Linking Formal Methods in Software Development
A Reflection on the Development of rCOS

Zhiming Liu(✉)

School of Computer and Information Science, Southwest University, Chongqing, China
zhimingliu88@swu.edu.cn

Abstract. The method of *refinement of object-oriented and component-based systems* (rCOS) has been developed based on the Unifying Theories of Programming (UTP) of Tony Hoare and Jifeng He. It is influenced by the doctrine of institutions, espoused by Joseph Goguen and Rod Burstall, for linking specification languages and verification techniques to support model-driven development of software systems. The research on rCOS has produced a body of knowledge and techniques, including the formal use of the Unified Modelling Language (UML), a theory of semantics and refinement of object-oriented programs, a theory of semantics and refinement of the component-based architecture of software systems, and prototypes of model-driven tools. These have been published in a number of papers and embodied in several lecture notes taught at many classes and training schools. In this Festschrift paper, I reflect on the research in the development of rCOS by giving a summary of the results with discussions on the fundamental ideas, the way it has been developed, its current status, and where it may take us in the future.

Keywords: UTP · institutions · rCOS · architecture modelling · Human-Cyber-Physical Systems

1 Introduction

I worked with Professor Jifeng He from 2001 to 2005 at the United Nations University International Institute for Software Technology (UNU-IIST, Macao). Our close collaboration started before that in 1998 and continued afterwards until 2007. Through those years and thereafter, Professor He has been a friend and mentor of mine. My research has developed under his inspiration and influence. The main outcome of the collaboration is the rCOS method for model-driven development of object and component systems.

Supported by the Chinese National NSF grant (No. 62032019) and the Southwest University Research Development grant (No. SWU116007).

J. P. Bowen et al. (Eds.): *Theories of Programming and Formal Methods*, LNCS 14080, pp. 52–84, 2023.
https://doi.org/10.1007/978-3-031-40436-8_3

In this Festschrift paper, I give a reflection on the development of the rCOS method to pay tributes to Jifeng's original contributions. I do this by providing a summary of the philosophical ideas, theory, methods and techniques embodied in rCOS. I also discuss how these can be related to practical software development. To this end, I first give a brief introduction to the areas where the contributions in rCOS are relevant.

Software Engineering: Software engineering has been developed from the studies and practices of writing *closed sequential programs*, through producing *software products*, to designing and implementing *software systems* (as products too), as described in the talk of Fred Brooks at ICSE 2018 [5]. In this process of advances, together with the advances in hardware and network technologies, the complexity of software development has been increasing in multiple dimensions, along with the increase in the size of applications and the complexity of requirements. As Brooks described in his talk, programming was mainly an activity where someone wrote programs used by themselves, and thus only required ad hoc testing and debugging; producing programs as products requires more work in documentation, more thorough and systematic testing, and quality assurance; designing and producing software systems requires even more work on the understanding of *interfaces*, analysis, verification, debugging and maintenance.

Theories, techniques, and tools have been developed and are still being developed for software development to support the mastering of the increasing complexity and requirements, and they are core constituents of software engineering as a system engineering discipline. The ever-growing complexities of systems and applications, as well as the theories, techniques, and tools, are reflected in the development of the software industry, as Brian Randell, who is another software engineering pioneer, described in his talk at ICSE 2018 [62], though gaps still exist.

Modelling: Modelling is essentially important and effective in all scientific and engineering disciplines. Let us take the view from Lee [33] that *"modelling is the trinity of the model, the thing being modelled and the modelling paradigm"*. Here the *modelling paradigm* refers to the underlying theory, techniques and tools for *analysis, simulation* and *verification* of properties of and relations between models. The "thing", either logical or physical, is either an engineering product or a system which is either pre-existing or to be constructed. In the former case, the models of the things are built for scientific study and analysis of the things that have expected or conjectured properties and relations through verification, simulation, and experiments. In the latter case, the models, properties, and relations serve as the *requirements specification* for the engineering product or system, and the correctness of the product and models are verified through logical reasoning, model-checking testing, and simulation.

It is important to note that a "thing" to be modelled usually has a well-defined structure, such that the thing is a composition of some constituent "things", and the structure is often hierarchical. Therefore, models are also *hierarchical* and, therefore, the modelling paradigm needs to support operations for *model compositions*. This is especially true in modelling engineering systems. A modelling paradigm which supports hierarchical model compositions is known

as *compositional modelling* and *components-based modelling*. However, the latter notional emphasis on compositions of *heterogeneous models* involving heterogeneous things requires the linking of different modelling (sub-)paradigms. The notion of *interface contracts* [22,40,65] plays an important role in heterogeneous component-based modelling.

A model is not the "thing" that the model represents. It is an abstraction of the thing instead, from a particular viewpoint of the modeller based on their interest in the problem that they are solving. Therefore, in general, a whole model of a thing is an integration of models representing the different viewpoints of different modellers and/or at different times. A paradigm supporting integration of models of different viewpoints is known as a *multi-view modelling paradigm*.

In an engineering process, models of different levels of abstraction are usually needed for, say, analysis and validation of requirements, verification of designs in different phases, and verification and testing of implementations. The modelling paradigm must support establishing the correct relation between models of different levels of abstraction. The relation is generally defined by the notion of *refinement*, meaning that all properties of a higher-level model are preserved by a lower-level model. Rules, techniques, and algorithms are required in a modelling paradigm to support manipulations of models for building models at one level from models at other levels. These manipulations are called *model transformations* in *model-driven development* [35,41,59]. Paradigms with features including multi-view, compositional, and component-based modelling, and model refinement and abstraction, are important in the provision of systematic support for the engineering principles of *separation of concerns*, *divide and conquer*, and *reuse*. The rCOS method has been developed to reflect these features.

Formal Methods: The term *formal methods* refers to, in computer science and engineering, mathematically rigorous techniques and tools for the specification, design, and verification of software and hardware systems [6,73]. A formal method is essentially a modelling paradigm as defined in the paragraph immediately above. However, a formal method directly employs *formal logic systems* consisting of a *formal language*, a *proof system* and *a formal semantics* of the languages, together with the *meta-theory* of the formal logic system [72]. The meta-theory includes *expressiveness, soundness, completeness* and *decidability* of the logic system. The reasons why a formal method particularly needs a formal language include that programming languages, which the method treats, are formal, and syntactic guided inductive techniques are effective and computer-aided tools are essential which only take formal languages as inputs. Modelling paradigms in traditional scientific and engineering disciplines directly use mathematical language. This avoids the need for a separate semantic theory. A formal language in general is defined by a (usually finite) set of symbols, called the *alphabet* or the *signature*, and a set of rules, called *syntactic rules*, for forming grammatically *correct sentences*, also called *well-formed formulas* or *statements*.

Another important feature of formal methods is their applications in the specification, design, and verification of software systems, hardware systems, and systems with both software and hardware. In the life cycle of such a system, a number

of formal methods are needed. There are a large number of formal methods [72] addressing different design concerns that model various viewpoints at different levels of abstraction. The use of multiple formal methods gives rise to the challenge of ensuring that they are used consistently. In fact, there is even not yet a commonly agreed notion of such consistency. This challenge is tackled using two approaches. One is through *linking* different formal methods in an *institution* and formal methods from different institutions [18]. This approach is based on *category theory*. The pioneering work on this approach is the *theory of institutions* [18]. The other approach is *unifying* different semantic theories of the languages used in formal methods. The most influential and well-studied work is *Unifying Theories of Programming* (UTP) [27] by Tony Hoare and Jifeng He.

The *Unified Modeling Language* (UML) proposed a framework for defining a collection of modelling languages that can be used in project development. However, there is little work on the relations of semantic models of the UML notations. The rCOS method is based on UTP, influenced by the theory of institutions, and it started with addressing the problem of formal and consistent use of UML notations.

This paper presents a reflection on the rCOS method, with discussions about its origins, where it contributes, and its possible further development. Through the discussion, where and how different formal methods are linked are shown based on the problems they address and their uses in various cycles of systems development. The organisation of the paper is as follows. Section 2 gives a unified overview of formal methods; Sect. 3 reflects on the ideas and development of the rCOS methods, including the formal use of UML, the semantics and refinement of objective-oriented (OO) systems that are effective, and the modelling of component-based architecture; Sect. 4 outlines some ideas on extending rCOS to modelling human-cyber-physical systems (HCPS); and Sect. 5 gives some concluding remarks and acknowledgements.

2 A Unified Overview of Formal Methods

Modern theories of computation and programming are developed from the computational models of *λ-calculus*, *recursive functions*, and *Turing machines*. Each of them can be seen as an extension to the formal logic system of arithmetic. These have provided the foundation for the design and implementation of programming languages, the design of programs, and the analysis of programs through logical reasoning about and decision algorithms for checking properties (i.e., model checking) of their executions. All programming languages so far are defined with formal syntax, as in formal logic systems, but they are usually implemented based on informal semantics. For rigorous reasoning and verification of programs, their *formal semantics* must be defined.

2.1 Semantic Theories

There are mainly three kinds of semantics theories, *operational semantics*, *denotational semantics*, and *axiomatic semantics*.

Operational Semantics: An operational formal semantics of a programming language is defined as a deduction system that the *execution configurations* must follow, which is similar to the deductions in *λ-calculus*. Such an operational semantics is also called a *term-rewriting semantics* [80]. The deduction system of operational semantics can also be defined by abstract state machines, similar to Turing machines, called *abstract machines* [56]. More abstract and more structured, thus generally used operational semantics are defined as a model of abstract machines, called *labelled transition systems* [61].

Denotational Semantics: The operational semantics of a programming language is regarded to be too close to particular implementations of the language. In other words, operational semantics expresses too many details of the execution of a program, and thus it is difficult to be used for abstract requirements specification and verification. Semantic theorists believe that the semantics of a programming language should be independent of its implementation to allow more implementations.

The denotational semantics employs the approach to defining an *interpretation* of the formal language of a logic system in a mathematical structure, called the *domain* of the interpretation [51]. In this approach, the meanings of the syntactic elements of a language are defined by mapping them to objects, called the *denotations* of the syntactic elements, in the domain.

For example, the alphabet of a formal language of a logic system or a programming language usually includes symbols representing variables, constants, functions and relations. The atomic sentences or statements, operators on sentences, etc., are defined using these symbols by syntactic rules. The denotations of constant symbols are given as elements of the denotational domain; the denotations of variables are given as *assignment functions* from the set of variables to the elements (called *values*) in the domain; the denotation of a sentence is given as a relation among elements in the domain, and the denotations of the operators on sentences are defined as operations on relations, inductively following the syntactic rules. The basics of the semantics of logic systems can be found in any book on mathematical logic, and the fundamental theory for denotational semantics of programming languages is the *Scott-Strachey domain theory* [17,67]. In our recent book [51], a unified view is presented on mathematical logic and the logic of programs.

Axiomatic Semantics: Axiomatic semantics defines the meaning of programs as formulas of a formal logic system, which is an extension to an existing formal logic system, such as first-order arithmetic. The semantics of an atomic sentence is defined as an axiom (scheme), and the semantics of a language operator is defined by an inference rule so that the semantics of a composite sentence can be inferred from the semantics of the atomic sentences. The properties of programs can then be reasoned about in the logic system. The pioneering work on axiomatic semantics is *Floyd-Hoare logic* [15,25], while Hoare Logic is the most well-studied and used axiomatic semantics.

Relations Between the Different Semantic Theories: Applying the approach of the semantic theory of formal logic systems to the development of denotational semantics of programming languages directly puts specification and analysis of program properties into the framework of formal logic systems, theorem proving and verification (including testing and model checking).

The correctness of the denotational semantics is usually justified based on operational semantics through an abstraction mapping from the operational semantics of sentences to their denotation, e.g., functions and relations on program states. A formal logic system which defines the axiomatic semantics of a programming language can be interpreted in its denotational semantics and operational semantics. This, together with the justification of denotational semantics based on operational semantics indicates the relation between the three approaches of semantics.

It is not difficult to understand that the semantics $[\![P]\!]$ of a program P is the set consisting the predicate pairs (p, q) such that $\{p\}P\{q\}$ holds in Hoare logic, i.e.:

$$[\![P]\!] = \{[p, q] \mid \{p\}P\{q\} \text{ is a theorem of Hoare logic}\}$$

This is then a denotational semantic view of axiomatic semantics. With this view, a pair $[p_0, q_0]$ of predicate formulas can be used as a requirements specification for the development of a program P such that $\{p_0\}P\{q_0\}$ is a theorem of Hoare logic. According to the *consequence rule* of Hoare logic, any program in the following set is correct with respect to the specification $[p_0, q_0]$ (or $\{p_0\}P\{q_0\}$ in Hoare logic):

$$\mathcal{P}([p_0, q_0]) = \{P \mid \{p\}P\{q\} \text{ and } (p_0 \to p) \wedge (q \to q_0) \text{ are theorems of Hoare logic}\}$$

This is the foundation of *programming from specifications* presented in [58].

Hoare and He's UTP presents a comprehensive theory which allows a program of different paradigms to be defined uniformly by a pair of first-order formulas, called the *pre-* and *post-conditions*, as its semantics. Here, the programs can be of "any" kind, concurrent and real-time programs as well as sequential programs. Further on, it is shown in the work of rCOS that a theory of semantics and refinement of object-oriented programs [10,24,78] and a theory of contract-based semantics and refinement [7,10,22,23] of component programming [70] are also established in this way.

2.2 Linking Formal Methods for Their Consistent Use

Formal methods in the early years were used to adopt formal logic (including Hoare logic) as the specification languages, interpreted in semantic models of programs, for formulating properties of programs. Reasoning and algorithm-based verification are carried out in the corresponding logic systems. Along with the increasing complexity of systems and dimensions of requirements, large numbers of abstract specification and modelling languages have been developed, and so are theories of their semantics. These together with the techniques and tools for

reasoning and verification constitute a large number of formal methods. Included in the *Formal methods* Wikipedia page (as of 31 March 2023), there are about 30 specification languages, together with more than a dozen model-checking tools, which all need specification languages. These do not include the many modelling languages, e.g., model-driven and component-based modelling languages, languages for modelling embedded systems, etc.

Different specification (or modelling) languages are proposed for describing abstractions of different concerns and from different viewpoints of software developers. We can roughly classify the different languages based on the aspects of concerns and viewpoints.

Event-Based Methods: The representatives of this kind include automata-based models (e.g., I/O automata) [55], CSP [26,63] and CCS [57], and those alike [4]. Operational semantics are defined for these languages, and different theories of denotational semantics also exist, such as the models of *traces*, *failures*, or *divergences* [63]. Based on these theories, techniques of algebraic reasoning through the relations of *simulation* [57] and *refinement* [63] are developed. Models described with these languages abstract the internal computation away and describe the behaviour of interactions and concurrency among different components.

Data State-Based Methods: One class of state-based formal methods is associated with operational semantics. Well-known methods of this kind include *action systems* [3], the *B-Method* [1,66], *Alloy* [28], and *TLA+* [30,31]. Another class of state-based specification languages have denotational semantics and axiomatic semantics in Hoare logic. Examples of these formal methods include *VDM* [29] and the *Z notation* [69].

Combination of Event-Based and State-Based Methods: Throughout the cycles of software development, formal methods for sequential programming and for concurrent and communicating programs are involved, and both event-based and state-based methods are needed. Especially in an event-based model, before the occurrence of an event is the *internal execution* of an *atomic action* which is implemented by a piece of the program. The functional correctness of these atomic actions is specified and verified at a refined level of abstraction. However, there are formal approaches that unify event-based and state-based modelling and verification: value-passing CCS and CSP (in which the combination is limited), the Occam programming language (developed based on CSP) [64], and Event-B [2], for example.

General Unification of Formal Methods: In addition to the need for consistent use of multiple formal methods in system development, new language abstractions are required for systems with more functional and performance requirements, such as spacial and timing requirements and energy constraints, and concurrency between discrete digital systems and continuous physical systems as well as intelligent system (both artificial and human) in the emerging human-cyber-physical system (HCPS) [52,76]. We believe that, instead of defining a new comprehensive specification language and its semantic models from

scratch, a method is desirable for extending and linking existing languages and their semantic models.

The theory of institutions of Goguen and Burstall and the theory and UTP of Hoare and He provide the theoretical basis and insight for this purpose. The former provides a theory and a method for linking different specification languages, models of sentences written in the languages, theories and proofs consistently; and the latter is a framework for defining new semantic models from existing ones. We present a very brief introduction to these two theories just to show the basic idea with some formalities. The reason is that I have found that the philosophical ideas of these two theories are not widely understood, by young researchers in particular. The purpose is to show their differences and relations (intuitively) and to propose a research topic on the study of the relationship between the two approaches.

2.3 Institutions

The theory of institutions is based on category theory. A *category* $\mathbf{C}$ is formed of two sorts of elements $ob(\mathbf{C})$ and $hom(\mathbf{C})$ which are respectively called *objects* and *morphisms*, such that:

- each morphism $m \in hom(\mathbf{C})$ has a *source* $a \in ob(\mathbf{C})$ and a *target* $b \in ob(\mathbf{C})$, the morphism is denoted as $m : a \to b$ and m is called a morphism or an *arrow* from a to b and $hom(a, b)$ denotes the set of arrows from a to b;
- a binary operation $\cdot$ on $hom(\mathbf{C})$, called *composition*, such that $m_2 \cdot m_1$ is defined for $m_1 \in hom(a, b)$ and $m_2 \in hom(c, d)$ if and only if $b = c$, and $m_2 \cdot m_1 \in hom(a, d)$;
- the operation $\cdot$ has the following two properties:
  - it is associative that if $(m_1 \cdot m_2)$ and $(m_1 \cdot m_2) \cdot m_3$ are defined, so are $m_1 \cdot (m_2 \cdot m_3)$ and it equals to $(m_1 \cdot m_2) \cdot m_3$;
  - for each object $o \in ob(\mathbf{C})$, there is an *identity morphism* $1_o : o \to o$ such that for any morphism $m : a \to b$, $1_b \cdot m = m = m \cdot 1_a$

It is easy to see that for any category $\mathbf{C}$, there is an *opposite category* $\mathbf{C}^{op}$ such that $ob(\mathbf{C}^{op}) = ob(\mathbf{C})$, and $hom(\mathbf{C}^{op}) = \{m^r : b \to a \mid m : a \to b \in hom(\mathbf{C})\}$. Another simple and important example of category is the *category of sets*, denoted as **Set** such that the objects of **Set** are sets and the set of morphisms $hom(S_1, S_2)$ are the total functions from S_1 to S_2.

Another important concept in category theory is the notion of *functors*. A functor F from a category $\mathbf{C}_1$ to category $\mathbf{C}_2$ consists of two mappings (F_o, F_m) where $F_o : ob(\mathbf{C}_1) \to ob(\mathbf{C}_2)$ is a mapping from the objects of $\mathbf{C}_1$ to those of $\mathbf{C}_2$, and $F_m : hom(\mathbf{C}_1) \to hom(\mathbf{C}_2)$ is a mapping from the morphisms of $\mathbf{C}_1$ to those of $\mathbf{C}_2$ such that for each morphism $m : a \to b$ of $\mathbf{C}_1$, $F_h(m)$ is a morphism from $F_o(a)$ to $F_o(b)$.

A category is called a **small category** if every of its object is a set, not a "proper class". Taking all small categories as the objects and the functors between the small categories as the morphisms, it forms a category, called *the category of small categories*, and it is denoted by **Cat**.

Now recall that a specification language is defined from a set Σ of symbols, called the *signature* of the language, a number of syntactic rules for generating a *set of sentences* (or *well-formed formulas*), and each sentence has a *set of models*. Now, these are put together to form an *institution*.

Definition 1. Institution *An* institution *consists of*

- *a category* **Sign** *of* **signatures**;
- *a functor* $Sen : \mathbf{Sign} \rightarrow \mathbf{Set}$ *which gives, for each signature* Σ*, the set of* **sentences** $Sen(\Sigma)$*, and for each signature morphism* $\sigma : \Sigma \rightarrow \Sigma'$*, the* **sentence translation morphism** *(or mapping)* $Sen(\sigma) : Sen(\Sigma) \rightarrow Sen(\Sigma')$*, where often* $Sen(\sigma)(\varphi)$ *is written as* $\sigma(\varphi)$;
- *a functor* $\mathbf{Mod} : \mathbf{Sign} \rightarrow \mathbf{Cat}^{op}$*, which gives, for each signature* Σ*, the* category of models $\mathbf{Mod}_{ob}(\Sigma)$*, and for each signature morphism* $\sigma : \Sigma \rightarrow \Sigma'$*, the* **reduct functor** $\mathbf{Mod}_{ob}(\Sigma') \rightarrow \mathbf{Mod}(\Sigma)$;
- **the satisfaction relation** $\models_{\Sigma} \subseteq \mathbf{Mod}(\Sigma) \times Sen(\Sigma)$*, where* $(M, \varphi) \in \models_{\Sigma}$ *is written as* $M \models_{\Sigma} \varphi$

such that for each signature morphism $\sigma : \Sigma \rightarrow \Sigma'$ *in* **Sign**, *the following* **satisfaction condition holds** *for each* $\varphi \in Sen(\Sigma)$ *and each* $M' \in \mathbf{Mod}(\Sigma')$

$$M' \models_{\Sigma'} \sigma(\varphi) \ \ iff \ \ \mathbf{Mod}_{ob}(\sigma)(M') \models_{\Sigma} \varphi$$

The whole theory and rigorous use of the theory involve advanced knowledge and techniques in mathematics, *co-algebra* in particular. However, the intuition is rather clear. The truth that the satisfaction relation is invariant under a change of notation in an institution provides insight into how different methods can be used consistently in system development. The change in specification notation is usually about *refinement* and *abstraction* in terms of models, or in algebraic terms, *enlargement* or *quotienting* of context.

2.4 Unifying Theories of Programming (UTP)

A semantic model of programming in a paradigm is defined based on considering what information needs to be observed in the execution of a program. UTP provides a unified framework for defining different theories and semantics of programs in different paradigms and for describing different properties. A theory **T** of programs in a paradigm is about the characterisation of the behaviour of the programs by a set of *alphabetised predicates*. We also denote the set of predicates of a theory **T** by **T**. A predicate in the theory contains free variables in a designated set of variables representing the *observables*, called the *alphabet* of the predicate, and the set **T** of predicates are constrained by a set of axioms called *healthiness conditions*. The theory also defines a set of operators on the set **T** and these operators form the *signature* of the theory. The linking among different theories is based on the theory of *complete lattices* and *Galois connections*. We now introduce its basic ideas.

Relational Calculus: In the book of UTP [27], a theory **R** of relations is first introduced. In this theory, the observables are represented by a given set X of variables, together with their decorated versions[1] $X' = \{x' \mid x \in X\}$, and $\alpha = X \cup X'$ is called the *alphabet* of the theory, representing the observables. A program (or a specification) in this theory is defined by a first-order logic formula P, called a *relation*, associated with a subset αP of α such that P only mentions variables in αP. The set αP is called the *alphabet* of the relation P. Thus, a relation is written in the form $(\alpha P, P)$, and $\alpha P = in\alpha P \cup out\alpha P$.

The sets αP, $in\alpha P$ and $out\alpha P$ are, respectively, called the *alphabet*, *input alphabet* and *output alphabet* of the relation. The input alphabet is undashed variables representing initial values and the output alphabet variables stand for the final values of the relation, respectively. We only consider the case when $out\alpha P = in\alpha' P = \{x' \mid x \in in\alpha P\}$, and in such a case P is called a *homogeneous relation*.

The predicate P is interpreted on the domain $\mathcal{D} = \{(s, s') \mid s, s' : X \to V\}$, and $(s, s') \models P$ if $P(X/s, X'/s')$ holds, where X/s and X/s' denote substitutions of every variable $x \in X$ and $x' \in X'$ by the values $s(x)$ and $s'(x)$ in the states s and s', respectively. For example, $x' = x + 1$ specifies the relation such that for each (s, s') in the relation, $s'(x') = s(x) + 1$. There is a special class of predicates which do not mention variables in the output and they are called *conditions*. Predicates appearing in programs are only conditions. We use lowercase letters p, q, r, etc., to represent these *program stated predicates*, and use b for Boolean expressions in particular.

Signature: In addition to the notation of *alphabets* and values, a theory is also characterised by a collection of *operations* to form *expressions* or *terms* and a collection of *operators* to *compose* relations. These operations and operators form the *signature*[2] of the theory, denoted by Σ. We assume the operations and first-order logic operators in the signature and introduce some operators used in programming languages, just to show the essential idea.

For an assumed alphabet $\beta = in\beta \cup in\beta'$, an *assignment* $x := e$ is defined to be the relation

$$x := e =_{df} (\beta, (x' = e \wedge \bigwedge \{y' = y \mid y \in in\ \beta \text{ and } y \text{ differs from } x\})$$

where the variables $\alpha(e)$ of e are in $in\beta$. Therefore, the final value of x is the value of expression e obtained from the initial values of variables in e.

To define the sequential composition, we adopt the convention to use a single variable v to represent the vector of undashed variables. Then:

$$\begin{aligned} P; Q &=_{df} \exists v_0 . P[v_0/v'] \wedge Q[v_0/v], \quad \text{provided } out\alpha P = in\alpha' Q \\ in\alpha(P; Q) &=_{df} in\alpha P \\ out\alpha(P; Q) &=_{df} out\alpha Q \end{aligned}$$

[1] More mathematically, the decoration $'$ is a bijective mapping from X to X'.

[2] On terminologies, variables, values, operations and operators are included in the alphabet of a formal logic system; and they are included in the signature in the theory of institutions.

In a similar way, we can define more operators used in programming languages. A *conditional choice* between a relation P and Q according to a Boolean condition b is represented by $P \lhd b \rhd Q$. It behaves like P if the initial value of b is true, or like Q if the initial value of b is false:

$$\begin{aligned} P \lhd b \rhd Q &=_{df} (b \wedge P) \vee (\neg b \wedge Q), \quad \text{provided } \alpha(b) \subseteq \alpha P = \alpha Q \\ \alpha(P \lhd b \rhd Q) &=_{df} \alpha P \end{aligned}$$

Conditional choice is *deterministic*, and the *non-deterministic choice* between P and Q is denoted by $P \sqcap Q$ and defined by disjunction:

$$P \sqcap Q =_{df} P \vee Q, \quad \alpha(P \sqcap Q) =_{df} \alpha P$$

The program which has no effect is defined by the identity relation

$$\begin{aligned} \mathbf{skip} &=_{df} v' = v, \text{ where } v \text{ is the vector of the input alphabet} \\ &\qquad\qquad \text{and } v' \text{ the output alphabet} \\ \alpha(\mathbf{skip}) &=_{df} v \cup v' \end{aligned}$$

We use $\mathbf{skip}_\beta$ to denote the design $(\beta, \mathbf{skip})$. Another important program is the one which has totally uncontrollable or chaotic behaviour. We represent this program by $\bot$ and it is defined by $\bot_\beta =_{df} \mathbf{true}$, where β is the alphabet of $\bot_\beta$. Symmetrically, the *miracle* program on an alphabet β is defined by $\top_\beta =_{df} \mathbf{false}$.

With the above definition, we can prove algebraic equations between relations, called laws of program. In what follows, we list a few laws.

$$\begin{array}{ll} P \lhd b \rhd P = P, & P \lhd b \rhd Q = Q \lhd \neg b \rhd P \\ P; (Q; R) = (P; Q); R, & (P \lhd b \rhd Q) \lhd c \rhd R = P \lhd b \wedge c \rhd (Q \lhd c \rhd R) \\ P; \mathbf{skip}_{\alpha P} = P = \mathbf{skip}_{\alpha P}; P, & (P \lhd b \rhd Q); R = (P; R) \lhd b \rhd (Q; R) \end{array}$$

We have not yet seen the definition of a loop program which is written in the form $b * P$ and behaves like "while b holds repeat P", where $\alpha(b) \subseteq \alpha P$. This definition depends on the *fixed point* of the function $(P; \mathbf{X}) \lhd b \rhd \mathbf{skip}$. The existence of the *least fixed point* and *greatest fixed point* of this function is ensured by the fact that the set of relations forms a *complete lattice* with the partial order $P \sqsubseteq Q =_{df} [Q \Rightarrow P]$, $\bot$ and $\top$ bottom and top elements, where $[Q \Rightarrow P]$ means that the universal closure of the implication $Q \Rightarrow P$ is valid. When $[Q \Rightarrow P]$ holds, Q is called a *refinement* of P. The loop program is defined by the least fixed point $b * P =_{df} \mu \mathbf{X}.((P; \mathbf{X}) \lhd b \rhd \mathbf{skip})$.

A Theory of Program Design: We use $\mathbf{R}$ to represent the theory of relations discussed above. We can easily see that neither $\mathbf{true}; P = \mathbf{true}$ nor $P; \mathbf{true} = \mathbf{true}$ holds for an arbitrary relation P in $\mathbf{R}$. However, they both should hold for an arbitrary program P in all practical programming paradigms. Now we briefly introduce a theory, denoted by $\mathbf{D}$, in which the above two equations hold. To this end, we introduce to the alphabet X and X' two new observables represented by the Boolean variables ok and ok'. The variables ok and ok' are not program

variables held in the store, but they represent the observations that the program has started well and the program terminated well, respectively.

Furthermore, instead of allowing any arbitrary predicates as in **R**, the predicates in **D** are restricted to form $P \wedge ok \Rightarrow Q \wedge ok'$. We called such a predicate a *design* and write it as $P \vdash Q$, where P and Q do not contain ok and ok'. The definitions of some operators need to be modified as follows, where $D_1, D_2 \in \mathbf{D}$

$$\begin{aligned} \mathbf{skip} &=_{df} \mathbf{true} \vdash \overline{v}' = \overline{v} \\ \bot &=_{df} \mathbf{false} \vdash \mathbf{true} \\ \top &=_{df} \neg ok \\ x := e &=_{df} \mathit{Defn}(e) \vdash x' = e \wedge \mathit{unchange}(\mathit{others}) \\ D_1 \lhd b \rhd D_2 &=_{df} (\mathit{Defn}(b) \Rightarrow (b \wedge D_1) \vee (\neg b \wedge D_2)) \end{aligned}$$

where $\overline{v}$ and $\overline{v}'$ respectively denote the vectors of input and output alphabets, predicate $\mathit{Defn}()$ denotes the argument expression is defined for the initial values, and $\mathit{unchange}(\mathit{others})$ in the context means that no other input variables than x in the given alphabet are changed.

An important theorem shows that the set of designs is closed under all the operators defined in theory **D**. More operators can be defined, these include the declaration and undeclaration of local variables using existential quantification: **var** $x =_{df} \exists x$ and **end** $x =_{df} \exists x'$. These are used in the semantic theory of rCOS OO programming language.

Healthiness Conditions: In predicate logic, we can prove that the left zero law $(\bot; P) = P$ and the *left unit law* $(\mathbf{skip}; D) = D$. The *right zero law* $(D; \bot) = \bot$ and *right unit law* $(D; \mathbf{skip}) = D$ do not generally hold for arbitrary $D \in \mathbf{D}$. However, they are general properties for sequential programs. To characterise programs for which these laws hold, UTP adopts the approach to extending a logic system by adding more axioms, which are called *healthiness conditions*. The following four healthiness conditions are given to restrict the predicates further:

H1 $R = (ok \Rightarrow R)$
H2 $[R[false/ok'] \Rightarrow \mathbf{true}/ok']$
H3 $R = R; \mathbf{skip}$
H4 $(R; \mathbf{true}) = \mathbf{true}$

For the intuitive meaning of these healthiness conditions, we refer the reader to the book of UTP.

In predicate logic, it can be proven that a general relation R with or without ok and ok' in the alphabet, **H1** and **H2** hold iff R is a design. However, **H3** and **H4** have to be imposed as axioms, although the right unit law **H3** holds for any design of the form $p \vdash Q$, where p is a state property which does not have dashed variables. Most properties of sequential programs can be proven from the specifications of this special form.

It is proven that **D** constrained with the four healthiness conditions forms a complete lattice with the order $\sqsubseteq$, $\bot$ and $\top$, and the operators are continuous. This implies that $b * P =_{df} \mu \mathbf{X}.(P; \mathbf{X}) \lhd b \rhd \mathbf{skip}$ is in **D**.

Linking Theories: We now understand theory **R** and **D** can be used as theories of programming. Theory **D** is a sub-theory of $\mathbf{R}^{\{ok,ok'\}}$ which extends **R** by adding two the two observable and the four healthiness conditions, meaning that the set of formulas in the former is a subset of the formulas in the latter.

In either **R** or **D**, we can encode Hoare logic and Dijkstra's calculus predicate transformer. Given a predicate P in **R** or **D**, and two state properties p and q, we define the *Hoare triple*

$$\{p\}P\{q\} =_{df} [P \Rightarrow (p \Rightarrow q')]$$

where p' is the predicate that all variables in q are replaced by their dashed version. Then, the axioms and inference rule hold in **D** and **R**.

Given a predicate P of **R** or **D** and a state property r, we define the *weakest precondition* of P for the postcondition r as:

$$\mathbf{wp}(P, r) =_{df} \neg(P; \neg r)$$

Then, the rules in the **wp** calculus are valid in **D** and **R**.

The above definitions show that the *theories* of Hoare logic and **wp** calculus can be mapped into the theory **D** and **R** and used consistently. It is noticed that **D** is a theory of *total correctness* of imperative sequential programming in which assignments have no side effects. **R** can be used for *partial correctness* analysis, although the left and right zero laws, as well as the left and right unit laws, can be imposed as healthiness conditions.

The linking between theories is in general studied by functions between them with desirable properties. For this, it is generally assumed that the theories are complete lattices. For any given theories **S**, **T** and **U**, a *link function* L from T to S is a total function declared by $L : \mathbf{T} \to \mathbf{S}$. The *identity function* $\mathbf{1_T}$ maps every element of **T** to itself, and the *composition* $M \circ L : \mathbf{T} \to \mathbf{U}$ of linking functions $L : \mathbf{T} \to \mathbf{S}$ and $H : \mathbf{S} \to \mathbf{U}$ is defined as $(M \circ L)(X) = M(L(X))$, for all $X \in \mathbf{T}$.

If a theory **S** is a subset of a theory **T**, there is always a very simple link H from **S** to **T** such $H(X) = X$, for every $X \in \mathbf{S}$, thus $X \in \mathbf{T}$. It is more interesting to seek a link in the opposite direction, from the super-set theory to the subset theory $L : \mathbf{T} \to \mathbf{S}$ such that $\mathbf{S} = \{L(Y) \mid Y \in \mathbf{T}\}$.

More general links are between theories with different observables and signatures. Such links are defined as *Galois connections*. The characterisation of Galois connections indicates their significance for the consistent use of different theories.

Definition 2. Galois connection *Given complete lattices* **U** *and* **T**, *let* L *be a function from* **U** *to* **T** *and* R *a function from* **T** *to* **U**. *The pair* (L, R) *is a* **Galois connection** *if for all* $X \in \mathbf{U}$ *and* $Y \in \mathbf{T}$

$$Y \sqsubseteq L(X) \quad \textbf{iff} \quad R(Y) \sqsubseteq X$$

Thus, a Galois connection allows us to analyse the properties in one theory and reuse the results in another. *I can intuitively see that the theory of institutions*

and UTP are closely related, but I do not know any formal study on the relation. We will see that the rCOS method reflects the key ideas of these two theories of unification, although it is formulated formally only within UTP. *However, rCOS supports consistent but still separated uses of different formal methods, i.e., their specification languages, semantic theories, techniques and tools, specially developed for different design concerns in system development.* Writing all specifications and analysing their properties in the uniform notation of designs $P \vdash Q$ through the whole development would be unrealistic.

3 A Reflection on rCOS

The work on rCOS has been developed for the needs of teaching formal methods to undergraduate and graduate students, as well as training UNU-IIST fellows. It started in 1988 when object-oriented design and the Unified Modeling Language (UML) were becoming popular. The research has been evolving along with the advances in techniques for component-based and service-oriented programming, the model-driven development methodologies in particular. This section presents a summary of the development of the research with a discussion about the principle ideas and pointers to the main publications. The discussion will focus on problems, ideas for solutions and the ways to develop the solutions. We refer the reader to the papers cited in the discussions for technical details and examples, both illustrative examples and running examples, and those examples in the paper [10] and the lecture notes [39] in particular.

3.1 Formal Use of UML

UML became known to the software engineering community in 1998 or so. It was in that year when the head of my department at the University of Leicester (UK) asked to take over the teaching of the course on software development. The focus of the course was object-oriented (OO) developments. It was quite a challenge to me because I, as a young researcher in formal theories, had little knowledge of software engineering, and even less of OO design. The keyword "Unified Modelling Language" caught my attention I decided to learn and use it as the modelling language in the course.

Although there was a lot of hype about UML, many people in the formal method community were quite critical of it at the beginning; some people even called it "Undefined Modelling Language" in private. The main criticisms were that the syntax of UML models was not well defined (possibly due to that people were not used to the meta-modelling defining framework), and there was no formal semantics (it still does not have a standard one).

I did feel these were real issues and the biggest challenge in my teaching was to teach the students how to use quite a few UML models consistently and systematically through the development phases, from requirements modelling, through design modelling, to coding. Without imposing necessary formal aspects, it would be hard to solve this challenge, at least for me. I then decided to start a

small project on "Formal Use of UML in Software Development" and obtained a small grant from EPSRC[3] in 1999 to support research visits by Jifeng and Xiaoshan Li.[4] I also spent an 8-month sabbatical at UNU-IIST in 2001, then joined UNU-IIST as a full-time member of staff in 2002. The close collaboration of us three had started then.

The first result of the work is presented in the paper "Formal and use-case driven requirements analysis in UML" [36]. There, the functional requirements of the system to be developed are defined by a set of related *business processes*. Each business process is presented by a *use case* and the relations of the business processes are represented in a group of *use case diagrams*. The realisation of the business processes involves *objects* of *concepts* in the domain. The notation of mathematical graph theory is used to represent the concepts, the nodes, called *classes*, and the relations among the objects of classes, called *associations*, by edges between the classes. This is a formal representation of *UML class diagrams at the level of requirements*, which we call *conceptual class diagrams* (CCD). More precisely, the graph notation with only nodes and edges is not expressive enough, and logical constraints on properties of objects of classes and associations among classes are often required and imposed on a CCD. Consider a library system for example, the constraint that 'a **Copy** of a **Publication** which *is Held for* a **Reservation** must be a **Copy** of the **Publication** which *is reserved by* the **Reservation**' is not able to be depicted in a CCD. Therefore, a UML comment with such as natural English sentence is required, and it can be formally specified by a sentence in a first-order logic language, e.g.:

$$\forall c : \mathbf{Copy}, r : \mathit{Reservation}.$$
$$\mathit{isHeldFor}(c, r) \rightarrow \exists p : \mathbf{Publication}.(\mathit{isReservedBy}(p, r) \wedge \mathit{isCopyOf}(c, p))$$

In the above formalisation, the bold words are names of class names, the italic words association names, and variables range over instances (objects) of the corresponding classes (types). *Object constraint language* (OCL), as part of UML, can be used for specifying such constraints. A CCD with such logical constraints is called a *conceptual class model* (CMM).

The classes of a CCD, are called *conceptual classes* and they model the relevant *domain concepts*, instead of software classes. Therefore, a class in a CCD, in general, only has attributes and associations with other classes, but it does not have methods. The methods of a class will be designed to represent the responsibilities of the objects of the class for the realisation of the functional requirements elucidated in the use cases. This means what responsibilities are assigned to an object, represented as methods of the object, can only be decided in the design stage when the global functionalities of the use case are to be decomposed and delegated to the objects.

[3] Thanks to Cliff Jones for his support. He was our referee who we were allowed to recommend in the application.

[4] Xiaoshan was a mutual friend and close collaborator of Jifeng and me at the University of Macao and, sadly, passed away too young a few years ago.

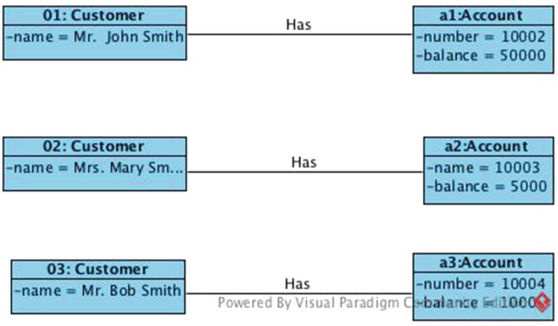
The semantics of a CCM is defined to be the set of allowable *object-diagrams* (OD) which satisfy the constraints specified by the CCD and additional logical constraints, to represent the *state space* of the system. For example, the class diagrams *SmallBank* in Fig. 1(a) and *BigBank* in Fig. 1(b) are conceptual models for a small bank system and a big bank system, respectively. The OD in Fig. 2(a) is a valid state of both *SmallBank* and *BigBank*, but the OD in Fig. 2(b) is a valid state of *SmallBank*, but it is not avalid state of *SmallBank*. It is easy to see that all valid states of *SmallBank* are valid states of *BigBank*.

(a) Small bank (b) Big bank

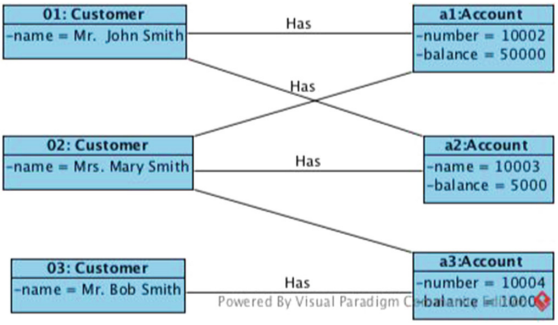
Fig. 1. Conceptual class diagrams

Functional requirements are represented as a *use case*, and the interactions between the *actors* and a use case of the system are regarded as *atomic actions* and their executions carry out transitions from valid states to valid states of the conceptual model, and thus their semantics are defined by the notion of designs in UTP. In this way, the notations of CCDs, ODs, use cases, and use case diagrams representing the functional architecture of the system to develop are unified, and their consistency is formalised. For example, suppose the current state of *BigBank* is in Fig. 2(b) and Mrs Mary Smith request system to transfer (say, represented by a use case action *transfer*()) 2,000 GBP from her account $a2$ to account $a3$ of Mr Bob Smith (that they share, say). The post-state after the execution is the state in Fig. 3.

An extension to the above work is presented in the paper [45]. There, a Java-style specification language is defined in which classes, attributes, and sub-classes (inheritance) in a conceptual class diagram are specified. Constraints on the state space modelled by object diagrams are specified by first-order predicate logic or OCL. For each use case, a *use case handler class* is introduced to declare *use case operations* as methods and the bodies of the methods are specified in UTP designs. Each *actor* of a use case is declared a class (corresponding to a process) and its method invokes methods of use case handler classes. Consistency between a conceptual class model and a use case model is formally defined as that the classes, attributes and inheritance relations can *fully support the specification of the methods in the use case handler classes*. A *refinement relation* on CCM is defined such that a CCM $\mathbf{CM}_1$ is a *refinement* of a CCM **CM** if $\mathbf{CM}_1$ supports any use case that **CM** supports. This captures the *use-case driven incremental and iterative* development process, known as the *Rational Unified Process* (RUP) to reflect the principle that *OO program design is mainly about the design of class structure*. For example, CCM *BigBank* is a refinement of CCM *SmallBank*, but not the other way around. The *transfer()* use case action is not supported by *SmallBank*.



(a) Small bank



(b) Big bank

Fig. 2. Object diagrams

To provide a higher level of abstraction and to better support the separation of design concerns, abstract semantic models for use case sequence diagrams are defined by *labelled transition systems* in the paper [48]. The semantics of general UML sequence diagrams is formalised in the papers [37], and this is used for the refinement from use case sequence diagrams to the sequence diagram in the design stage. This is presented in a follow-up work [37]. Further work on consistency among the use of UML models based on the models in the above papers can be found in [8,38,44,46]. Through the research, we have come to the understanding that *it is reasonable for UML not to have standard semantics as different semantics should be defined for modelling different aspects of the system in different applications, and the meta-modelling technology has its advantage in developing tools for model transformations.*

There was a period of quite active concerns about the rigorous use of UML[5] and there is a volume of work on formalising models for UML by translating them to formal notations, such as CSP and B. However, there has been little work on

[5] There was even a "Rigorous UML Group", although its members were not necessarily from the formal method community.

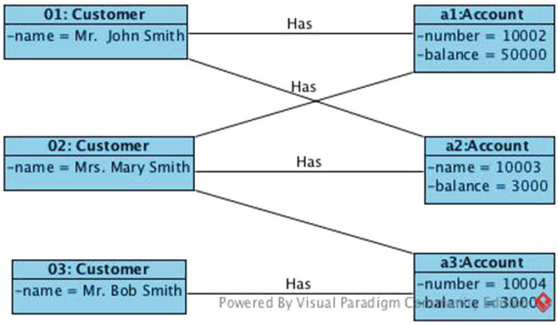


Fig. 3. Post-state of transfer operation

formal treatment of consistency and refinement of UML models. We are aware of the work in [14], which also noticed that UML lacks an explicit set of rules for ensuring that diagrams at different levels of abstraction are consistent. The authors defined such a set of rules, called diagram refinement, that is applicable to several kinds of UML diagrams (mostly for structural diagrams but also for use case diagrams). The work is based on the mathematical theory of graph homomorphisms.

3.2 Theory of Object-Oriented Semantics and Refinement

The work on formal and consistent use of UML discussed in the previous subsection can be classified into the so-called *lightweight formalism*. They alone do not address the complex issues of object-oriented programming languages, such as *side effects*, *polymorphism*, and *dynamic typing in method invocations*. Without well-studied semantics for object-oriented programs, it would have been difficult to address the refinement from requirements, through design, to implementation.

We thus proposed an abstract language for OO programming and called it the *rCOS OO language*, which is rather like the Java programming language. The semantic definition follows the ideas of UTP with the following features.

- To support type casting, dynamic type binding of method invocations and type safety analysis, variables are typed by *public*, *protected* and *private* and types of values of a variable can be of *primitive types* or *classes*. An object is defined recursively as a graph structure with nodes as objects and directed edges labelled by the names of the attributes in the source node, standing for the references from nodes to nodes (think of a UML object diagram). There is a unique root node of such a graph which represents the current object.
- At any time of its execution, the state of the program is an object graph, called a *state graph*, representing the object of the main program, i.e., the root is the object of the main class. The object nodes in a state graph contain the dynamic types of objects.

- The execution of a command of the program changes from such a state to another, by creating a new object and adding to the graph (e.g., to open a new account in the small bank system or the big bank system), changing the values of attributes of some objects in the graph (e.g., the *transfer()* action discussed earlier) or changing the edges of the graphs (e.g. make a customer to access an account in the big bank system). Therefore the semantics of the command (including method invocations) is defined as a relation between states in the UTP design.
- To support incremental program development, a class declaration is also defined as a design which modifies the changes in the static class structure of the program, which can be considered a textual formalisation of a UML class diagram. Class declaration is a development action done before program compilation.

Based on this semantic theory, *OO refinement* is defined at three levels. They are the *refinement of commands* including method invocations, *refinement of classes*, and *refinement of programs*. Class refinement also characterises *subtyping*. Refinement of programs including extension and modification of the class declarations of the program, as well as refinement of the main method and methods in the classes. This work is presented in the paper [24]. It was in this publication that the term "rCOS" was first used, standing for *Refinement Calculus of Object Systems*. The healthiness conditions **H1**–**H4** in Sect. 2.4 on the theory of program designs are inherited here, and based on them, algebraic laws of OO programs at the command level are also proven [68].

Further work on OO refinement is presented in [78] based on the semantic theory of OO programming. There, a set of refinement rules are given and shown to be *sound and relatively complete*. The first completeness theorem shows that without changing commands in the main method, any OO program can be transformed to a program in which there are only *inheritance relations* between classes without attributes which have types of classes (i.e., the edges in the corresponding UML class diagram are only inheritance associations). The second completeness states that any OO program can be refined to a non-OO program (an imperative procedural program) if equivalence transformation of the main method is also allowed.

It is worth emphasising that, in our theory, the General Responsibility Assignments Patterns (GRASP) for OO design [32] and refactoring rules proposed in [16] are shown to be refinement rules. These design patterns are very effective in OO design and maintenance, but to our best knowledge, there is little work on their study in a formal semantic theory. The GRASP approach is used to decompose a "grand functionality" of a class into "a number of functionalities" and assign these to *appropriate* classes.

The most effective pattern, which is thus most often used, is called the *Expert Pattern*. This assigns a responsibility to the information expert, i.e., the class which has information necessary to fulfil the responsibility. The refinement rule for Expert Pattern in rCOS is shown in Fig. 4. On the left of $\sqsubseteq$ in the figure, it specifies an operation with the functionality of *operation*() of class **C** which

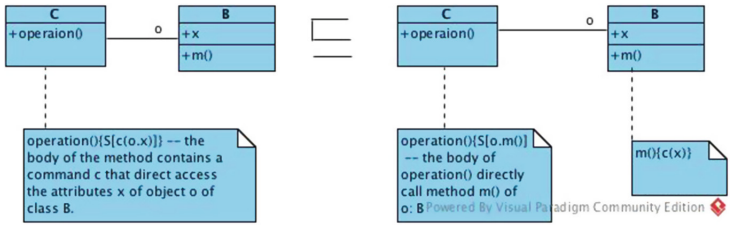


Fig. 4. Expert pattern as an OO refinement rule

information (data) x of class **B**. If the specified functionality can be realised (or refined) by a method $m()$ of **B**, the class model on the right $\sqsubseteq$ is in the figure is a refinement of the one on the right. Note that in rCOS bodies of methods are allowed to be specifications, instead of code.

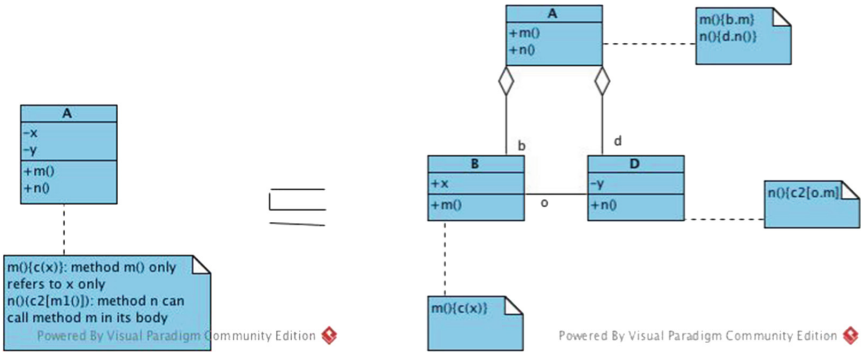
The *High Cohesion Pattern* of GRASP is, for the purpose of better reuse and maintenance, aimed at avoiding having classes in a design with too many unrelated functionalities. It is represented as a refinement rule shown in Fig. 5. The refinement is in general for class decomposition of a complex class to a composition of micro and logically cohesive classes. The pattern is explained as follows:

- assume class **A** in Fig. 5a contains two sets of attributes x and y (including role names of associations with class **C**);
- class **C** are given responsibilities represented by methods m and n, and m only refers to attributes x;
- we can decompose (i.e., refinement) **A** into the model in Fig. 5b which consists of three classes **A**, **B** and **D** such that **B** maintains x only and it is assigned with the responsibility m;
- class **A** is responsible for coordinating the responsibilities of classes **B** and **D**.

To decrease the overhead of object interactions and improve reuse and for easy maintenance, the *Low Coupling Pattern* is to have a model with fewer associations among classes. For example, the refined model in Fig. 5b can be further refined to the model in Fig. 6, which has lower coupling.

The refinement rules for GRASP can be used in the context of any larger class models that contain them. Furthermore, a class model $\mathcal{C}$ is a structural refinement of a class model $\mathcal{C}_1$, if $\mathcal{C}$ by be obtained by one of the following changes made to $\mathcal{C}_1$:

- adding a class,
- adding an attribute to a class,
- adding an association between two classes,
- increasing the multiplicity of a role of an association (that is equivalent to adding attributes at the level of program code),
- promoting an attribute of a subclass to its superclass,



(a) A low cohesion class

(b) Refined class model of high cohesion

Fig. 5. Refinement for class decomposition

– promoting an association of a subclass to its superclass,
– adding a method to a class, and
– promoting a method of a subclass to its superclass.

We call these refinement rules *OO structural refinement rules*. The systematic and rigorous study of OO structural refinement, together with the work on the formal use of UML, provides the theoretical confidence towards my understanding and improved my teaching of software development with UML [39]. Now we can see that the rCOS method provides a comprehensive use of UML with a formal OO semantic basis on UTP. Furthermore, OO structural refinement, such as the rules for the patterns of Expert, Low Coupling and High Cohesion characterise the essential features of *microservice architectures* [74], which are now popular in the software industry.

3.3 Component-Based Architecture Modelling

It was in the early 2000s when *component-based development* was causing the attention of the software engineering community and *component-based diagrams* were introduced into UML, although the term "component-based programming" had come much earlier. I remember one day when Jifeng called me to his office and said "we should extend our rCOS method to component-based programming". He showed me Szyperski's book [70] and said that he was reading it and would spend a week or so to finish it. I must confess that I could not read such a book that fast.

Naturally, our initial work on component-based modelling [23,43] extends the rCOS model for OO programs and formulates the key notions about component software in Szyperski's book. *Interfaces* are defined to be first-class model elements. An interface $\mathcal{I} = (M, A, O)$ consists of a list of class declarations M, a list of field variables A with their types declared in M, and a list of operations

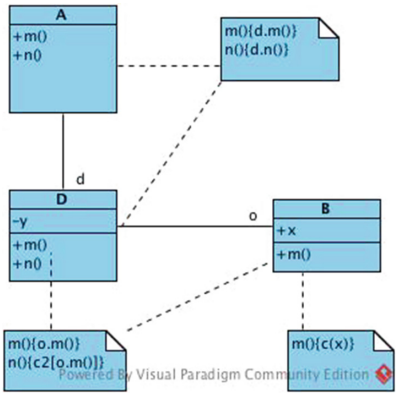


Fig. 6. A refinement of low coupling

O as methods specified in the rCOS OO language. The component specification is described by a *contract* that gives each operation a design. A *closed component* is an *implementation* of an interface in the rCOS OO language that has a *provided interface*. The implementation of an operation in the provided interface $\mathcal{I}$ can invoke services implemented in other components. The methods invoked by the operations in the provided interface form the *required interface* $\mathcal{R}$. Thus, an (open) component P is a parameterised program which takes a component implementing $\mathcal{R}$ as its input services. Therefore, the semantics of a component with a provided interface $\mathcal{I}$ and a required interface $\mathcal{R}$ is defined as a relation on the contracts of $\mathcal{I}$ and $\mathcal{R}$:

$$[\![P]\!] = \{(\mathcal{C}_\mathcal{I}, \mathcal{C}_\mathcal{R}) \mid (\mathcal{C}_\mathcal{I} \sqsubseteq P(\mathcal{C}_\mathcal{R})\}$$

where $P(\mathcal{C}_\mathcal{R})$ is the contract for the provided interface $\mathcal{I}$, calculated from the semantics of component program P with input services specified by the contract $\mathcal{C}_\mathcal{R}$. The relation $[\![P]\!]$ means that given any provided services which implement the required interface $\mathcal{R}$, the component with this input is a refinement of the contract $\mathcal{C}_\mathcal{I}$ of the provided interface $\mathcal{C}_\mathcal{I}$. The composition of interfaces, composition of components and refinements of components are defined.

In this model, the interactions between components are in general OO method invocations from a component to the components which provide it with the required services. To have distributed implementations of the components, middlewares, such as CORBA, are needed. However, the semantics of middleware are not formalised in this work. Further, a contract defined in the previous subsection specifies the static functionality of a component that does not require synchronisation when the operations are used. Such components are often used in the functional layer of a system. Processes and business rules are, however, accomplished by invoking particular sequences of operations. Also, synchronisations are needed when resources are shared. This means a synchronisation protocol using the functional operations must be imposed, often by composing a component in the functional layer and a component in the system layer.

To model synchronisation and interaction protocols, we define the notion of reactive design by introducing two fresh observables, *wait* and $wait'$, for synchronisation. A design D on an alphabet α is called a *reactive design* if it satisfies the healthiness condition $\mathcal{W}(D) = D$, where:

$$\mathcal{W}(D) =_{df}= (true \vdash wait' \wedge ((\alpha' = \alpha) \wedge (ok = ok'))) \lhd wait \rhd D$$

For writing a specification of the *interface contract* for a *reactive component*, we introduce the notation of *guarded designs* of the form $g\&D$, where D is a design and g is a Boolean expression of the alphabet, called the *guard* of the design. The semantics of $g\&D$ is defined as $D \lhd g \rhd (true \vdash wait' \wedge ((\alpha' = \alpha) \wedge (ok = ok'))$, where α is the alphabet of the design D. We can prove that all guarded designs are reactive designs, and $\mathcal{W}^2(D) = \mathcal{W}(D)$ for all designs.

For the *implementation* of a reactive component, we use a language of *guarded methods* in which each guarded method is of the form $g\&m(in; out)$ and the body method $m(in; out)$ is written as a command c in the rCOS OO language. The semantics of $g\&m(in; out)$ is defined as the guarded design $g\&\mathcal{W}(D_c)$, where D_c is the semantics of c, defined in the subsection immediately above.

The work in [23] also shows the unification and their separation of uses of *designs for local functionality specification*, *traces for reactive behaviours*, *failures for deadlock* and *divergences for livelock*.

More concretely speaking, a (reactive) *contract* serves as a specification of an interface and now is defined as a tuple $\mathcal{C} = (\mathcal{I}, Init, \mathcal{S}, \mathcal{P})$ of an interface $\mathcal{I}$, an *initial condition* $Init$ specifying the allowable starting states, a *specification* $\mathcal{S}$ specifying each operation in $\mathcal{I}$ as a guarded design, and *protocol* $\mathcal{P}$ which is a set of sequences $\langle ?m_1(x_1), \ldots, ?m_k(x_k) \rangle$ of *invocations* to the interface operations acceptable by the interface. With the specification $\mathcal{S}$ of a contract, the *divergence set* $\mathcal{D}_\mathcal{C}$ and the *failure set* $\mathcal{F}_\mathcal{C}$ of a contract $\mathcal{C}$, as those defined for CSP processes in [63], are defined and the triple $(\mathcal{D}_\mathcal{C}, \mathcal{F}_\mathcal{C}, \mathcal{P})$ is called the model of *dynamic behaviour* for the contract. The *trace set* of a contract can be defined too.

A contract $\mathcal{C}_1$ is refined by a contract $\mathcal{C}_2$ if:

- the attributes of the two contracts are the same;
- the set O_1 of operations of $\mathcal{C}_1$ is a subset of the operations of $\mathcal{C}_2$, i.e. $\mathcal{C}_2$ provides no fewer services;
- $\mathcal{C}_2$ is not more likely to diverge, $\mathcal{D}_{\mathcal{C}_2} \subseteq \mathcal{D}_{\mathcal{C}_1}$;
- $\mathcal{C}_2$ is not more likely to block the client, $\mathcal{F}_{\mathcal{C}_2} \subseteq \mathcal{F}_{\mathcal{C}_1}$.

The refinement of one contract by another can be proven by *downward simulation* and *upward simulation* [19]. Another important point is that the specification $\mathcal{S}$ and the protocol $\mathcal{P}$ might not be *consistent* and the consistency can be checked. With the theory of contracts and refinements for components, the meaning of rCOS is extended to *Refinement of Component and Object Systems*.

In the keynote paper [22], special kinds of components, including *coordinators*, *connectors* and *controllers* are characterised, and the concepts in the semantic theory are related to notations used in software engineering. In the paper [7], the relation between specification $\mathcal{S}$ and the protocol $\mathcal{P}$ is further elaborated

and introduced the notion of *processes* as a special kind of components. In this way, functional (or service) components are specified in the rCOS OO language without using guards, and interaction protocols are modelled as processes at the system level to control synchronisation. The behaviour of a process can then be specified by *traces*, *failures*, or *divergences* according to the properties to be analysed, and by input/out automata [55], *UML state diagrams* or especially by *interface automata* [12,13]. *This provides explicit support to multi-view modelling and separation of concerns to different modelling notations and underlying theories to be used consistently in component-based software development.*

3.4 rCOS Support for Model-Driven Development

To apply the rCOS method, we embed the modelling notations into software development processes and use them consistently according to the underlying theory of rCOS. To this end, we identify activities of the RUP process for model-driven development and associate them with modelling notations defined in rCOS.

In a top-down process, a requirements model of the use cases is identified and each use case is modelled as a component in the following steps:

1. their provided interfaces are the interactions with the actors and field attributes are modelled by conceptual class diagrams (which can be unified into a single one);
2. interaction protocols of use cases are modelled by sequence diagrams, and dynamic behaviour by state diagrams;
3. the functionalities of interface operations are specified by designs (pre- and post-conditions), focusing on what new objects are created, which attributes are modified, and the new links of objects that are formed; and
4. the requirements architecture is modelled by UML *component-based diagrams* reflecting the relations among use cases in the use case diagram.

A design process consists of the following modelling steps:

1. it takes each use case and designs each of its provided operations according to its pre- and post-conditions by using the OO refinement rules, the four patterns of GRASP in particular;
2. this decomposes the functionality of each use case operation into internal object interaction and computation, thus refining the use case sequence diagram into a *design sequence diagram* of the use case [10];
3. in the process of decomposition of the functionality of use-case operations to internal object interaction and computation, the requirements class model is refined into a *design class model* by adding methods and visibilities in classes according to responsibility assignments and directing of method invocations [10];
4. identify some of the objects in a design sequence diagram as the *component controller*, satisfying *six semantic invariant properties* (which can be checked by model checking) and then transform the design sequence diagram to a component-sequence diagram [34];

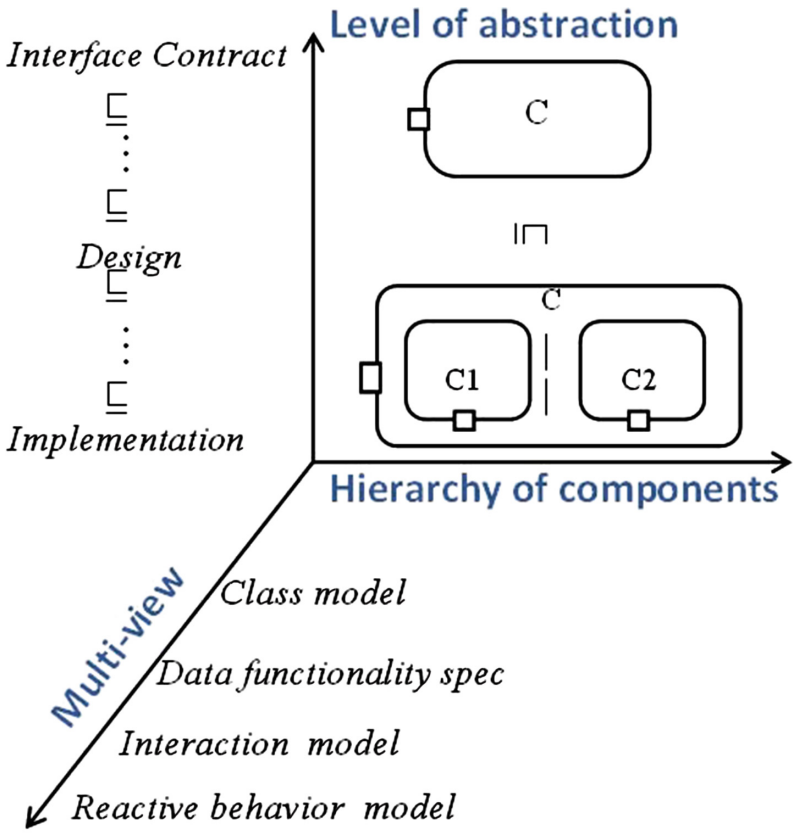


Fig. 7. Features of the rCOS modelling paradigm

5. a component diagram for each use case is generated (automatically) which is a decomposition of the use-case component in the requirements model to a composition of sub-components, and then the whole *component-based architecture model* at the requirements level is decomposed into a *component-based design architecture model* [34];
6. the coding from the design architecture model is not difficult and can be largely automatic [53].

The key features of the rCOS modelling paradigm, as shown in Fig. 7, are being *refinement-driven*, *hierarchically component-based*, and with *multi-view modelling in multiple notations with a unified semantics*. The component-based design process from a component-based requirements model is depicted in Fig. 8. The development process is applied to the CoCoMe benchmark problem [9] and the details are elaborated in [10]. The modelling and analysis framework of rCOS can also be applied for component integration in bottom-up development processes or a mixed top-down and bottom-up process, allowing the use and reuse

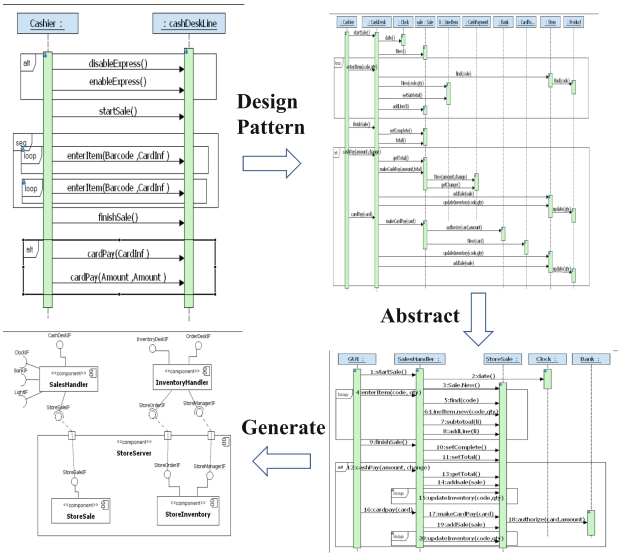


Fig. 8. Transformations from requirements model to design model

of existing models and implementations of components, as long as they have explicitly specified interfaces. Interface contracts in rCOS of component-based architecture support the characterisation of different security threats and for the design of protection and recovery from damages of security attacks [50]. The rCOS model of component-based architecture supports the design of fault-tolerant mechanisms so as restrict the propagation of *errors* caused by *faults* and *failures* [47] across interfaces of components [77].

The UML diagrams are translated to the rCOS notation by the rCOS Modeller. The tool for transformation from design sequence diagrams to component-sequence diagrams is presented in [34]. A tool for automatic prototyping from the requirements model is presented in [75]. The use of *low coupling* with *high cohesion* patterns supports the *microservice architecture* design [71,74].

4 Extension of rCOS to Model HCPS

The research on rCOS is not yet finished (I hope). We currently have a project on "Theory of Human-Cyber-Physical Computing and Software Defined Methodology". We are extending the rCOS component-based modelling notation to model system architectures of HCPS. We take the view of the architecture of an HCPS as comprising of *cyber systems*, a *communication network*, *physical processes*, *human processes*, and *interfaces*, where:

– the physical processes include, for example, mechanical, electrical and chemical processes;

- the cyber systems (information systems) are computing systems where:
  - some cyber systems (i.e., information systems) are responsible for data collection and processing, and
  - some cyber systems (controlling systems) are responsible for taking control decisions, based on the information provided by the information systems, to control physical processes;
- the human processes make control decisions based on the information provided by the information systems to physical processes;
- the interfaces are middleware systems between the physical systems, cyber systems and humans, including sensors and actuators, A/C and C/A converters, etc.;
- the sensors sense the physical processes, collect data about the behaviours of the physical processes and the data are transmitted to the information systems through the network;
- computer control decisions of the controlling systems and human control decisions are transmitted in the form of control commands through the network to corresponding actuators to carry out the control actions.

There is system software for the *coordination* and *orchestration* of the behaviours of the component systems and for *scheduling* the *physical, network, hardware, human*, and *software resources*. A particular need is components which are responsible for the switching of control between human and computer controllers.

The initial work is to extend the rCOS model of interfaces to *cyber-physical interfaces* (*CP-interfaces*) or *hybrid interfaces*. A CP-interface includes as its field variables both *signals* representing information on the states of physical processes, as well as program variables; and it includes both program operations and signals for interactions with the environment of the component. Also, a signal can be either discrete or continuous. A contract of a CP-interface then consists of a provided CP interface, a required CP interface, and a specification describing the functionality of the program operations and the behaviour (in differential or difference equations) of the signals in the interfaces.

With the above considerations, we propose a horizontally and vertically open and hierarchical component-based approach for a systems of systems model of HCPS architecture. This model supports top-down and bottom-up development and interface-based black-box system integration. It also supports continuous maintenance through sub-system upgrades and replacements of sub-systems. Furthermore, services and functions of an existing HCPS can be used as infrastructure to develop further sub-systems to be integrated into the system in a plug-and-play manner, and thus the architecture is ever evolving. The architecture style also supports component based design of security and fault-tolerance.

The dynamic behaviour of such a contract corresponds to two *hybrid input/output* automata [54], one for the provided CP interface and one for the required CP interface. It should be possible to specify them in Hybrid CSP [20] and analyse them in Hybrid Hoare logic [79] both developed with fundamental contributions from Jifeng. The initial ideas on the extension are presented in [40,42,52] and a proof of concept example is given in [60]. The *hybrid relation*

calculus [21] proposed by Jifeng would also provide a theoretical foundation for further development of this work. Longer-term research is to link the method to that of the semantics of Simulink and SysML.

A significant challenge to modelling HCPS is that there is no computational model and theory for human interactions with cyber and physical systems. We are proposing a *human-cyber-physical automata* (HCPA). In this model, the human behaviour is represented by a neural network and the controller for control switching between human and machine is modelled as an oracle with a learning model too. It is important to note that we are not modelling general human intelligence, but the behaviour of a human in a given application when carrying out their tasks instead. The research on the full theory will involve tackling the difficulties in composing traditional computation models with AI models. An initial model of HCPA is defined with only one human process to control a physical process in collaboration with digital controllers. This model and a proof of concept case study are presented in the invited talk [76]. For the research problems in this project, we refer the reader to the editorial paper [52] and the lecture notes in [40]. The research will heavily involve the controllability and composability of AI systems, their composability with traditional computational systems, and the trustworthiness of these hybrid systems.

5 Conclusions and Acknowledgements

This paper has presented a summary of the development of the rCOS method, showing that Hoare and He's UTP is the theoretical root of rCOS. The summary focuses on a unified understanding of different formal theories with the belief that *uniformity in theory is for consistent use of formal methods to support the separation of design concerns in software systems development.*

Through the discussion, we show how the ideas, theoretical results and techniques in different publications on the rCOS method are linked and how they are related to well-studied formal methods.[6] The rCOS method helps to narrow the gap from the semantic theories to the engineering technologies and supports their consistent use in practice. The lecture notes in [39], although never officially published, have been used since 1998 and the teaching has been improved along with the development of the rCOS method. A plan and an architecture for the rCOS tool development were carefully proposed in [11,49], although it was not fully developed due to the lack of stable human resources. The current status of the tool is available at https://rise-swu.cn/rCOS.

Acknowledgements. There is a long list of names of collaborators who have contributed to the research on rCOS and I would like to thank them. In alphabetic order, the list includes Xin Chen, Zhenbang Chen, Ruzhen Dong, Dan Van Hung, Bin Lei, Dan Li, Xiaoshan Li, Jing Liu, Quan Long, Charles Morriset, Zongyan Qiu, Anders Ravn, Martin Schäf, Leila Silva, Volker Stolz, Shuling Wang, Ji Wang, Jing

[6] What a nice coincidence it is that there are 80 references in this paper to celebrate Jifeng's 80th birthday.

Yang, Lu Yang, Yilong Yang, Naijun Zhan, Miaomiao Zhang, Liang Zhao. They all spent time at UNU-IIST, at different periods, as fellows, PhD students, postdoctoral research fellows, or visitors. We all owe a big thanks to Jifeng for his guidance and/or influence over the years. We all congratulate him on his academic achievements, and wish him a very happy 80th birthday!

I would also like to thank Jonathan Bowen and Shmuel Tyszberowicz for their careful reading and comments on draft versions of this paper.

References

1. Abrial, J.R.: The B-Book: Assigning Programs to Meanings. Cambridge University Press, Cambridge (1996)
2. Abrial, J.R.: Modeling in Event-B: System and Software Engineering. Cambridge University Press, Cambridge (2010)
3. Back, R.J.R., von Wright, J.: Trace refinement of action systems. In: Jonsson, B., Parrow, J. (eds.) CONCUR 1994. LNCS, vol. 836, pp. 367–384. Springer, Heidelberg (1994). https://doi.org/10.1007/978-3-540-48654-1_28
4. Baeten, J.C.M., Bravetti, M.: A generic process algebra. In: Algebraic Process Calculi: The First Twenty Five Years and Beyond. BRICS Notes Series NS-05-3 (2005)
5. Brooks, F.P.: Learn the hard way - a history 1845–1980 of software engineering. In: Keynote at 40th International Conference on Software Engineering (ICSE 2018), Gothenburg, Sweden, 27 May–3 June 2018 (2018). https://www.icse2018.org
6. Butler, R.W.: What is formal methods? (2001). https://shemesh.larc.nasa.gov/fm/fm-what.html
7. Chen, X., He, J., Liu, Z., Zhan, N.: A model of component-based programming. In: Arbab, F., Sirjani, M. (eds.) FSEN 2007. LNCS, vol. 4767, pp. 191–206. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-75698-9_13
8. Chen, X., Liu, Z., Mencl, V.: Separation of concerns and consistent integration in requirements modelling. In: van Leeuwen, J., Italiano, G.F., van der Hoek, W., Meinel, C., Sack, H., Plášil, F. (eds.) SOFSEM 2007. LNCS, vol. 4362, pp. 819–831. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-69507-3_71
9. Chen, Z., Li, X., Liu, Z., Stolz, V., Yang, L.: Harnessing rCOS for tool support—the CoCoME experience. In: Jones, C.B., Liu, Z., Woodcock, J. (eds.) Formal Methods and Hybrid Real-Time Systems. LNCS, vol. 4700, pp. 83–114. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-75221-9_5
10. Chen, Z., Liu, Z., Ravn, A.P., Stolz, V., Zhan, N.: Refinement and verification in component-based model driven design. Sci. Comput. Program. **74**(4), 168–196 (2009)
11. Chen, Z., Liu, Z., Stolz, V., Yang, L., Ravn, A.P.: A refinement driven component-based design. In: 12th International Conference on Engineering of Complex Computer Systems (ICECCS 2007), pp. 277–289. IEEE Computer Society (2007)
12. De Alfaro, L., Henzinger, T.: Interface automata. ACM SIGSOFT Softw. Eng. Notes **26**(5), 109–120 (2001)
13. Dong, R., Zhan, N., Zhao, L.: An interface model of software components. In: Liu, Z., Woodcock, J., Zhu, H. (eds.) ICTAC 2013. LNCS, vol. 8049, pp. 159–176. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39718-9_10
14. Faitelson, D., Tyszberowicz, S.S.: UML diagram refinement (focusing on class- and use case diagrams). In: Uchitel, S., Orso, A., Robillard, M.P. (eds.) Proceedings of

the 39th International Conference on Software Engineering, ICSE, Buenos Aires, Argentina, pp. 735–745. IEEE/ACM (2017). https://doi.org/10.1109/ICSE.2017.73
15. Floyd, R.W.: Assigning meanings to programs. Proc. Am. Math. Soc. Symposia Appl. Math. **19**, 19–31 (1967)
16. Fowler, M.: Refactoring - Improving the Design of Existing Code. Addison-Wesley, Menlo Park (1999)
17. Gierz, G., Hofmann, K.H., Keimel, K., Lawson, J.D., Mislove, M., Scott, D.S.: Continuous Lattices and Domains, Encyclopedia of Mathematics and its Applications, vol. 93. Cambridge University Press (2003)
18. Goguen, J., Burstall, R.: Institutions: abstract model theory for specification and programming. J. ACM **39**(1), 95–146 (1992)
19. He, J.: Simulation and process refinement. Formal Aspect Comput. **1**(3) (1989)
20. He, J.: From CSP to hybrid systems. In: Roscoe, A.W. (ed.) A Classical Mind: Essays in Honour of C. A. R. Hoare, chap. 11, pp. 171–189. International Series in Computer Science, Prentice Hall, New York (1994)
21. He, J., Qin, L.: A hybrid relational modelling language. In: Gibson-Robinson, T., Hopcroft, P., Lazić, R. (eds.) Concurrency, Security, and Puzzles. LNCS, vol. 10160, pp. 124–143. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-51046-0_7
22. Jifeng, H., Li, X., Liu, Z.: Component-based software engineering. In: Van Hung, D., Wirsing, M. (eds.) ICTAC 2005. LNCS, vol. 3722, pp. 70–95. Springer, Heidelberg (2005). https://doi.org/10.1007/11560647_5
23. He, J., Li, X., Liu, Z.: A theory of reactive components. Electron. Notes Theor. Comput. Sci. **160**, 173–195 (2006)
24. He, J., Liu, Z., Li, X.: rCOS: a refinement calculus of object systems. Theor. Comput. Sci. **365**(1–2), 109–142 (2006)
25. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM **12**(10), 576–580 (1969)
26. Hoare, C.A.R.: Communicating sequential processes. Commun. ACM **21**(8), 666–677 (1978)
27. Hoare, C.A.R., He, J.: Unifying Theories of Programming. Series in Computer Science, Prentice Hall, London (1998)
28. Jackson, D.: Software Abstractions: Logic, Language, and Analysis. MIT Press, Cambridge (2006)
29. Jones, C.B.: Systematic Software Development using VDM. International Series in Computer Science, Prentice Hall, Englewood Cliffs (1990)
30. Lamport, L.: The temporal logic of actions. ACM Trans. Program. Lang. Syst. **16**(3), 872–923 (1994)
31. Lamport, L.: Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers. Addison-Wesley, Boston (2002)
32. Larman, C.: Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process, 2nd edn. Prentice Hall, Upper Saddle River (2001)
33. Lee, E.A.: The past, present and future of cyber-physical systems: a focus on models. Sensors **15**(3), 4837–4869 (2015)
34. Li, D., Li, X.S., Liu, Z.M., Stolz, V.: Automated transformations from UML behavior models to contracts. Sci. China Inf. Sci. **57**(12), 1–17 (2014). https://doi.org/10.1007/s11432-014-5159-8
35. Li, D., Li, X., Stolz, V.: QVT-based model transformation using XSLT. SIGSOFT Softw. Eng. Notes **36**, 1–8 (2011)

36. Li, X., Liu, Z., He, J.: Formal and use-case driven requirement analysis in UML. In: 25th International Computer Software and Applications Conference (COMPSAC 2001), Invigorating Software Development, Chicago, IL, USA, 8–12 October 2001, pp. 215–224 (2001)
37. Li, X., Liu, Z., He, J.: A formal semantics of UML sequence diagram. In: 15th Australian Software Engineering Conference (ASWEC 2004), Melbourne, Australia, 13–16 April 2004, pp. 168–177. IEEE Computer Society (2004)
38. Li, X., Liu, Z., He, J.: Consistency checking of UML requirements. In: 10th International Conference on Engineering of Complex Computer Systems, pp. 411–420. IEEE Computer Society (2005)
39. Liu, Z.: Software development with UML. Technical report. Technical Report 259, UNU-IIST: International Institute for Software Technology, the United Nations University, Macao (2002)
40. Liu, Z., Bowen, J.P., Liu, B., Tyszberowicz, S., Zhang, T.: Software abstractions and human-cyber-physical systems architecture modelling. In: Bowen, J.P., Liu, Z., Zhang, Z. (eds.) SETSS 2019. LNCS, vol. 12154, pp. 159–219. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-55089-9_5
41. Liu, Z., Chen, X.: Model-driven design of object and component systems. In: Liu, Z., Zhang, Z. (eds.) SETSS 2014. LNCS, vol. 9506, pp. 152–255. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-29628-9_4
42. Chen, X., Liu, Z.: Towards interface-driven design of evolving component-based architectures. In: Hinchey, M.G., Bowen, J.P., Olderog, E.-R. (eds.) Provably Correct Systems. NMSSE, pp. 121–148. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-48628-4_6
43. Liu, Z., Jifeng, H., Li, X.: Contract oriented development of component software. In: Levy, J.-J., Mayr, E.W., Mitchell, J.C. (eds.) TCS 2004. IIFIP, vol. 155, pp. 349–366. Springer, Boston, MA (2004). https://doi.org/10.1007/1-4020-8141-3_28
44. Liu, Z., He, J., Li, X.: Towards a rigorous approach to UML-based development. In: Mota, A., Moura, A.V. (eds.) Proceedings of the Seventh Brazilian Symposium on Formal Methods, SBMF 2004. Electronic Notes in Theoretical Computer Science, Recife, Pernambuco, Brazil, 29 November–1 December 2004, vol. 130, pp. 57–77. Elsevier (2004)
45. Liu, Z., Jifeng, H., Li, X., Chen, Y.: A relational model for formal object-oriented requirement analysis in UML. In: Dong, J.S., Woodcock, J. (eds.) ICFEM 2003. LNCS, vol. 2885, pp. 641–664. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-39893-6_36
46. Liu, Z., He, J., Liu, J., Li, X.: Unifying views of UML. In: de Boer, F.S., Bonsangue, M.M. (eds.) Proceedings of the Workshop on the Compositional Verification of UML Models, CVUML 2003, Electronic Notes in Theoretical Computer Science, San Francisco, CA, USA, 21 October 2003, vol. 101, pp. 95–127. Elsevier (2003)
47. Liu, Z., Joseph, M.: Specification and verification of fault-tolerance, timing, and scheduling. ACM Trans. Program. Lang. Syst. **21**(1), 46–89 (1999)
48. Liu, Z., Li, X., He, J.: Using transition systems to unify UML models. In: George, C., Miao, H. (eds.) ICFEM 2002. LNCS, vol. 2495, pp. 535–547. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-36103-0_54
49. Liu, Z., Mencl, V., Ravn, A.P., Yang, L.: Harnessing theories for tool support. In: Proceedings of the Second International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2006), pp. 371–382. IEEE Computer Society (2006)

50. Liu, Z., Morisset, C., Stolz, V.: A component-based access control monitor. In: Margaria, T., Steffen, B. (eds.) ISoLA 2008. CCIS, vol. 17, pp. 339–353. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-88479-8_24
51. Liu, Z., Qiu, Z.: Introduction to Mathematical Logic - The Natural Foundation for Computer Science and System. China Science Publishing & Media Ltd. (2022). (in Chinese)
52. Liu, Z., Wang, J.: Human-cyber-physical systems: concepts, challenges, and research opportunities. Front. Inf. Technol. Electron. Eng. **21**(11), 1535–1553 (2020). https://doi.org/10.1631/FITEE.2000537
53. Long, Q., Liu, Z., Li, X., He, J.: Consistent code generation from UML models. In: Australian Software Engineering Conference, pp. 23–30. IEEE Computer Society (2005)
54. Lynch, N., Segala, R., Vaandrager, F.: Hybrid I/O automata. Inf. Comput. **185**, 105–157 (2003)
55. Lynch, N.A., Tuttle, M.R.: An introduction to input/output automata. CWI Q. **2**(3), 219–246 (1989)
56. McCarthy, J.: Recursive functions of symbolic expressions and their computation by machine. Part I. Commun. ACM **3**(4), 184–219 (1960)
57. Milner, R.: Communication and Concurrency. International Series in Computer Science, Prentice Hall, New York (1989)
58. Morgan, C.: Programming from Specifications. International Series in Computer Science, Prentice Hall, New York (1994/1998). https://www.cs.ox.ac.uk/publications/books/PfS/
59. Object Management Group: Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, Version 1.1 (2009)
60. Palomar, E., Chen, X., Liu, Z., Maharjan, S., Bowen, J.P.: Component-based modelling for scalable smart city systems interoperability: a case study on integrating energy demand response systems. Sensors **16**(11), 1810 (2016). https://doi.org/10.3390/s16111810
61. Plotkin, G.D.: The origins of structural operational semantics. J. Logic Algebraic Program. **60**(61), 3–15 (2004)
62. Randell, B.: Fifty years of software engineering or the view from Garmisch. In: Keynote at 40th International Conference on Software Engineering (ICSE 2018), Gothenburg, Sweden, 27 May–3 June 2018 (2018). https://www.icse2018.org
63. Roscoe, A.W.: Theory and Practice of Concurrency. International Series in Computer Science, Prentice Hall, Engelwood Cliffs (1997)
64. Roscoe, A.W., Hoare, C.A.R.: The laws of OCCAM programming. Theor. Comput. Sci. **60**(2), 177–229 (1988). https://doi.org/10.1016/0304-3975(88)90049-7
65. Sangiovanni-Vincentelli, A., Damm, W., Passerone, R.: Taming dr. frankenstein: contract-based design for cyber-physical systems. Eur. J. Control **18**(3), 217–238 (2012)
66. Schneider, S.: The B-Method: An Introduction. Cornerstones of Computing Series, Palgrave Macmillan, London (2001)
67. Scott, D., Strachey, C.: Toward a Mathematical Semantics for Computer Languages. No. PRG-6 (1971)
68. Silva, L., Sampaio, A., Liu, Z.: Laws of object-orientation with reference semantics. In: Cerone, A., Gruner, S. (eds.) Sixth IEEE International Conference on Software Engineering and Formal Methods, SEFM 2008, Cape Town, South Africa, 10–14 November 2008, pp. 217–226. IEEE Computer Society (2008). https://doi.org/10.1109/SEFM.2008.29

69. Spivey, J.M.: The Z Notation: A Reference Manual, 2nd edn. Prentice Hall, New York (1992)
70. Szyperski, C.: Component Software: Beyond Object-Oriented Programming, 2nd edn. Addison-Wesley Longman Publishing Co., Inc., Boston (2002)
71. Tyszberowicz, S., Heinrich, R., Liu, B., Liu, Z.: Identifying microservices using functional decomposition. In: Feng, X., Müller-Olm, M., Yang, Z. (eds.) SETTA 2018. LNCS, vol. 10998, pp. 50–65. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-99933-3_4
72. Wang, J., Zhan, N., Feng, X., Liu, Z.: Overview of formal methods. J. Softw. **30**(1), 33–61 (2019). (in Chinese)
73. Wing, J.M.: A specifier's introduction to formal methods. Computer **23**(9), 8–22 (1990)
74. Xiong, J.L., Ren, Q.R., Tyszberowicz, S.S., Liu, Z., Liu, B.: MSA-lab: an integrated design platform for model-driven development of microservices. J. Softw. (2023). https://doi.org/10.13328/j.cnki.jos.006813. (in Chinese)
75. Yang, Y., Li, X., Ke, W., Liu, Z.: Automated prototype generation from formal requirements model. IEEE Trans. Reliab. **69**(2), 632–656 (2020)
76. Zhang, M., Liu, W., Tang, X., Du, B., Liu, Z.: Human-cyber-physical automata and their synthesis. In: Seidl, H., Liu, Z., Pasareanu, C.S. (eds.) ICTAC 2022. LNCS, vol. 13572, pp. 36–41. Springer, Cham (2022). https://doi.org/10.1007/978-3-031-17715-6_4
77. Zhang, M., Liu, Z., Morisset, C., Ravn, A.P.: Design and verification of fault-tolerant components. In: Butler, M., Jones, C., Romanovsky, A., Troubitsyna, E. (eds.) Methods, Models and Tools for Fault Tolerance. LNCS, vol. 5454, pp. 57–84. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-00867-2_4
78. Zhao, L., Liu, X., Liu, Z., Qiu, Z.: Graph transformations for object-oriented refinement. Formal Aspects Comput. **21**(1–2), 103–131 (2009)
79. Zou, L., Zhan, N., Wang, S., Fränzle, M., Qin, S.: Verifying simulink diagrams via a hybrid hoare logic prover. In: Ernst, R., Sokolsky, O. (eds.) Proceedings of the International Conference on Embedded Software, EMSOFT 2013, Montreal, QC, Canada, 29 September–4 October 2013, pp. 9:1–9:10. IEEE (2013). https://doi.org/10.1109/EMSOFT.2013.6658587
80. Şerbănuţă, T.F., Rosu, G., Meseguer, J.: A rewriting logic approach to operational semantics. Inf. Comput. **207**(2), 305–340 (2009)