems* [6]. We see the application of these systems in our everyday life, such as in transportation, health, banking, and enterprise applications. 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* and thus the *interoperability* of the distributed components becomes important, for instance, eHealth systems. 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*.

A complex system is open to total breakdown [7], and we suffer from the long lasting software crisis[1]. Consequences of system breakdowns are sometimes catastrophic and very costly, e.g., the famous Therac-25 Accident 1985–1987 [8], Ariane-5 Explosion in 1996 [9], and the yearly cost of about USD 60 billion of the U.S. economy due to software bugs alone[2]. 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 [10]. Given that the global economy, as well as our everyday life, depends on software systems, we cannot give up advancing the theories and engineering methods to master the increasing complexity of software development.

1.2 Formal model-driven development

The model-driven approach, known as *model-driven architecture* (MDA) [6,11,12], proposes as the key engineering principles for mastering software complexity and improving dependability and predictability through building system models in all stages of the system development. Key features of the approach are: abstraction for information hiding in order to focus on a concern at a time, decomposition to divide and conquer, and incremental development so as to allow the use of different techniques and tools. Although there has been a lot of hype about MDA, there is a huge gap between the reality and the potential that model-driven software engineering offers to improve the safety and predictability of software systems. The main reason is that it lacks *systematic techniques* and *tool support* with an integrated theoretical underpinning for the models and their relations. These are needed to make the models analyzable and the engineering process repeatable.

On the other hand, ensuring the correctness of computer systems is the main goal of formal theories and methods. In the past 40 years 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 have been applied in an *ad hoc* manner to modules, algorithms or programs at the source code level. The experience, e.g., in [5], and investigation reports on software failures, such as those of the Therac-25 Accident 1985–1987 [8] and the Ariane-5 Explosion in 1996 [9], show that the cause of a simple bug that can lead to catastrophic consequences is very complex and *ad hoc* application of formal specification and verification to programs or to models of programs at one level of abstraction will not be enough to detect and remove these causes. Formal modeling and verification have to be systematically used in all stages of a development process along with safety analysis that identifies risks, vulnerabilities and consequences of possible risk incidents. In other words, techniques of formal modeling have to be systematically used in a model-driven development process to avoid and remove errors through the tool support in model construction, manipulation (transformation), analysis, and verification.

1.3 The aim and theme of rCOS

The aim of the rCOS project is to research, develop, and teach a method and its tools for predictable development of reliable software. Its scope covers theories, techniques, and tools for modeling, analysis, design, validation, and verification. rCOS

[1] Booch even calls this state of affairs "normal" [3].
[2] See http://www.cse.lehigh.edu/~gtan/bug/softwarebug.html

contributes to *innovation of software engineering* by combining existing theories and techniques, rather than developing new theories and techniques. The rCOS method has a strong theme of component-based and model-driven software development, allowing techniques and tool support to be integrated into engineering processes and have formal theories unified to underpin and justify the development of tools and their integrated applications. 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.

With rCOS we promote the idea that formal methods are not only, or even mainly, for producing software that is safety critical (and so must be shown to satisfy certain properties before being commissioned); they are just as much needed when producing a software system that is too complex to be produced without tool assistance.

1.4 Organization

We present the rCOS approach to handling software complexity in Section 2, thus motivating the different models in rCOS. In Section 3, we discuss the semantic foundation of these models and point the reader to the literature on the relevant theories and techniques. Section 4 defines the models of components, their refinement relations and compositions. Section 5 discusses how the different models are built and validated in a development process. Concluding remarks are given and future work is discussed in Section 6.

2 Mastering complexity with rCOS

This section discusses the main ideas behind the development of the rCOS method that aim at mastering software complexity by separation of concerns, decomposition, and rigorous use of abstraction. Essentially, rCOS proposes a method for model-driven and component-based development techniques and tools to be seamlessly used in a development process.

2.1 rCOS support to model-driven development

The rCOS method follows the fundamental idea of model-driven development that all development activities are organized based on *building models* of software artifacts.

Like in all mature engineering disciplines, system models are built to gain understanding and confidence in requirements and designs.

Furthermore, rCOS provides explicit means through its modeling notations, refinement calculus and tools to support the following principles of model-driven development and formal methods.

- *Abstraction* for information hiding when building a model so as to be well-focused and problem oriented with regard to the concerns at the development stage. As shown by the vertical dimension in Fig. 1, the models have different levels of abstraction at different stages, and they are related by the formally defined *refinement relation* [13–15] among models to ensure *correctness by design*.
- *Decomposition* to divide and conquer for incremental and compositional design and analysis. At any level of abstraction, the model of the system is a composition of a number of components at a lower level. A component at a higher level is designed as a composition of a number of components at a lower level [13,16]. The architectural (or hierarchical) dependency among components is represented by the horizontal dimension of Fig. 1.
- *Separation of concerns* to allow the separation of design or implementation of different viewpoints of a component at a certain level of abstraction. This is achieved through multi-view modeling. At any level of abstraction, the model of a component consists of a consistent set of viewpoint models, including the model of types (and classes when OO is used), static model of interface data functionality, model of interactions with the environment, reactive behavior model, etc. (see the third dimension of Fig. 1). These different views are modeled in different notations, Hoare-logic [17] for data functionality, sequence diagrams for interactions and state

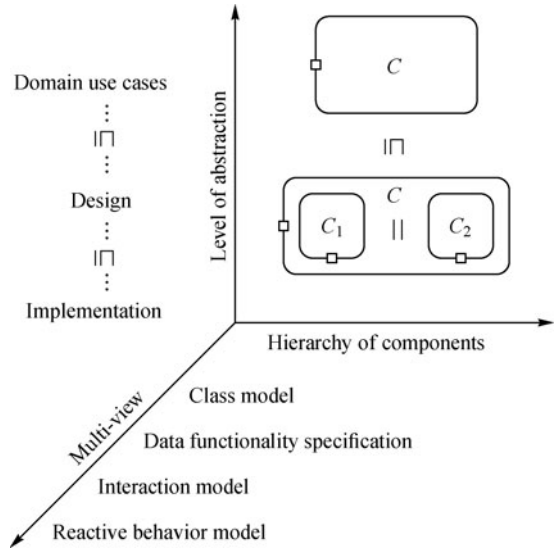


Fig. 1 rCOS modeling framework

machines for reactive behavior, using different techniques and tools for reasoning and verification [14,18]. The consistency of the models of these viewpoints is defined and can be checked automatically. This is the key for rCOS to support the integrated use of different techniques and tools in a design process [19,20].

- *Model transformations* to automate refinement of models from one level to another for correct design, and abstraction of models for reasoning and verification. A model transformation may generate *proof obligations*, i.e., conditions on the models before and after the transformation. Such a transformation thus brings in techniques and tools of reasoning and verification for proving these obligations.
- *Use of formalization* or rigor for the models to be analyzable and process repeatable. This requires a unified semantic theory for underpinning the integrated tool support to different techniques used on the different viewpoint models [21–23].

In summary, the discussion shows that in the rCOS method, it is as important for the development and usage of tools as for the correct construction of models (and programs). In an rCOS design process, model transformations automate design activities, driving the process and linking the different tools for reasoning and verification of properties of the models. We will later, in Section 5, show the main model transformations in an rCOS design process.

2.2 rCOS support to component-based development

In rCOS, we explicitly specify the contracts of the interfaces of components [11] so that

- Components can be used in third party compositions, without the need to know the design and implementation details. Components are designed and implemented to be deployed independently.
- The design, proofs, and code of components can be reused in different applications.

The software architecture is always important in model-driven development and therefore model-driven design is known as model-driven architecture. To support seamless model-driven development, rCOS provides a component-based architecture definition language, called rCOSP. The static architectural view of the model of a component is shown in Fig. 2, and its behavior is defined later. However, only when the ideas, principles, techniques, and tools for component-based modeling are integrated into a model-driven development process, that defines a flow of activities of model constructions, transformations, and validation, the software complexity can be effectively handled and the quality of the software can be improved (see Section 5).

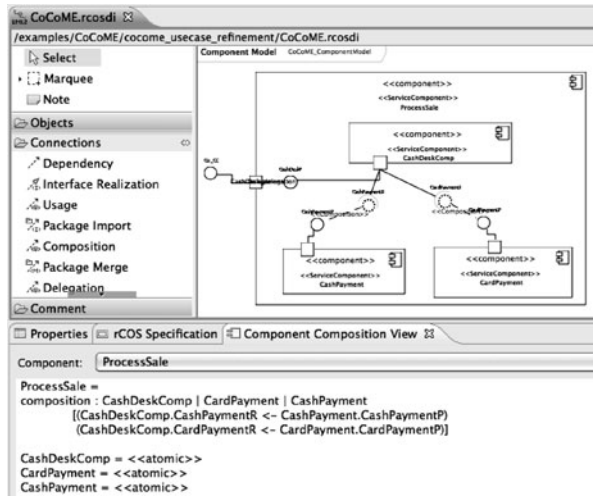


Fig. 2 rCOS component architecture — static view

2.3 The format of component specifications in rCOSP

Modern programming languages provide structures and mechanisms that are abstract enough to se used in formal specifications. For example, the class structures, data and behavior encapsulation and inheritance have the same expressive power as the schemas, classes or modules, and the import mechanisms in Z [24] and VDM [25] specification languages. Similar to JML [26], rCOSP proposes a Java-style specification language. But it allows specifications of components as well as classes, predicative specification of functionality as well as program commands and interactions and reactive behavior specification. The obvious advantage is that it is understandable to both formal method people and practical software engineers.

The format of a component specification in rCOSP is shown in Fig. 3, and it is explained below.

In rCOSP, the architectural operators of “parallel composition”, “plugging”, “disjoint union”, and “coordination” are defined [13,16]. Their semantics is defined in rCOS and the laws are proven in rCOS calculus. In the rest of this paper, we will concentrate on the rCOS semantics and calculus.

• ***Interfaces of components*** The provided interface declares a list of methods or services that can be invoked or requested by clients. The required interface is optional and when it is empty the specified component is called a *closed component*, otherwise an *open component*. Both provided and

```
component K{
    Tx=c; // state of component
    provided interface I {
        // provided methods
        m1(parameters) {/* functionality definition */}
        m2(parameters) {/* functionality definition */}
        …
        m(parameters) {/* functionality definition */}
        protocol {?m1(?m2+?m3)* }// regular expression
    }
    actions  {// no parameters and autonomous actions
        a1 {g1 &c1}
        …
        ak {gk &ck}
    }
    internal interface {// locally defined methods
        m1(parameters) {/* functionality definition */}
        m2(parameters) {/* functionality definition */}
        …
        m(parameters) {/* functionality definition */}
    }
    requiredinterface J {// required services
        Ty=d ;
        n1(parameters), n2(parameters), n3(parameters)
        protocol {!n1(n2n3+n3n2)* }
    }
    class C1{…}; class C2{…}; …// used in the above specification
}
```

Fig. 3 Format of rCOS components

required interfaces allow definitions of state variables. A component can also have an internal interface that declares the methods private to the component, and thus they can only be called by the provided methods, the private methods themselves, and the actions (to be defined later).

• ***Data functionality*** The data functionality of each provided method and private method is defined in their bodies, and its specification is given in the rCOS OO specification language. The rCOS OO specification language specifies the functionality of a method at different levels of abstraction, from predicate specification in the form of unifying theories of programming (UTP) designs

$$precondition \vdash postcondition$$

to program code [22]. The specification

$$m() \{ precondition \vdash postcondition \}$$

means that when method $m()$ is invoked, if the input parameters and the current state of the component satisfy the *precondition*, the execution of the invocation will terminate and the final state satisfies the *postcondition*. At the level of program code, invocations of the methods in the required interface, as well as those in the provided interface and the private methods, can occur in the bodies of the provided methods. The private methods of the interface class form the *internal interface* of the component.

• ***Actions*** In a component, internal autonomous actions are allowed and they are specified as guarded designs and implemented as guarded commands, which are formally defined in Section 3. Such an action can be executed atomically in a state in which its guard holds.

• ***Interaction protocols*** A component is *reactive* and controls access to the services it provides. In the above specification format, the interaction protocol of the provided interface specifies the constraints on the order in which the environment calls the provided methods. It is specified as a set of traces of interactions defined by regular expressions. It is known that regular expressions have limited expressive power. In general, languages generated by finite state machines with guarded transitions, traces of CSP processes [27,28], or interface automata [29] can be used. Similarly, the protocol of the required interface specifies the requirements on how the required component should provide the services. Obviously, the required protocol is determined by the protocol of the provided interfaces and its functionality is implemented by the interface class.

• ***Class structure and data types*** Fields of component interfaces can be of either primitive data types (integers, Booleans, etc.) or classes in OO programming languages. The declarations at the bottom of Fig. 3 define such classes, and they are formalized as the class declaration section in rCOS, and can be represented graphically by a UML class diagram.

Example 1 The specification below specifies the behavior of a memory cell. It provides two methods for writing a value to and reading the content from the memory cell of type *Z*, requiring that the first operation has to be a write operation.

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

To illustrate some of the architectural operators of rCOS, we consider a faulty memory cell fM in which a fault may autonomously occur to corrupt the content of the memory. For this, we specify an internal autonomous action $fault$ that corrupts the contents.

```
component fM {
  provided interface fMIF {
    Z d;
```

```
    W(Z v) { d:=v }
    R(; Z v) { v:=d }
    actions { fault { b'≠b } }
    protocol { ?W({?W,?R})* }
  }
}
```

To build a fault-tolerant implementation of the memory M using three faulty memory cells, we design the majority voting component that is an open component.

```
component V { // a connector
  provided interface VIF {
    W(Z v) { fM1.W(v); fM2.W(v); fM3.W(v) }
    R(; Z v) { // majority voting
      v:=vote(fM1.R(v), fM2.R(v), fM3.R(v))
    }
    protocol { ?W({?W,?R})* }
  }
  required interface { // union of fM1, fM2, and fM3
    protocol { /* interleaving of all fMi's protocols */ }
  }
}
```

We use renaming to define additional faulty memory cells fM_i as

$$fM[fM_i.W/W, fM_i.R/R].$$

We compose the three faulty memory components $fM_1 \parallel fM_2 \parallel fM_3$ and then plug the composed component into component V to obtain the fault-tolerant component $V \ll fM_1 \parallel fM_2 \parallel fM_3$. Fig. 4 shows the architecture. The verification of fault-tolerance requires the assumption that at any time the content of at most one of the faulty memory cells is corrupted, and this requires use of auxiliary variables in the specification [30].

Note that autonomous state changes can happen. These active internal actions are the effect of a coordination implemented by an active process that keeps calling enabled methods of the component. We allow the explicit specification of active processes (see Section 4) that are used to pass data between components and coordinate the behavior of components through synchronization [14,16]. In fact, we could have specified the fault action $fault$ as a method $fault()$ in the provided interface of fM, and then specify a process $Fault$ that keeps invoking $fault()$. The effect of this is defined by the coordination composition $fM \odot Fault$ in Section 4.

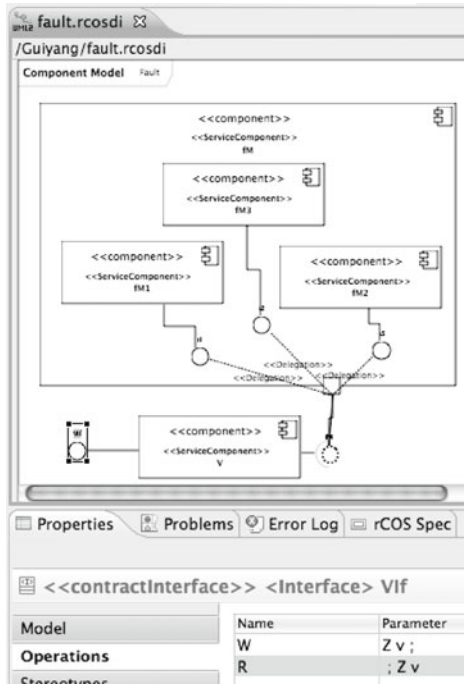


Fig. 4 rCOS component architecture of fault-tolerant memory

3 Unifying theories for component-based software modeling

The rCOS method supports programming software components for different systems, including sequential, concurrent, distributed, and reactive systems. It allows different programming paradigms too, including modular (procedural), object-oriented, and service-based programming. The model of a component is separated into a number of related models of different concerns, including static structure, static data functionality, interaction protocol, and dynamic control behavior. This separation of concerns is crucial to a) control the complexity of the models, and b) allow the use of different techniques and tools of modeling, analysis, design, and verifications appropriate for the different models. This requires a unified semantic theory of models of programs.

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 the lower level *refined models*. Therefore, the unified modeling theory must be equipped with a *refinement calculus*. The rCOS theory is based on the UTP [31]. This section presents this theory and its use in linking different theories.

3.1 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 syntax below:

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

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 that given initial values for 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 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 $\mathscr{X}$ representing *state variables* with an associated value space $\mathscr{V}$. We define a *state* of $\mathscr{X}$ as a function $\sigma : \mathscr{X} \to \mathscr{V}$ and use $\mathscr{S}$ to denote the set of all states of $\mathscr{X}$. This allows us to study all the programs written in our language. For a subset X of $\mathscr{X}$, we call the restrictions $\mathscr{S}|_X$ of $\mathscr{S}$ on X the states of 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 $\mathscr{S}|_{\alpha P}$ the *states of program P*.

• ***State properties and state relations*** A state property is a subset of the states $\mathscr{S}$ and can be specified by a predicate over $\mathscr{X}$, called a *state predicate*. For example, $x > y + 1$ defines the set of states σ such that $\sigma(x) > \sigma(y) + 1$ holds. We say that a state σ satisfies a predicate F, denoted by $\sigma \models F$, if it is in the set defined by F.

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

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

holds, denoted by $(\sigma, \sigma') \models R$. 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, $x' \geqslant x + 1$ 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, state predicate F can also be interpreted as a relational predicate such that F holds for (σ, σ') if σ 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

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

where x and x' represent the set 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 (σ, σ') satisfies $R_1; R_2$ iff there exists a state σ_0 such that (σ, σ_0) satisfies R_1 and (σ_0, σ') 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 what 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 (Designs) 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 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 σ satisfying p and if its execution terminates, it terminates in a state σ' such that $(\sigma, \sigma') \models R$.

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 following operations on designs over the same alphabet.

$$\begin{aligned}
&x := e && \mathrel{\hat{=}} true \vdash x' = e \textstyle\bigwedge_{y \in \alpha, y \neq x} y' = y,\\
&c_1; c_2 && \text{is defined by Equation (2),}\\
&c_1 \lhd b \rhd c_2 && \mathrel{\hat{=}} b \wedge c_1 \vee \neg b \wedge c_2,\\
&c_1 \sqcap c_2 && \mathrel{\hat{=}} c_1 \vee c_2,\\
&b * c && \mathrel{\hat{=}} (c; b * c) \lhd b \rhd \mathbf{skip},
\end{aligned} \tag{3}$$

where we have $\mathbf{skip} \mathrel{\hat{=}} true \vdash \bigwedge_{x \in \alpha}(x' = x)$. Also we have $\mathbf{chaos} \mathrel{\hat{=}} false \vdash true$.

However, for the semantics definition to be sound, we need to show that the set $\mathscr{D}$ of designs is closed under the operations defined in Equation (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 $\mathscr{D}$.

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

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

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) weakening the precondition: $p_h \Rightarrow p_l$, and
2) strengthening the postcondition: $p_l \wedge R_l \Rightarrow R_h$.

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

Theorem 2 The set $\mathscr{D}$ of designs and the refinement relation $\sqsubseteq$ satisfy the following properties:

1) $\mathscr{D}$ is closed under the sequential composition ";", conditional choice " $\lhd\, b \,\rhd$ " and non-deterministic choice "$\sqcap$" defined in Equation (3),
2) $\sqsubseteq$ is a partial order on the domain of designs $\mathscr{D}$,
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) $(D, \sqsubseteq)$ forms a CPO and the recursive equation $b * D = (D; b * D) \lhd b \rhd \mathbf{skip}$ has a smallest fixed point and which may be calculated from the bottom element **chaos** in $(D, \sqsubseteq)$.

For the proof of the theorems, we refer to the book on UTP [31]. 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 $D_1 \sqcap D_2 = D_2$ iff $D_1 \sqsubseteq D_2$. Therefore, the relation $\sqsubseteq$ is fundamental for the development of the refinement calculus to support correctness 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 variables implemented by a list. We may also compare designs given by different programmers. Then 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 $\mathscr{T}(p \vdash R)$ defined below:

$$\mathscr{T}(p \vdash R) \mathrel{\hat{=}} ok \wedge p \Rightarrow ok' \wedge R.$$

This predicate describes the execution of a program in that if the execution starts successfully ($ok = true$) in a state σ such that precondition p holds, the execution will terminate ($ok' = true$) in a state σ' for which $R(\sigma, \sigma')$ holds. A design D is called a *complete correctness design* if $\mathscr{T}(D) = D$. Notice that $\mathscr{T}$ is a *healthy lifting function* from the domain $\mathscr{D}$ of partially correct designs to the domain of complete correct designs $\mathscr{T}(\mathscr{D})$ in that $\mathscr{T}(\mathscr{T}(D)) = \mathscr{T}(D)$. The refinement relation can be lifted to the domain $\mathscr{T}(\mathscr{D})$, and Theorem 1 and 2 both hold. For details of UTP, we refer to the book [31]. In the rest of the paper, we assume the complete correctness semantic model, and omit the lifting function $\mathscr{T}$ in the discussion.

• ***Linking theories*** We can unify the theories of Hoare-logic [17] and the predicate transformer semantics of Dijkstra [32]. The Hoare-triple $\{p\}D\{r\}$ of a design D 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 version.

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.

3.2 Designs of object-oriented programs

We emphasize the importance of a semantic theory for concept clarification, development of techniques and tool support for correcting by design and verification. The semantic theory presented in the previous subsection 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 of related objects. 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 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 a 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 [15,22], a graph-based operational semantics of OO programs [23] and a refinement calculus [15] 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 the general reader.

• ***OO specification in rCOS*** The rCOS OO specification language is defined in [22]. 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:

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

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 can 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 program 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*:

class *Main* { **private** *vars*; **method** *main*(){*c*} }

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

- expressions: $e ::= x \mid a \mid null \mid this \mid e.a \mid (C)e \mid f(e)$
- assignable expressions: $le ::= x \mid e.a$
- commands: $c ::=$ **skip** | **chaos** | **var** $T\ x;\ c;$ **end** $x \mid$
  $c; c \mid c \lhd b \rhd c \mid c \sqcap c \mid b * c \mid$
  $e.m(e; le) \mid le := e \mid C.new(le)$

Notice that the creation of a new object $C.new(le)$ is a command not an expression. It returns in le the object newly created and plays the same role as $le = new\ C()$ 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 rep-

resent object identities. An object also has a runtime type. Thus, we define an object by a triple $o = (r, C, \sigma)$ of its identity r, runtime type C, and state σ. The state σ is a *typed function*

$$\sigma : \mathscr{A}(C) \to \mathscr{O} \cup \mathscr{V},$$

where

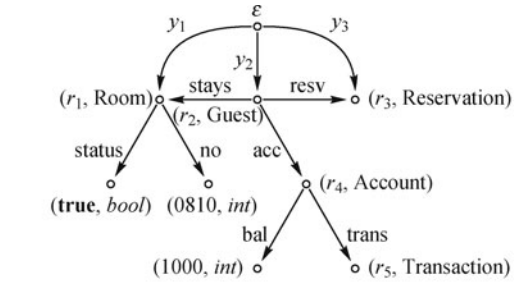
- $\mathscr{O}$ is the set of all objects of all classes,
- $\mathscr{V}$ the value space of all the primitive data types,
- $\mathscr{A}(C)$ is the set of names of the attributes of C, including those inherited from all its superclasses, and
- σ maps an attribute a to an object or value of the type 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* [15,23], 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 [15], 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 labels for the attributes form a *class graph* that is called the *type graph* of the object that the object graph represents [15,23].

• ***States of programs*** Given an OO program $P = ClassDecls \bullet Main$, a *global state* of P is defined as a mapping $\sigma : vars \to \mathscr{O} \cup \mathscr{V}$ that assigns each variable $x \in vars$ 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. 5 is a global state of the accompanied program outline, and its type graph (and the corresponding UML class diagram) is given in Fig. 6.




```
program Hotel{
 class Person{
   public Accountacc = null;
   method m1( ) {...}
 }
 class Guest extends Person{
   public
     Room stays = null,
     Reservationresv = null;
   method
     makeReservation( ) {...}
     checkOut( ) {...}
 }
 class Account{
   public int bal = 0;
   method checkBalance( ) {...}
 }
```

```
class Room{
  public
    int no=0,
    bool status = false;
  method changeStatus( ) {...}
}
class Transaction{
  ...
}
class Main{
  private
    Room y1, Guest y2,
    Reservation y3;
  method main( ) {...}
}
} //end of program
```

Fig. 5 An example of object graph

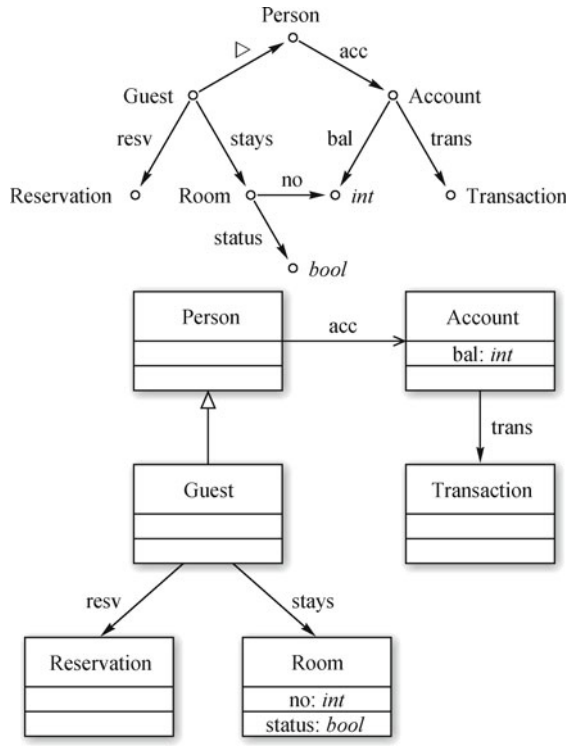


Fig. 6 An example of class graph and diagram

Global states are enough for defining a UTP-based denotational semantics [22] 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 the 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 labeled by \$ to the root n of G and another outgoing edge labeled by x to x_0. 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 7 shows an example of a state graph that characterizes the local scopes below:

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

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

• ***Semantics*** We explain the semantics informally, but precisely. Both a denotational semantics and an operational semantics can be defined by what changes the execution of a command makes on a given state graph. This can be easily 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';

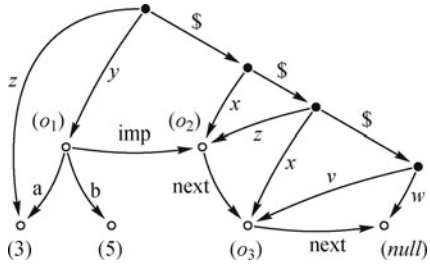


Fig. 7 An example of state graph with local scopes

- *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

$$\begin{aligned}&\textbf{var}\ this = e, in = e_1, y^* = le, return;\\ &c; y^*.a := return;\\ &\textbf{end}\ this, in, y^*, return\end{aligned}$$

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 [22] and the graph-based operational semantics is given in [23].

• ***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 between commands takes exactly the same view as in the previous subsection about traditional sequential programs that 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 the **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*.

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 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 $Main_h$.

An essential advantage of OO programming is that classes can be reused in different applications that are programmed by 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 and 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 4 (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 paper [15], 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 [33] 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 [34]. The use of these patterns is an essential practice in OO program design [14]. The combination of class graph transformations and command transformations is illustrated in Fig. 8. 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 and transformation ρ_c on commands. Then ρ is a class refinement if the diagram commutes for all OG of CG and all commands c.

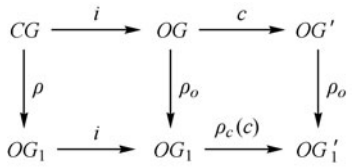


Fig. 8 Class graph transformations and command transformations

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 [15].

3.3 Concurrent programs

The programs that have been considered so far in this section are sequential programs. For such a program, our semantic definition only concerns the relation between the initial and final states and termination. In general, a concurrent program consists of a number of processes, each executes a sequence of *actions* of computation. However, these processes interact with each other to exchange data and to synchronize their behavior. Termination of the processes is usually not a required property, though the *enabling condition* and *termination* of execution of individual actions are essential for the execution of all processes to make progress, that is, to have no *livelock* or *deadlock*.

There are two different paradigms of programming interaction and synchronization, shared memory-based programming and message-passing programming. We define a semantics for concurrent programs with shared variables and for reactive programs with message passing in Section 4.

• ***Concurrent program with shared variables*** In general, a concurrent program can be considered as a set of *atomic actions* programmed in a concurrent programming language that change a set of variables. The set of actions can be portioned into a number of processes [30,35] that can be executed on a single processor system through a scheduler or a multiple processor system.

Like a sequential programming command, the execution of an atomic action changes the current state of the program to another state. This *data functionality* of an action can thus be specified as a design $p \vdash R$. However, the execution requires resources that might be occupied by other processes or a synchronization condition. Then the execution is suspended in a *waiting* state. For allowing the observation of the waiting state, we introduce the Boolean state variables *wait* and *wait′* and define the following lifting function on designs:

$$\mathscr{H}(D) \mathrel{\hat{=}} wait' \lhd wait \rhd D,$$

meaning that no more execution can proceed in a waiting state. Then, we call a design D a *reactive design* if $\mathscr{H}(D) = D$. Notice that $\mathscr{H}(\mathscr{H}(D)) = \mathscr{H}(D)$.

Theorem 3 (Reactive designs) The domain of reactive designs has the following closure properties:

$$\mathscr{H}(D_1 \vee D_2) = \mathscr{H}(D_1) \vee \mathscr{H}(D_2),$$

$$\mathscr{H}(D_1; D_2) = \mathscr{H}(D_1); \mathscr{H}(D_2),$$
$$\mathscr{H}(D_1 \lhd b \rhd D_2) = \mathscr{H}(D_1) \lhd b \rhd \mathscr{H}(D_2).$$

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

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

Theorem 4 If D is a reactive design, so is $g \;\&\; D$.

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

Theorem 5 (Guarded designs) For guarded designs, we have

$$g_1 \;\&\; D_1 \vee g_2 \;\&\; D_2 = g_1 \vee g_2 \;\&\; ((D_1 \vee D_1) \lhd g_1 \wedge g_2 \rhd (D_1 \lhd g_1 \rhd D_2)),$$
$$g_1 \;\&\; D_1 \lhd b \rhd g_2 \;\&\; D_2 = (g_1 \lhd b \rhd g_2) \;\&\; (D_1 \lhd b \rhd D_2),$$
$$g_1 \;\&\; D_1; g_2 \;\&\; D_2 = g_1 \;\&\; (D_1; g_2 \;\&\; D_2).$$

Corollary 1 For the disjunction and sequential composition, we have

$$g \;\&\; D_1 \vee g \;\&\; D_2 = g \;\&\; (D_1 \vee D_2),$$
$$g \;\&\; D_1; D_2 = g \;\&\; (D_1; D_2).$$

This theorem is important as it lays down the foundation for defining concurrent programming languages. It says that in general a concurrent program P is a set of atomic actions and each action is a *guarded command* $g \;\&\; c$ where c is a command in the following syntax:

$$c ::= x := e \mid c; c \mid c \lhd b \rhd c \mid c \sqcap c \mid g \;\&\; c \mid b * c, \qquad (4)$$

where the semantics of a command is defined inductively from

$$x := e \mathrel{\hat{=}} \mathscr{H}(true \vdash x' = e \wedge_{y \in \alpha, y \notin x} y' = y).$$

The semantics and reasoning of concurrent programs written in such a powerful language is quite complicated. The semantics of an atomic action is 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* [36], a UNITY program [35], and a TLA specification [37].

A concurrent program can be represented as

$$P = (vars, init, A),$$

where $vars$ is the set of program variables (not including ok and $wait$), $init$ the initial condition defining the allowable initial states, and A the set of atomic actions. For a state σ over $vars \cup \{ok, wait\}$, an action $a \in A$ is said to be *enabled* at σ if $a[\sigma(x)/x] \Rightarrow wait' = false$, *disabled* otherwise.

Definition 5 (Semantics of concurrent programs) Let $P = (vars, init, A)$ be a concurrent program. The *semantics* of P is defined by a pair $(divergence(P), failure(P))$ of *divergences* and *failures*, where

1) A divergence in $divergence(P)$ is a finite execution sequence $\sigma_0 \xrightarrow{a_1} \sigma_1 \xrightarrow{a_2} \cdots \xrightarrow{a_n} \sigma_n$ where $n \geqslant 0$ and σ_i $(0 \leqslant i \leqslant n)$ are states over $vars \cup \{ok, wait\}$, such that there exists $0 \leqslant k \leqslant n$ and for all $0 \leqslant i \leqslant k$,
   (a) σ_0 is an allowable initial state, $\sigma_0 \models init$,
   (b) for any state transition $\sigma_{i-1} \xrightarrow{a_i} \sigma_i$, a_i is enabled at σ_{i-1},
   (c) the state change made by each state transition $\sigma_{i-1} \xrightarrow{a_i} \sigma_i$ conforms to the specification of the action a_i, i.e., changes the state according $(\sigma_{i-1}, \sigma_i) \models a_i$, and
   (d) there exists $k \leqslant n$, $\sigma_0 \xrightarrow{a_1} \sigma_1 \cdots \sigma_{k-1} \xrightarrow{a_k} \sigma_k$ and σ_k is a divergent state, i.e., $\sigma_k(ok) = false$.
2) The set $failure(P)$ contains all the pairs (tr, X) where tr is a finite execution sequence of P and $X \subseteq A$ such that one of the following conditions holds
   (a) tr is empty denoted by ε and there exists an allowable initial state σ_0 such that a is disabled at σ_0 for any $a \in X$,
   (b) $tr \in divergence(P)$ and X can be any subset of A,
   (c) $tr = \sigma_0 \xrightarrow{a_1} \cdots \xrightarrow{a_k} \sigma_k$ and for any σ in the sequence, $\sigma(ok) = true$ and $\sigma_k(wait) = false$, and a is disabled at σ_k.

The semantics takes both traces and data into account. X in $(tr, X) \in failure(P)$ is called a set of *refusals* after the execution sequence tr. We call the subset $Trace(P) = \{tr \mid (tr, <>) \in failure(P)\}$ of execution traces the *normal execution traces*.

Definition 6 (Refinement of concurrent systems) Let $P_l = (vars, init_l, A_l)$ and $P_h = (vars, init_h, A_h)$ be two pro-

grams. P_l is a *refinement* of P_h if

$$init_l \implies init_h,\ g(A_l) \iff g(A_h),\ next(A_l) \implies next(A_h),$$

where $g(A_l)$ and $g(A_h)$ are the disjunctions of the guards of the actions of A_l and A_h, respectively, and

$$next(A_l) = \bigvee_{a \in A_l} guard_l(a) \wedge body_l(a),$$
$$next(A_h) = \bigvee_{a \in A_h} guard_h(a) \wedge body_h(a),$$

and $guard_l(a)$ and $body_l(a)$ (or $guard_h(a)$ and $body_h(a)$) denote the guard and the body of actions a in A_l (or A_h), respectively.

The first condition ensures that the allowable initial states of P_l are allowable for P_h; the second ensures that P_l is not more likely to deadlock; and the third guarantees that P_l is not more non-deterministic, thus not more likely to diverge, than P_h. Notice that we cannot weaken the guards of the actions in a refinement as otherwise some safety properties can be violated. This semantics unifies the theories of refinement in [30,35–37].

Example 2 We give an example of a concurrent program which models the interaction between the memory component in Example 1 and the processor controller.

```
program Memory-Processor {
 Z v, d;
 bool start = false, write = true, read = false, ready = true;
 actions {
  Wm { write & (d:=v; ready:=true) } // memory executes write
  Rm { read & (v:=d; ready:=true) } // memory executes read
  Wp {
   ready & (if ¬ start then start:=true; (v' in Z); write:=true)
  } // requests a write
  Rp { (start ∧ ready) & (read:=true) } // requests a read
 }
}
```

Design and verification of concurrent programs are challenging and the scalability issue 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.

4 Models of components

In general, a reactive program interacts with its environment. Such a program K has two kinds of actions, the *interface actions* $K.IF$ and the *internal actions* $K.iA$. Both kinds of actions change the state of the program when they are executed. However, an internal action can be executed autonomously when it is enabled, but the environment and program K must be *synchronized* on the execution of an interface action. There are quite a few programming paradigms such as channel-based and method invocation-based programming. There are mainly two kinds of abstract models of reactive program, channel/event-based processes [27,38] algebras and I/O automata [39]. All these models provide abstract representation of interaction actions, but when message passing is involved the relation between input and output values are modeled at a low level of granularity. For example in CSP [27] or CCS [38], input is represented as $c?x$ (or $c(x)$ in CCS) and output by $c!(e)$ (or $\overline{c}(e)$, respectively). This has a limited scalability when modeling programs with synchronized method invocations.

In rCOS, we define two kinds of reactive programs, that we call *components* and *processes*. A component provides services represented as methods to be called by the environment. However, it can also have internal actions that change its state. Processes do not provide services but only make invocations to components following its own flow of control so as to coordinate the behavior and pass data among components. Processes implement business workflows.

4.1 Components

We first consider *closed components* that only provide services but do not require services from other components. Such a component K has a set $K.F$ of variables (also called *fields*), a provided interface $K.pIF$, an internal set of autonomous actions $K.iA$, and a set $K.iIF$ of private methods called the internal interface such that

- An initial condition $K.init$ defines the allowable initial states of the component,
- An interface, either $K.pIF$ or $K.iIF$, is a set of method signatures of the form, $m(in; out)$ with possible input or output parameters that are typed,
- Each method signature $m()$ is given a body that is a guarded command written in Syntax (4), extended with method invocations $n(e; y)$, where $n()$ is in $K.pIF \cup K.iIF$,
- Each action name a in $K.iA$ is given a body that is a guarded command (so an action can be seen as a method without parameters).

It is required that $K.pIF \cap K.iIF = \emptyset$.

Informally speaking, a component K repeatedly accepts requests (invocations) from the environment to its enabled services (or methods) in $K.pIF$ to execute, or executes an

enabled action of its own, until no provided methods or actions that are enabled. The formal abstract semantics, however, only concerns the interaction behavior of K with the environment. We define an *observable transition relation*, denoted by $\sigma \xrightarrow{\overline{e}} \sigma'$, for an event in pIF, if there exists a sequence $a_1 \dots a_n$ (possibly empty) of events in $K.iA$, such that $\sigma \xrightarrow{e} \sigma_1 \xrightarrow{a_1} \cdots \xrightarrow{a_n} \sigma'$.

Definition 7 (Semantics of components) The semantics of K is again defined by a pair of *failure* set $failure(K)$ and *divergence* set $divergence(K)$, where

- $divergence(K)$ is a set of traces over method invocations $m(v;u)$, where $m() \in K.pIF$ such that there is a sequence of state transitions $\sigma_0 \xrightarrow{\overline{e}_1} \cdots \xrightarrow{\overline{e}_k} \sigma_k$ and $\sigma_k(ok) = false$.
- $failure(K)$ consists of the pairs (tr, X), where $tr = a_1 \dots a_n$ $(n \geqslant 0)$ is a trace and X a set of invocations $m(v;u)$ to methods in $K.pIF$, such that one of the following conditions holds.
  1) The trace is a divergence, i.e., $tr \in divergence(K)$ and X is any set of invocations to methods in $K.pIF$.
  2) There exists an execution sequence $\sigma_0 \xrightarrow{\overline{a}_1} \cdots \xrightarrow{\overline{a}_n} \sigma'$ such that there exists an internal action $a \in K.iA$ enabled at σ' and at least one invocation to the provided methods in pIF is enabled. In this case σ' is a stable state in which not only internal actions are enabled, and X can be any set of invocations to methods in $K.pIF$.
  3) There exists an execution sequence $\sigma_0 \xrightarrow{\overline{a}_1} \cdots \xrightarrow{\overline{a}_n} \sigma'$ such that there exists no internal action enabled at σ' and all actions in X are disabled.
  4) There exists an execution sequence $\sigma_0 \xrightarrow{\overline{a}_1} \cdots \xrightarrow{\overline{a}_n} \sigma'$ such that only internal actions are enabled at σ'. In this case σ' is a livelock (or deadlock if no action enabled at all), and X can be any set of invocations to methods in $K.pIF$.

Example 3 The memory component M in Example 1 can be defined by the initial condition $init = \neg start$, then $M.pIF = \{W(Z\ v), R(;\ Z\ v)\}$ such that

$$W(Zv)\{\mathbf{true}\&(d := v; start := \mathbf{true})\}$$
$$R(;Zv)\{start\&v := d\}$$

The fault-prone memory fM is obtained from M by adding the internal action $fault$.

4.2 Refinement between closed components

Refinement between two components K_h and K_l compares the services that they provide to the clients.

Definition 8 (Refinement between components) A component K_l is a refinement of a component K_h, denoted by $K_h \sqsubseteq K_l$ if $K_h.pIF = K_l.pIF$ and

1) K_l is not more likely to block clients,
$$failure(K_l) \subseteq failure(K_h),$$
2) K_l is not more likely to diverge,
$$divergence(K_l) \subseteq divergence(K_h).$$

We define the provided traces of a component K

$$trace(K) = \{tr \mid (tr, \{\}) \in failure(K)\}.$$

Notice that because of nondeterminism caused by internal actions and non-deterministic functionality of actions, some of the traces are non-deterministic that the client can still get blocked event it interacts with K following such a trace. Therefore $trace(K)$ cannot be used as the provided protocol for the client to use. We call a trace $tr = a_1 \dots a_n$ *non-blocking* or *input deterministic* if for any of its prefix $tr' = a_1 \dots a_k$, there exists no set X of provided events of K such that $a_{k+1} \in X$ and $(tr', X) \in failure(K)$.

Definition 9 (Provided protocol of components) The provided protocol of a component K, denoted as $protocol(K)$ is the set of all its non-blocking traces.

Obviously, if $K_h \sqsubseteq K_l$, then $protocol(K_h) \subseteq protocol(K_l)$.

There is a special class of components, such as the one in Example 3, that have no internal actions, i.e., $K.iA$ is empty. They are called *primitive component models*. In our earlier work [16], we proved that in general every closed component model is equivalent to a primitive component model. This is easy to see if the flow of control of the component forms a finite state machine as the finite number of internal actions from a state can be collapsed into their proceeding transitions of provided actions. This result implies that we have two forms of interface behavior model of a component.

Definition 10 (Contract) A *component contract* is just like a primitive component C, 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 that is the vehicle of model checking.

From the behavioral semantics of a contract C defined by Definition 7, we obtain the following model interface behavior.

Definition 11 (Publication) A *component publication* $B = (F, init, pIF, pProt)$, where the body of each method m in pIF is just a design $p_m \vdash R_m$ of the data functionality and $pProt$ is a set of traces that defines the provided protocol of B.

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 specification of components in system synthesis.

Theorem 6 (Component refinement) Let

$$K_h = (init_h, pIF_h, iIF_h) \quad \text{and} \quad K_l = (init_l, pIF_l, iIF_l)$$

be two primitive components. $K_h \sqsubseteq K_l$ iff $pIF_h = pIF_l$ and for all $m \in pIF_h$,

$$guard_h(m) = guard_l(m) \wedge body_h(m) \sqsubseteq body_l(m).$$

4.3 Open components and component composition

We can put two closed components K_1 and K_2 in parallel, denoted by $K_1 \parallel K_2$, so that a client can use the provided services of the two components in an interleaving manner. We can also construct components that provide new services using the services provided by a component. Such a new component K provides services through its provided interface $K.pIF$, but to deliver a service $m() \in K.pIF$, it can request services from other components. Similarly, an internal action in $K.iA$ and a private method in $K.iIF$ may also request services. The requests to these services are made through the *required interface* $K.rIF$. Obviously, it is required that $K.rIF$ is disjoint with $K.pIF$ and $K.iIF$. A component with a non-empty required interface is called an *open component*.

Definition 12 (Component composition) Let K_1 and K_2 be two components such that

$$K_1.F \cap K_2.F = \emptyset,\ K_1.pIF \cap K_2.pIF = \emptyset, K_1.iA \cap K_2.iA = \emptyset,$$
$$(K_1.rIF \cup K_2.rIF) \cap (K_1.iIF \cup K_2.iIF) = \emptyset.$$

The *parallel composition* $K_1 \parallel K_2$ defines the component K such that

$$\begin{aligned}
K.F &= K_1.F \cup K_2.F,\\
K.iA &= K_1.iA \cup K_2.iA,\\
K.iIF &= K_1.iIF \cup K_2.iIF \cup\\
&\quad (K_1.pIF \cup K_2.pIF) \cap (K_1.rIF \cup K_2.rIF),\\
K.pIF &= (K_1.pIF \cup K_2.pIF) - (K_1.rIF \cup K_2.rIF),\\
K.rIF &= (K_1.rIF \cup K_2.rIF) - (K_1.pIF \cup K_2.pIF).
\end{aligned}$$

Notice that any provided methods that are "plugged into" required methods are not provided to the environment anymore. When $K_1.rIF \subseteq K_2.pIF$ and $K_2.rIF \cap K_1.pIF = \emptyset$, we use $K_1 \ll K_2$, called K_2 *plugged into* K_1, to denote $K_1 \parallel K_2$.

When K_1 and K_2 have all their interfaces disjoint from each other, we use $K_1 \oplus K_2$ to denote $K_1 \parallel K_2$, called *disjoint union*.

Then the semantics of an open component K is defined by a functional $[\![K]\!] : \mathscr{C} \longrightarrow \mathscr{C}$, where $\mathscr{C}$ is the set of closed components. So given a closed component K_1, $[\![K]\!](K_1)$ is defined if $K_1.pIF = K.rIF$ and it is the closed component $K \ll K_1$. For two open components K_h and K_l, $K_h \sqsubseteq K_l$ if for all K such that $K_h \ll K \sqsubseteq K_l \ll K$. Notice that if $K_h \sqsubseteq K_l$, then $K \ll K_h \sqsubseteq K \ll K_l$. A component composition in the graphical and textual representation can be seen in Fig. 2. The semantics of an open component will be shown in Example 4.

Definition 13 (Composability) Two open components K_1 and K_2 are *composable* if there exists a closed component K such that $K.pIF = ((K_1 \parallel K_2).rIF$ and the provided protocol of $(K_1 \parallel K_2) \ll K$ contains non-empty trace.

Composability is essential for *decompositional design* and component synthesis.

Definition 14 (Decompositional design) Let C and K be a contract and an open component, respectively. C is *implementable* by K if there is a component K_1 such that $C \sqsubseteq (K \ll K_1)$.

The definition can also be defined for a publication, and in that case K_1 provides all the methods required by K and the provided protocol of K_1 must be a superset of the *required protocol* of K, denoted by $K.rProt(B.pProt)$, with respect to the protocol specified in B, denoted by $B.pProt$. That is, $K.rProt(B.pProt)$ is the set of the sequences of invocations to K_1 made by K to deliver the protocol $B.pProt$.

Notice that interface adaption operations can be implemented by simple open components, that are classified as *connectors*:

1) Renaming the interface methods $m_i()$ in $K.pIF$ to $m_i^r()$ can be realized by $K^r \ll K$, denoted by $K[m_1^r/m_1 \ldots m_n^r/m_n]$ where for each $m_i()$ in $K.pIF$, $m_i^r()\{m_i()\}$ is provided in $K^r.pIF$. Thus $K^r.rIF = K.pIF$.
2) Using "dummy" required methods, access to some provided methods of K can be restricted (or hidden) from the client. To do this, first rename K to K^r by renaming each provided m of K to m^r. Then we design K^h such that $K^h.pIF = K.pIF - M$, and for each $m() \in K^h.pIF$ $m()\{m^r()\}$, $K^h.rIF = K^r.pIF$, and $K^h.iIF = K.iIF \cup M$. Then $K \backslash M = K^h \ll K^r$ is the component with methods in M hidden from the client.
3) Using connectors, the provided protocol of K can be further constrained. We first rename K to K^r, and then design an open component K^c such that $K^c.pIF = K.pIF$, $K^c.rIF = K^r.pIF$, $m()\{m^r()\}$, and use local variables and guards of methods to ensure that $protocol(K^c)$ is the wanted subset of $protocol(K)$.

4.4 Processes

A component defined so far is rather passive in terms of its external behavior, waiting for the client to invoke its provided services. Now we define another class of components that are used to actively coordinate and pass data around service components. We call these components *processes*.

Definition 15 (Processes) A *process* is a quadruple $P = (F, init, rIF, A)$, where

1) F is a set of typed variables, called the fields of P and denoted by $P.F$,
2) $init$ is the initial condition of the variables, denoted by $P.init$,
3) rIF is a set of *guarded method invocations* of the form $g \;\&\; m(e; y)$, called the *required interface* of P and denoted by $P.rIF$, and
4) A is a set of actions, denoted by $P.iA$, written in the programming language defined by Syntax (4) (no method invocations allowed).

We also use $P.rIF$ to denote the set of method signatures in the guarded method invocations. The semantics of a process is also defined in the same as for a closed component in Definition 7. The observable events are the events in $P.rIF$. We also define the set of traces of P in the same $traces(P) = \{tr \mid (tr, <>) \in failure(P)\}$. The interaction protocol of P is defined as $traces(P)$. Refinement between processes is also defined in the same way as refinement between closed components.

Processes do not interact with each other and the parallel composition $P_1 \;|||\; P_2$ of P_1 and P_2 is the interleaving of the behaviors of P_1 and P_2. Formally, let $P = P_1 \;|||\; P_2$, where $P_1.F \cap P_2.F = \emptyset$, $P_1.rIF \cap P_2.rIF = \emptyset$ and $P_1.iA \cap P_2.iA = \emptyset$. Then $P.F = P_1.F \cup P_2.F$, $P.rIF = P_1.rIF \cup P_2.rIF$ and $P.iA = P_1.iA \cup P_2.iA$.

Definition 16 (Process coordination of components) Let K_1 be a closed component and P a process such that $K_1.F \cap P.F = \emptyset$, $K_1.iA \cap P.iA = \emptyset$ and $P.rIF \subset K_1.pIF$. The *coordination* of K_1 by P is the closed component $K = K_1 \odot P$ such that

1) $K.F = K_1.F \cup P.F$ and $K.init = K_1.init \wedge P.init$,
2) $K.pIF = K_1.pIF - P.rIF$,
3) $K.iA = \{guard_{K_1}(m) \wedge guard_P(m) \;\&\; body(m) \mid m \in P.rIF\} \cup K_1.iA$,
   where $guard_{K_1}(m)$ and $guard_P(m)$ are the guards of m in K_1 and P, respectively, and $body(m)$ is the body of m defined in K_1.

Similar to the compositions of components, we define that a process P and a closed component K are *composable* if $protocol(K \odot P)$ contains non-empty traces.

Example 4 We now show how an open component and a process can be used to glue two components using an example of buffers.

```
component Buffer { // one place buffer
  seq(int) b = empty;
  provided interface {
    put(int x) { b = empty & b:=<x> }
    get(; int y) { b≠empty & y:=head(b); b:=<> }
  }
}

component Connector {
  int z;
  provided interface { shift() { get1(; z); put(z) } }
  required interface { get1(; int z); put(int z) }
}
process P {
  required interface { shift() }
}
```

Let $Buff_1 = Buffer[get_1/get]$, $Buff_2 = Buffer[put_2/put]$, $BB = Connector \ll (buff_1 \oplus buff_2)$. Then $B2B = BB \odot P$ is a two-place buffer. However if we define a process as follows:

```
process P1 {
```

int x, *bool* c = **true**
required interface { c & get1(; x); ¬c & put(x) }
internal interface { c & c:=**false**; ¬c & c:=**true** }
}

$BB \odot P1$ is a three-place buffer, as $P1$ is also a place holder.

It is easy to see 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 of the modeling methods 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 term of practicability, most connectors and coordinating processes in applications are dataless, thus having a purely event-based interface behavior. This allows rCOS to enable the separation of design concerns of data functionality from interaction protocols.

5 Integrating theories, techniques, and tools into the development process

The rCOS theory of semantics and refinement is to underpin the techniques for model construction, transformation, analysis, and verification. The formal theory is not to be directly accessed by practical software engineers. The formal techniques, such as those for writing formal specifications, carrying verifications, and changing models following refinement laws, are usually hard for practitioners to directly apply in software projects. Therefore, it is essential for the techniques to have automated tool support so that they can be used correctly and effectively. The effort in the rCOS tool development aims at supporting the formal techniques in building, analyzing, and verification of models in a model-driven development process.

5.1 UML profile of rCOS

The first effort in the rCOS tool development is to ease the difficulty of users in creating formal models. To this end, we have defined and implemented a UML profile for rCOS [40,41]. The profile defines:

1) An rCOS class declaration section *ClassDecls* as a UML class diagram, e.g., Fig. 6, in which attributes and method signatures of classes are defined,
2) The object interactions in methods of classes as object sequence diagrams,
3) Component specifications and compositions, e.g., Fig. 3 and Example 1, as a UML component diagram, e.g., Fig. 2 and Fig. 4, in which fields and interface methods are defined,
4) The interaction among components and the interaction of a component with the environment as *component sequence diagrams*, in which interface protocols are defined,
5) The reactive behavior of components, as UML state diagrams, that are used for analysis and verification.

The functionality of a class method or a component provided method that changes local data of the class or the component is specified by a pair of pre- and post-conditions or program statements that do not involve invocation to methods of another class or component.

The above graphical models can be constructed using the rCOS plug-ins to Eclipse, and be automatically translated and integrated as an rCOS model of an OO architecture or a component-based architecture. The tool supports the construction of rCOS models with the formal rCOS notation, and the rCOS notation defines the precise semantics of the graphical models. Later in this section, we will see that more complicated diagrams, such as component sequence diagrams, component diagrams, and interface protocol specifications are automatically generated without the need to be constructed by the designer. The tool supports the consistency checking among the class diagrams, sequence diagrams, state diagrams, and the textual specification of local functionalities. This involves the translation of component sequence diagrams and component interface state diagrams to CSP process expressions [27,28] and invoking the FDR2 model checker for CSP.

5.2 Top-down development

We discuss the use of the rCOS method and tool support by tailoring the traditional Waterfall model of software development process and discuss what and how rCOS models are generated and analyzed at each stage of the process. For a detailed case study we refer to our earlier publication in [14,42].

• ***Initial Requirements Model*** The rCOS methodology considers the development of component-based reactive software systems from use cases. The requirements elucidation and analysis are based on understanding the application domain through the identification, discussion, and description of *use cases*. The target artifact of the requirements

elucidation and analysis is a *model of the requirements*, that is, the model of the software architecture at the level of requirements. We only focus on functional requirements and leave the non-functional requirements (or extra-functional requirements) out of this paper.

Use cases represent specific workflows within the software application. For example, in a classical library reservation system [43], borrowing or returning a book, including the necessary data entry, are modeled as separate use cases. In the CoCoME case study for a supermarket cash desk system, where we previously applied our methodology by hand [42], buying items, paying for the purchase, and reporting are separate use cases. The specification of a use case describes the flow of interactions with the environment that consists of a set of *actors*, with the system in the execution of a *business process*. In other words, it is about the flow of actions of the actors using the system in carrying out a business process. We represent each interaction of an actor with the system as an invocation (a request) to a method (a service) provided by the system. Such an invocation may often pass data into the system and receive return values from the system. The execution of a method by the system will change the state of the system, i.e., the values of the variables or the state of the objects stored in the system. Therefore, a use case identifies a set of methods that the system provides to the actors and these methods are related by the business process.

Thus, a use case is modeled as an rCOS component that has a provided interface of the methods required by the business process, the interaction protocol defines the flow of the interactions (i.e., the acceptable orders of the interactions), and the data and objects for the definition of the parameters of the method invocations and the state of the component. The model of such a component consists of models of the structure views: including a class diagram defining the classes and data types of variables and parameters of the component and a component diagram (that contains one component only) defining the provided interface of the component. It also contains models of dynamic views that are the interaction protocol as a component sequence diagram describing the flow of the interactions between the actors and the component (as part of the system), and a state diagram model of the reactive behavior. The other view is the data functionality of the execution of the interface methods, that are specified as UTP designs (i.e., pre- and post-conditions). Such a use case is rather primitive and they do not use or contain other use cases.

However, more complicated use cases are related and a use case can use, contain, or extend some other use cases, as described by a UML diagram. Requirement analysis includes the identification of these relations and decomposition of a use case as compositions of use cases. These compositions can be modeled by the rCOS component-based architecture operators. Thus, at the end of the requirement analysis, a model of component-based architecture is constructed, including the following models defined in the UML profile (see Fig. 9):

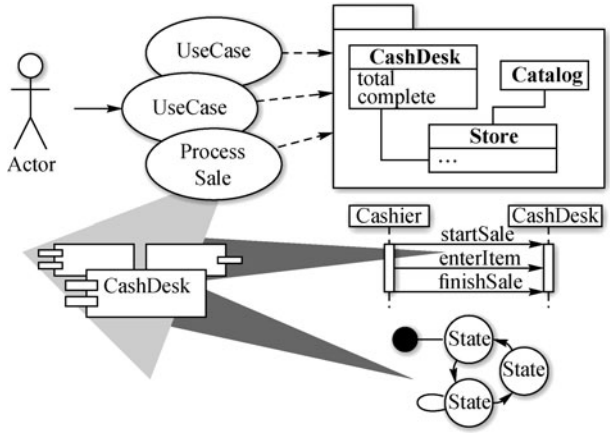


Fig. 9 Overview of UML artifacts

1) A component diagram in which each component represents a use case,
2) A class diagram or a number of class diagrams (packages) defining the classes and types of data and objects of the components,
3) A set of component-sequence diagrams represents the interactions of the components and their interactions with the actors,
4) A set of state diagrams, one for each component,
5) Local data functionality of provided methods of the components.

These models form a consistent whole model of the requirements represented in rCOS in the format shown in Fig. 3. The consistency should be checked by the rCOS tool, and the correctness should be verified to avoid deadlock during interactions among components by FDR model checking on the CSP processes generated from the sequence diagrams, and on the safety properties of state diagrams.

• ***Design through refinement and model transformations*** The methods of components in the requirements model are in general not executable, as they are specified as pre- and post-conditions in relational predicates. The design is to produce from these specifications an executable code in a modern object-oriented programming language like Java or C++. Our interest is not in generating such an executable code, but in producing specifications of the methods of the

components in the syntax defined in Section 3, called the *detailed designed model*. Obviously the detailed design model is very close to an executable implementation, but it has to be produced from the model of requirements, using the rCOS refinement and model transformations. The rCOS methodology proposes the following two major design steps.

1) *OO design of provided methods.* Consider each component K in the requirements model and each of its provided method $m()\{Spec\}$. $Spec$ is a UTP design specification $p \vdash R$ if K is a primitive component, otherwise it is a specification formed from UTP design specifications and invocations to methods of other components using command constructors. The OO design of method $m()$ refines each UTP design specification by decomposition and assignments of responsibilities of the specification to objects, called *expert objects*. This is done by following the design patterns for responsibility assignments [14,34], the Expert Pattern in particular. These patterns are formalized as refinement laws in rCOS [14,15]. The responsibilities assigned to objects are defined as methods of the classes of the objects, and the UTP design specification is then refined as method invocations to these methods of the expert objects.

Using the rCOS tool to work on the refinement of each interface method $m()$ of the component with the sequence diagram of the component actually transforms the component sequence diagram to an *object sequence diagram* in which the UTP design specifications are refinement into method invocations among the expert objects. The tool also automatically refines the class diagram (called a *conceptual class diagram*) of the requirements model by adding the methods of the expert objects to their classes, and thus the conceptual class diagram is transformed to a *design class diagram*. The component diagrams and the state diagrams of the components in the requirements model are not changed by the OO refinement.

2) *Generating a component-based design architecture model.* This step considers the object sequence diagrams of each component constructed in the above step in turns. The designer uses the rCOS tool [41] to select a number of objects lifelines in the object sequence diagram to change them into "components". Not an arbitrary selection of objects can be changed to components and the validity conditions are defined and can be automatically checked. If the validity checking passes, the tool automatically transforms the object sequence diagrams into a component sequence diagram, with the "unselected objects automatically falling into the appropriate selected component candidates as their fields".

These transformations of object sequence diagrams of components of the requirements model decompose the components into compositions of sub-components. A component diagram is automatically generated, which refines the component diagram of the requirements model. The protocols of the new components and their state diagrams of the new components are also automatically generated. This way, a model of component-based design architecture is generated, consisting of the design component diagram, the refined component sequence diagrams, the state diagrams of the components, and the design class diagram(s).

• ***Code generation*** After finally all non-executable constructs in the model have been refined (the tool indicates whether there is still work to be done), the model can then be used for code generation. We have established the principal mapping from rCOS to the Java programming language, and with the help of a few annotations (mostly on the libraries supplied by the tool), we can generate the code automatically. The code generator can easily be adapted to generate code for other object-oriented languages. Currently, the tool can only generate monolithic, non-distributed programs, and does not consider component deployment.

• ***Validation and verification*** Apart from the static checking of the model on the UML level (completeness and consistency, mostly given in the Object Constraint Language OCL [44,45]), and type checking of designs, rCOS provides further validation and verification techniques to ensure the correctness of the models generated in each development phase.

First, the requirements model can be validated using the automated prototyping tool, AutoPA [43]. This tool automatically generates executable code of the use case components directly from the requirements. For verification of deadlock freedom of component interactions, the component sequence diagrams are transformed to the input notation of the CSP as the input for FDR model checker [46] for verification of deadlock freedom. Application dependent safety and liveness properties are verified by model checking the state diagrams of the components. The correctness of the functionality of methods are mainly ensured by the application of refinement laws and correctness preserving model transformations, such as those from object sequence diagrams to component sequence diagrams. The designers can also define their own *ad hoc* model transformation or model refactoring. Then the correctness of these transformations can be proved with the aid of the theorem proving tool presented in [47]. For preliminary tool support to rCOS component-based testing, we refer to the work in [48].

6 Conclusions

In this paper, it is the first time that we have given a comprehensive introduction to the rCOS method for formal model-driven software development. This includes the ideas behind the development of the method. 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 model transformations. It effectively attacks the challenges of consistent integration of models of different viewpoints of the software system under development, such that different theories, techniques and tools can be applied effectively to the corresponding models. 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. It provides a seamless combination of OO design and component-based design. As the semantic foundation presented in Section 3 and the models of components in Section 4 show, rCOS allows the classical specification and verification techniques, Hoare Logic and calculus of 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 also integrated into the methods. In particular, design patterns and refactoring techniques in object-oriented design are formalized. Construction of models and model refinements are supported by the rCOS tool. The method has been tested on enterprise systems [14,42], remote medical systems [49], and service oriented systems [50].

Similar to JML [26], rCOS method intends to convey the message that the design of a formal modeling notation can and should take advantage of the advanced features of the architectural constructs in modern programming languages like Java. This will make it is easier to use and understand for practical software engineers, who have difficulties to comprehend heavy mathematical constructs and operators.

This paper is a combination of the results published in several papers. However, it is the first time that we present the full semantic foundation of rCOS. Also, the models of closed components semantics and the coordinating processes are improved, mainly to ease the understanding and link to existing theories, compared to the definitions we gave earlier, including the internal autonomous actions. This has allowed us to define the coordination of a component by a process much more smoothly. The link of the rCOS models to classical semantic models is presented in this paper. We have not spent space on related work as that has been discussed in the papers that we have referred to in this paper. 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 is applied to workflow management, health care workflows in particular [51].

Acknowledgements Many of our former and current colleagues have made contributions to the development of the rCOS method and its tool support. Together with the second and third authors of this paper, He Jifeng started the investigation of the semantic theory of rCOS and established the first layer of the foundation for the rCOS method. Jing Liu and her group have made significant contributions to the link of rCOS to UML and service oriented architecture; Xin Chen, Zhenbang Chen, E-Y Kang, 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 Ruzhen Dong 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 research on rCOS has been supported in parts over the years by the projects HighQSoftD, HTTS, ARV and GAVES funded by Macao Science and Technology Development Fund and the National Nature Science Foundation of China (Grant Nos. 60970031, 61073022).

References

1. Dijkstra E W. The humble programmer. Communications of the ACM, 1972, 15(10): 859–866, ACM Turing Award lecture
2. Brooks Jr F P. No silver bullet: Essence and accidents of software engineering. IEEE Computer, 1987, 20(4): 10–19
3. Booch G. Object-Oriented Analysis and Design with Applications. Boston: Addison-Wesley, 1994
4. Brooks Jr F P. The mythical man-month: After 20 years. IEEE Software, 1995, 12(5): 57–60
5. Holzmann G J. Conquering complexity. IEEE Computer, 2007, 40(12): 111–113
6. Wirsing M, Banâtre J P, Hölzl M, Rauschmayer A. Software-Intensive Systems and New Computing Paradigms — Challenges and Visions. Lecture Notes in Computer Science, 2008, 5380
7. Peter L. The Peter Pyramid. New York: William Morrow, 1986
8. Leveson N G, Turner C S. An investigation of the Therac-25 accidents. IEEE Computer, 1993, 26(7): 18–41
9. Robinson K. Ariane 5: Flight 501 failure — A case study. http://www.cse.unsw.edu.au/~se4921/PDF/ariane5-article.pdf, 2011
10. Johnson J. My Life Is Failure: 100 Things You Should Know to Be a Better Project Leader. West Yarmouth: Standish Group International, 2006
11. Szyperski C. Component Software: Beyond Object-Oriented Program-

ming. Boston: Addison-Wesley, 1997
12. Object Management Group. Model driven architecture — A technical perspective. Document number ORMSC 2001-07-01, 2001
13. Liu Z, Kang E, Zhan N. Composition and refinement of components. In: Butterfield A, eds. Post Event Proceedings of UTP08. Lecture Notes in Computer Science, 2009, 5713
14. Chen Z, Liu Z, Ravn A P, Stolz V, Zhan N. Refinement and verification in component-based model driven design. Science of Computer Programming, 2009, 74(4): 168–196
15. Zhao L, Liu X, Liu Z, Qiu Z. Graph transformations for object-oriented refinement. Formal Aspects of Computing, 2009, 21(1–2): 103–131
16. 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, Lecture Notes in Computer Science, 2007, 4767: 191–206
17. Hoare C A R. An axiomatic basis for computer programming. Communications of the ACM, 1969, 12(10): 576–580
18. Chen X, Liu Z, Mencl V. Separation of concerns and consistent integration in requirements modelling. In: Leeuwen J, Italiano G F, Hoek W, Meinel C, Sack H, Plášil F, eds. Proceedings of 33rd Conference on Current Trends in Theory and Practice of Computer Science. Lecture Notes in Computer Science, 2007, 4362
19. Liu J, Liu Z, He J, Li X. Linking UML models of design and requirement. In: Proceedings of the 2004 Australian Software Engineering Conference. Washington: IEEE Computer Society, 2004, 329–338
20. Li X, Liu Z, He J. Consistency checking of UML requirements. In: Proceedings of 10th International Conference on Engineering of Complex Computer Systems. Washington: IEEE Computer Society, 2005, 411–420
21. He J, Li X, Liu Z. A theory of reactive components. Electronic Notes in Theoretical Computer Science, 2006, 160: 173–195
22. He J, Liu Z, Li X. rCOS: A refinement calculus of object systems. Theoretical Computer Science, 2006, 365(1–2): 109–142
23. 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. Lecture Notes in Computer Science, 2009, 5885: 347–366
24. Spivey J M. The Z Notation: A Reference Manual. 2nd ed. Upper Saddle River: Prentice Hall, 1992
25. Jones C B. Systematic Software Development Using VDM. Upper Saddle River: Prentice Hall, 1990
26. 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. Lecture Notes in Computer Science, 2006, 4260: 2–34
27. Hoare C A R. Communicating Sequential Processes. Upper Saddle River: Prentice-Hall, 1985
28. Roscoe A W. Theory and Practice of Concurrency. Upper Saddle River: Prentice-Hall, 1997
29. Alfaro Ld, Henzinger T A. Interface automata. SIGSOFT Software Engineering Notes, 2001, 26(5): 109–120
30. Liu Z, Joseph M. Specification and verification of fault tolerance, timing, and scheduling. ACM Transactions on Programming Languages and Systems, 1999, 21(1): 46–89
31. Hoare C A R, He J. Unifying Theories of Programming. Upper Saddle River: Prentice-Hall, 1998
32. Dijkstra E W, Scholten C S. Predicate Calculus and Program Semantics. New York: Springer-Verlag, 1990
33. Fowler M. Refactoring — Improving the Design of Existing Code. Menlo Park: Addison-Wesley, 1999
34. Larman C. Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process. 3rd ed. Upper Saddle River: Prentice-Hall, 2005
35. Chandy K M, Misra J. Parallel Program Design: A Foundation. Reading: Addison-Wesley, 1988
36. Back R J R, von Wright J. Trace refinement of action systems. In: Proceedings of 5th International Conference on Concurrency Theory. Lecture Notes in Computer Science, 1994, 836: 367–384
37. Lamport L. The temporal logic of actions. ACM Transactions on Programming Languages and Systems, 1994, 16(3): 872–923
38. Milner R. Communication and Concurrency. Upper Saddle River: Prentice-Hall, 1989
39. Lynch N A, Tuttle M R. An introduction to input/output automata. CWI Quarterly, 1989, 2(3): 219–246
40. Chen Z, Liu Z, Stolz V. The rCOS tool. In: Fitzgerald J, Larsen P G, Sahara S, eds. Modelling and Analysis in VDM: Proceedings of the Fourth VDM/OvertureWorkshop, number CSTR-1099 in Technical Report Series. Newcastle: University of Newcastle Upon Tyne, 2008, 15–24
41. Li D, Li X, Liu Z, Stolz V. Interactive transformations from object-oriented models to component-based models. Technical Report 451, IIST, United Nations University, Macao, 2011
42. 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. Lecture Notes in Computer Science, 2008, 5153 (Chapter 3): 116–145
43. Li X, Liu Z, Schäf M, Yin L. AutoPA: Automatic prototyping from requirements. In: Margaria T, Steffen B, eds. Proceedings of 4th International Conference on Leveraging Applications of Formal Methods. Lecture Notes in Computer Science, 2010, 6415: 609–624
44. Object Management Group. Object constraint language, version 2.0, May 2006
45. Warmer J, Kleppe A. The Object Constraint Language: Precise Modeling with UML. Boston: Addison-Wesley, 1999
46. Chen Z, Morisset C, Stolz V. Specification and validation of behavioural protocols in the rCOS modeler. In: Arbab F, Sirjani M, eds. Proceedings of 3rd IPM International Conference on Fundamentals of Software Engineering. Lecture Notes in Computer Science, 2009, 5961: 387–401
47. Liu Z, Morisset C, Wang S. A graph-based implementation for mechanized refinement calculus of oo programs. In: Davies J, Silva L, Silva Simão Ad, eds. Proceedings of 13th Brazilian Symposium on Formal Methods. Lecture Notes in Computer Science, 2010, 6527: 258–273
48. Lei B, Li X, Liu Z, Morisset C, Stolz V. Robustness testing for software components. Science of Computer Programming, 2010, 75(10): 879–897
49. 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. Elec-

tronic Notes in Theoretical Computer Science, 2010, 266: 77–92

50. Liu J, He J. Reactive component based service-oriented design–a case study. In: Proceedings of 11th IEEE International Conference on Engineering of Complex Computer Systems. Washington: IEEE Computer Society, 2006, 27–36

51. Bertolini C, Liu Z, Schäf M, Stolz V. Towards a formal integrated model of collaborative healthcare workflows. Technical Report 450, IIST, United Nations University, Macao, 2011. In: Proceedings of 1st International Symposium on Foundations of Health Information Engineering and Systems (In press)

Wei Ke is a researcher and lecturer of Macao Polytechnic Institute. He received his MSc from Institute of Software of the Chinese Academy of Sciences. He is currently a PhD student of School of Computer Science and Engineering, Beihang University. His research interests include programming languages, formal methods and tool support for object-oriented and component-based engineering and systems. His recent research focus is model-driven architectures in health informatics.

Xiaoshan Li is an associate professor of Department of Computer and Information Science, University of Macau. He received his PhD in 1994 from Institute of Software of the Chinese Academy of Sciences. His research interests include formal specification and verification of concurrent and real-time systems, and sound methods for object-oriented and component-based engineering and systems. His recent research focus is software engineering methods in health care.

Zhiming Liu is a Senior Research Fellow of UNU-IIST and the head of Information Engineering and Technology in Health Programme (IETH). Before UNU-IIST, he was a University Lecturer at the University of Leicester and a Research Fellow at the University of Warwick. He holds a master degree from the Institute of Software of the Chinese Academy of Sciences, and a PhD from the University of Warwick. His research interest is in formal theories and techniques of software engineering. He is internationally known for his work on the Transformational Approach to Fault-Tolerance and Real-Time computing, and the rCOS Method of Model-Driven Design of Component Software. The research of IETH extends and applies these methods to human and environmental health care.

Volker Stolz is a post-doc in the Precise Modelling and Analysis group in the Department of Informatics at the University of Oslo, Norway, and Adjunct Research Fellow at UNU-IIST, where he is Principal Investigator of the “Applied Runtime Verification” project. He holds a master and PhD degree in Computer Science from RWTH Aachen, Germany. His current interest is integration of formal methods into main-stream software engineering approaches and tools.