Multi-dimensional Abstraction and Decomposition for Separation of Concerns

Zhiming Liu(✉), Jiadong Teng(✉), and Bo Liu(✉)

School of Computer and Information Science, Southwest University, Chongqing, China
{zhimingliu88,liubocq}@swu.edu.cn, swu20201518@email.swu.edu.cn

Abstract. Model-driven engineering (MDE) or model-driven architecture (MDA) holds significant appeal for the software industry. Its primary aim is to address software complexity by enabling automated model creation and transformation. Consequently, many software development firms are actively seeking integrated development platforms (IDP) to enhance automation within their software production processes. However, the adoption of MDE and the utilisation of IDPs remain low, with doubts surrounding their success. To tackle this issue, this paper uses the formal refinement of component and object systems (rCOS) as a framework to identify different types of requirements and their relationships, with the goal of supporting MDE. We emphasise the necessity for families of formal languages and transformations among them, as well as the indispensability of architecture modelling and refinement in MDE. Furthermore, to enhance the handling of changes during the development and operation of systems, there is a paramount need for formal methods that facilitate abstractions and decompositions, leading to a multi-dimensional separation of concerns.

Keywords: Abstraction · Refinement · Decomposition · Separation of Concerns · Architecture modelling · Software Complexity

1 Introduction

Software engineering has focused on the development, study, and practice of methods for mastering inherent complexity [6–8] in the software development activities of requirements analysis, development processes, design and implementation (also called software construction), verification and validation, respectively. The goal has been to improve software comprehensibility, support reuse, enable evolution, and increase automation of development. In this paper, a *method* mainly consists of its *underlying theory*, a suite of *sound techniques*, and related *tool supports* (also called a platform) to the use of these techniques. Furthermore, a *formal method* (FM) refers to a method whose underlying theory is a *formal theory* consisting of a *formal language*, a *proof system*, and *model theory*, essentially constituting a *formal logic system* [48].

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

H. Hermanns et al. (Eds.): SETTA 2023, LNCS 14464, pp. 152–170, 2024.
https://doi.org/10.1007/978-981-99-8664-4_9

Some formal methods (FMs) offer languages for modelling specific aspects and their properties, while others provide languages for specifying various types of requirements. An aspect represents concerns related to particular types of requirements, including data, functionality, concurrency, communication, and more.

Separation of Concerns by Abstractions. An abstraction allows developers to simplify complex or domain-specific realities, focusing only what is essential. It introduces a layer of indirection between a general idea and specific implementations. By showcasing the core attributes of a problem, abstractions make problem clearer. This clarity aids in managing complexity, letting developers concentrate on a single aspect without being bogged down by intricate details. Such a method promotes a step-by-step approach to understanding and problem-solving. It also enhances reusability, as either the high-level problem or its solution can be reused in various scenarios, given that their connection is traceable. Automating tasks becomes feasible, especially when translating from an abstract to a concrete representation. Examples of these layers of indirection in software include pointers, APIs, the symbolic assembler that transform assembly programs to programs in machine language, the interpreter or compiler which translates a program in a high-level programming language to an machine language program, as well as concepts like super- and sub- class in object-oriented programming and super- and sub- type in type theory.

Separation of Concerns by Decompositions. Decompositions are essential strategies for mastering software complexity, involving the process of breaking down software into smaller, more manageable, clear, and well-defined components. Decomposition allows developers to focus on understanding and perfecting individual parts rather than being overwhelmed by the entire system. This not only improves software comprehensibility, as each part becomes self-contained with a specific responsibility, but also supports reuse. These components can be designed in a way that enables their use in multiple places within the same application or even across different projects, eliminating the need to recreate the same functionality. In terms of software evolution, decompositions allow for easier scalability and adaptability. If a single component needs to be updated or replaced, it can be done without disrupting the entire system. Finally, decompositions lend themselves well to the automation of development processes. Tools can be better designed for the automatic construction and testing of smaller and simpler components, as well as for hierarchical components and system integration with clearly defined interfaces.

Process of Decomposition and Refinements. A *top-down development (sub)process* is a sequence of decompositions and implementation mechanisms or representations of abstractions, which are termed *refinements* of abstractions. Conversely, a the bottom-up (sub)process involves a sequence of activities centred around the integration of subcomponents, each of which has been developed through refinements. For large-scale systems, the vast design space encompasses the processes of decompositions, refinements, and compositions. Moreover, the requirements for decomposition and refinement are intricate [40,47]. Accordingly, it is essential to incorporate large-scale reusable components, ensure low coupling and high cohesion, and to facilitate potent, non-invasive adaptation and customisation capabilities. It is also crucial to maintain trace-

ability of refinements and decompositions throughout the software lifecycle to minimise the impact of changes and substitutability.

Existing Approaches. To address the above challenges and requirements, substantial research has been made in the areas of decomposition and abstraction. Notably, *modularisation* is an important technique for partitioning a system into modules, premised on related functional units [14,41]. This approach is also referred to as algorithm-based decomposition. On the other hand, *object-oriented decomposition* adopts a distinct philosophical stance, envisioning a system as an ensemble of interacting objects [3,6]. These objects are software representations congruent with those discerned in the application domain. In light of their promising potential to enhance reusability, enable independent development, and facilitate standalone deployment, *component-based development* (CBD) [46] and *service-oriented architecture* (SOA) [5] are increasingly gaining prominence in contemporary academic and industry discussions.

Aims and Objectives of the Paper. Despite advancements in software engineering research, significant challenges persist in current model-driven engineering (MDE) frameworks to abstraction and decomposition. Many of these challenges arise from the inability of these approaches to effectively separate concerns as highlighted in previous literature [16,24,47]. These deficiencies have caused them to fall short of achieving the primary objectives of MDE as described in the referenced article [16]. We identify the following limitations in the current modelling approaches:

(1) Insufficient tool support for life-cycle management. Changes in the generated models or some constructed models often lead to round-trip issues.
(2) Ineffectiveness in reducing the sensitivity of primary artefacts to various changes, such as personnel adjustments, requirement updates including those in the development platform, and changes in deployment platforms.
(3) Limited flexibility to accommodate development team members with diverse methodological expertise due to each approach's adherence to fixed theories, techniques, and tools.
(4) Constrained dimensions for abstractions and decompositions in modelling approaches. For instance, MDA only emphasises on abstractions which move from platform-independent models to platform-specific ones. Decompositions, on the other hand, primarily focus on modules, objects, components, or services, neglecting the separation of different aspects.

Several of these challenges have been identified and discussed in existing literature [16,24,40,47]. Notably, these references represent early explorations in this field. The discussions in [16,24] offer broad conceptual discussions, without proposing concrete technical solutions. The work presented in [40] discusses the importance of these challenges, demonstrating with examples that decomposing a program into modules differs fundamentally from decomposing it based on its flowchart steps. The research in [47] introduces an approach to decomposition by presenting the concept of hyperslices, which capture concerns that cross cut various modules or methods across different classes. However, this approach is primarily presented within the context of programming languages, and it appears to lack a formal definition of what constitutes "concerns". Importantly, formal methods are not the primary focus of these studies.

In this paper, we utilise the modelling methodology of the Refinement Calculus of Object and Component Systems (rCOS) as a formal framework to further investigate these issues. Our objective is to showcase how a multi-dimensional architectural modelling technique can be devised to rectify and surpass the limitations found in mainstream approaches. It is our believe that to improve the limitations discussed above, MDA requires an IDPs to support consistent integration of models of different FMs.

2 Linking Formal Methods

As mentioned in Sect. 1, an FM consists of the techniques and tools developed underpinned the formal theory. These include

- Techniques and tools for constructing models and specifying the properties of software systems using the formal language of the theory.
- Theorem proving techniques and tools for proving the required properties and laws of refinements of software systems within the proof system.
- Techniques, algorithms, and tools for model checking and solving constraint satisfaction problems.

The research on FMs actually started with beginning of computer science, and it experienced a period of rapid development. There are now a large number FMs have been developed. There are a number of well-known classifications, including

(a) *By formalism*: model-based methods (e.g. those based on state transition systems), deductive methods which focus on the derivation of correct systems using formal logic, and process algebras (e.g. CCS, CSP, and the π-calculus).
(b) *By Phase of Application*: specification method, development (correct by construction) method, verification methods and validation methods.
(c) *By Verification Technique*: model checking, theorem proving, and type systems.
(d) *By level of formalisation*: lightweight formal methods and full/heavyweight formal methods.

FMs are also typically classified into *operational methods* or *denotational methods*, according to the semantics of their languages. For the purpose of this paper, we describe a classification of FMs according to the aspects of software they model, or *classification for separation of concerns*.

2.1 Formal Theories of Different Aspects

We primarily focus on discussing mechanisms of abstraction (vs. refinement) and decomposition (vs. composition) to achieve the purpose of separation of concerns. Consequently, we classify FMs based on the aspects their languages are designed to express. These aspects are determined by the viewpoints from which developers observe the system. They can be orthogonal but are often interrelated. Different aspects are typically modelled using different formal theories, and there are various theories for modelling the same aspect. A complete model of a system is the integration of models from different aspects.

Typically, an FM's language is defined with a specific set of symbols known as the *signature*, which is often finite and nonempty. This signature is commonly partitioned into disjoint subsets termed *sorts* (borrowed from algebras). Using this signature, expressions and statements are formed using symbols of *operations* following specific *grammatical rules*. For tool development in the FM, the grammatical correctness (also called *well-formedness*) of these expressions and statements needs to be machine-checkable. As a result, the language's design adheres to automata theory principles, making the language a set of symbol strings produced by a certain type of automaton.

The signature and grammatically rules of an FM's language are designed based one the aspect to be specified by the language. The signature and grammatical rules represents what relevant to the aspect the modeller need to observe in the system behaviour. We present the important aspects and give examples of their modelling theories below.

Data Types. All software systems require to process data of different types. Their representation, operations and implementations can be separated from controls flow and data flows of programs. Abstract data types (ADTs) are a formal theory for data types, in which each ADT is modelled by an abstract (or formal) algebra. In object-oriented programming paradigms, an ADT is implemented as class and a general class is also regarded as an ADT. In rCOS, the theory of semantics and refinement of class declarations (also represented by UML class diagrams) serves the purpose of *theory of data model* [18,32].

Local Functionality. Theories like Hoare Logic [20] and Calculus of Predicate Transformers [12] are used to specify and verify sequential programs, possibly with nondeterminism. We also include in this class those theories, such as VDM [23], Z [45] and B [1], which are based these and have mechanisms for modularity. Object-Z [44], rCOS theory of semantics and refinement for OO programs [18] and Java Modeling Language (JML) [28] can be regarded as OO extensions, and the latest VDM also treats object-orientation. These theories do not explicitly deal with concurrency and synchronisation.

Communication and Concurrency. Process algebras, such as CSP [21,43] and CCS [38,39], are event-based and abstract local computation away and explicitly treat concurrency and synchronisation. Petri Net [42] is another popular even-based formalism for concurrency. The the theory of input-output automata is control state and event based theory for communication and synchronisation.

Communication and Concurrency with Data Functionality. There are formalisms which combine local functionality with concurrency and synchronisation, where local computations are represented as atomic actions and specified in away like that in Hoare-Logic. Among them, we have logic systems like Temporal Logic of Actions (TLA) [25, 26] and UNITY [9], Action Systems [4], and Event-B [2]. A formal theory of this kind are more power than the above event-based theories in that they can specify properties of shared states. The apparently disadvantage they do not explicitly describe concurrency and synchronisation activities.

Performance and Quantity Aspects. A formalism for dealing with performance, such as timing, spacial aspects and energy consumption, can be defined by extending the

signatures of an existing formal theory in the above list. In theory, security and privacy requirements can be treated without the need of fundamentally new formalism [35], neither does fault-tolerance [33].

The list of formal theories referenced earlier is by no means exhaustive. We believe that there is no single FM that is sufficient for addressing all issues across every aspect of a system's lifecycle. In particular, the development of modern, complex software systems requires a family of FMs to build models for different aspects of the artefacts, which can then be analysed and verified. Moreover, for 2 and 3 outlined in the paragraph of aims and objectives in Sect. 1, an integrated model-driven development environment should support the utilisation and transformation of models, proofs, verification algorithms, and verified theorems from various formal methods for the same aspect. Thus, from both educational and industrial adoption perspectives of Model-Driven Engineering (MDE), understanding the interrelationships between different FMs is crucial.

2.2 UTP for Linking Formal Theories

he Unifying Theories of Programming (UTP) [22], developed by Tony Hoare and He Jifeng, presents a unified approach to defining formal programming theories. A theory $\mathbf{T}$ of programming in a specific paradigm aims to characterise the behaviour of programs using a set of alphabetised predicates. In $\mathbf{T}$, a predicate contains free variables from a designated set known as the alphabet of the predicate. The set of predicates, also denoted as $\mathbf{T}$, is subject to constraints defined by a set of axioms known as *healthiness conditions*.

The theory also establishes a set of symbols representing operations on the set $\mathbf{T}$. These symbols, along with the alphabets, collectively form the *signature* of $\mathbf{T}$. The connection between different theories is established based on the theory of complete lattices and Galois connections

As an example, we define a relational theory $\mathbf{R}$ for imperative sequential programming. In this theory, observables are represented by a given set X of input variables, along with their decorated versions $X' = x' | x \in X$ as output variables. The set $\alpha = X \cup X'$ is referred to as the *input alphabet*, *output alphabet*, and *alphabet* of $\mathbf{R}$.

In theory $\mathbf{R}$, a program (or specification) is expressed as a first-order logic formula P, associated with a subset αP of α in such a way that P only mentions variables in αP, which is known as the alphabet of P. Thus, a relation is written in the form $(\alpha P, P)$, and αP is the union of the input and output alphabets of P, denoted as $in\alpha P \cup out\alpha P$. We always assume that $in\alpha P \subseteq X$ and $out\alpha P = in\alpha' P = \{x \mid x \in in\alpha P\}$.

It is easy to define the meaning of operations symbols in sequential programming

$$(\alpha, \mathbf{skip}) \stackrel{def}{=} \bigwedge_{x \in in\alpha} x' = x, \ \ (\alpha, x := e) \stackrel{def}{=} x' = e \wedge \bigwedge_{x \in in\alpha - \{x\}} x' = x$$

The sequential composition is defined as for given relations D_1 and D_2 and conditional choice $D_1 \lhd b \rhd D_2$, where b is a Boolean expression

$D_1; D_2 \overset{def}{=} \exists v_0.D_1[v_0/out\alpha D_1] \wedge D_2[v_0/in D_2]$, provided $out\alpha D_1 = in\alpha D_2$

$in\alpha(D_1; D_2) \overset{def}{=} in\alpha(D_1)$, $\quad out(D_1; D_2) \overset{def}{=} out\alpha(D_1)$

$D_1 \lhd b \rhd D_2 \overset{def}{=} b \wedge D_1 \vee \neg b \wedge D_2$ $\quad$ provided $\alpha b \subset in\alpha D_1 = in\alpha D_2$

$\alpha(D_1 \lhd b \rhd D_2) = \alpha(D_1) = \alpha(D_2)$

We say that a predicate D_2 is refinement of predicate D_1 in theory $\mathbf{R}$, denoted as $D_1 \sqsubseteq_{\mathbf{R}} D_2$, if the implication $D_2 \rightarrow D_1$ is valid, i.e. the *universal closure* $[D_2 \rightarrow D_1]$ of $D_2 \rightarrow D_1$ holds. The refinement relation is a partial order, and *true* and *false* are the bottom and top elements. The formulas of $\mathbf{R}$ then forms a complete lattice and thus, according to Tarski's fixed point theorem, the least fixed-point $\mu X.(D; X) \lhd b \rhd \mathbf{skip}$ exists and it is defined to be loop state $b * D$. If we want to have nondeterministic choice $D_1 \sqcap D_2$, it is defined to be the disjunction $D_2 \vee D_2$.

We can readily observe that neither $true; D = true$ nor $D; true = true$ holds for an arbitrary relation D in $\mathbf{R}$. However, in all practical programming paradigms, they should both hold for an arbitrary program D if we use *true* to define the chaotic program $\bot$. Therefore, these *healthiness conditions* are imposed as axioms of $\mathbf{R}$.

Theory $\mathbf{R}$ is for specification and verification of partial correctness reasoning as it is not concerned *termination* of programs. When termination becomes an aspect of concern, it is necessary to extend $\mathbf{R}$ to a theory, denoted by $\mathbf{D}$, for specification and reasoning about total correctness of programs. To this end, we introduce two fresh observables *ok* and ok' which are Boolean variables. $\mathbf{D}$ contains the specifications of the form $P \vdash Q$ called *designs*, where P and Q are predicates in $\mathbf{R}$. The meaning of $P \vdash Q$, is however defined by the formulation $(P \wedge ok) \rightarrow (Q \wedge ok')$. We still use input alphabet, output alphabet and alphabet of P and Q as those of the design $P \vdash Q$.

We can then refine the meaning of operations on programs, where we omit the alphabets for simplicity:

$$\mathbf{skip} \overset{def}{=} true \vdash x' = x \text{ for all } x \in \alpha$$
$$\bot \overset{def}{=} false \vdash true = \texttt{true}$$
$$\top \overset{def}{=} \neg ok$$
$$x := e \overset{def}{=} defined(e) \vdash x' = e \text{ and } \forall y \in (\alpha - \{x\}).y' = y$$
$$D_1 \lhd b \rhd D_2 \overset{def}{=} defined(b) \rightarrow (b \wedge D_1 \vee \neg b \wedge D_2)$$
$$b * D \overset{def}{=} \mu X.(D; X) \lhd b \rhd \mathbf{skip}$$

Latter we allow to write $\mathbf{true} \vdash Q$ as $\vdash Q$. The least fixed-point of loop statement is defined for the refinement relation on $\mathbf{D}$ which is still implication, $\bot$ and $\top$ defined above.

It is easy to prove that both $true; D = true$ and $D; true = true$ hold in $\mathbf{D}$. Likewise, the left zero law $(\bot; D) = D$ and the left unit law $(\mathbf{skip}; D) = D$ also hold in $\mathbf{D}$. However, neither the right zero law $(D; \bot) = \bot$ nor the right unit law $(D; \mathbf{skip}) = D$ holds in $\mathbf{D}$. In UTP, healthiness conditions were imposed to ensure that they hold.

In either $\mathbf{R}$ or $\mathbf{D}$, we can encode Hoare logic and Dijkstra's calculus predicate transformer. Given a predicate D in $\mathbf{R}$ or $\mathbf{D}$, and two state properties p and q, we define the Hoare triple as $\{p\}P\{q\} \stackrel{def}{=} P \rightarrow (p \rightarrow q')$. where p' is the predicate obtained from p by replacing all free variables in p with their dashed version. Then, the axioms and inference rule hold in $\mathbf{D}$ and $\mathbf{R}$. Given a predicate D of $\mathbf{R}$ or $\mathbf{D}$ and a state property r, we define the *weakest precondition* of D for the postcondition r as $wp(D, r) \stackrel{def}{=} \neg(D; \neg r)$. Then, the rules in the wp calculus are valid in $\mathbf{D}$ and $\mathbf{R}$.

The above definitions show that the theories of Hoare logic and calculus of predicate transformers can be mapped into the theory $\mathbf{D}$ and $\mathbf{R}$ and used consistently. Furthermore, we can treat $\mathbf{D}$ as *sub-theory* of $\mathbf{R}$ with alphabet $X \cup X' \cup \{ok, ok'\}$ by the translation mapping $\mathcal{T} : \mathbf{D} \rightarrow \mathbf{R}$ such that $\mathcal{T}(P \vdash Q) = (P \rightarrow Q)$ for each design $(P \vdash Q)$ in $\mathcal{D}$. It is important to note the differences and relations between the languages, proof systems and models (i.e. semantics) of $\mathbf{R}$, $\mathcal{D}$, Hoare logic and the calculus of predicate transformers.

Another important theory for unifying or linking formalisms is the theory of *institution* [15] by Goguen and Burstall. The theory is based category theory and using a group of related mappings to define *meaning serving* translations between formal languages, and their associated mappings between specifications, theorems, proofs, and models. It is an interesting research problem to establish the formal relation of UTP and the theory of institution. It is important to say that the the purpose of unification is actually for consistent use to support separation of concerns.

3 rCOS Theory for Component-Based Architecture

The formal theory of rCOS is an extension to the design calculus $\mathbf{D}$ introduced in the previous section for modelling and refinement of OO and component-based software systems [17,18,31]. It has the advantage of supporting modelling and refinement of continuously evolving architectures that accommodate open design [36].Here, open design means that the architecture allows the use of subsystems that are designed and operated by different organisations.

3.1 rCOS Theory of Semantics and Refinement of OO Programming

The theory defined an abstract OO programming language (OPL). A normal form of an OO program P is *Classes* • *Main*, where

- *Classes* is a finite (possibly empty) class declarations $C_1; \ldots; C_n$.
- A class declaration C has a name N; a list of typed attributes and a list of methods $m_1(in_1; out_1)\{c_1\}; \ldots; m_k(in_k; out_k)\{c_k\}$, where a method $m(in, out)\{c\}$ has a name m, a list *in* of input parameters and *out* of output parameters with their types, a body c which is a command written in OPL.
- *Main* is the main class which it has a list of attributes and its main method $main\{c\}$.
- The syntax of the OPL is defined as:

$$c ::= P \vdash Q \mid \mathbf{skip} \mid \mathbf{chaos} \mid \mathbf{var}\ x := e \mid \mathbf{end}\ x \mid le := e \mid C.New(x, e) \mid (c; c) \mid c \lhd b \rhd c \mid le.m(e)$$

The type of an attribute or a program vairaible can be either a primitive type or or a class, and the order of the methods in a class is not essential.

Notably, we allow to use a design as a command, and *le* is called an assignable expression defined by $le ::= x \mid a \mid le.a \mid slef$, i.e. it is either a simply variable name x, an attribute name, a trace of attributes names or *self*. An expression e, which can appear on the right hand side of assignment symbol $:=$, is defined as $e ::= x \mid slef \mid null \mid e.a \mid f(e)$. Here $f(e)$ is an expression constructed using operations of primitive types. The full language also include in encapsulations of class attributes, and commands of type casting and type testing, but we omit them here as they are not particularly discussed.

The rCOS theory on OO programming is presented in the paper [18]. There, an object of class is formally defined and the state space of the a program is all the objects of the *Main* class. For intuitive explanation, we use graphs as follows.

- An object of a class C is represented as a directed graph with a root that represents the object. From the root, each edge points to a node, which can be an object of the class of an attribute or a node that represents a value in the type of an attribute with a primitive type. The edges are labelled with the names of attributes. Nodes that represent values of primitive types or null objects are considered leaves. Non-leaf nodes represent objects and are recursively defined as sub-graphs.
- A state of a program is an object of the *Main* class.
- An execution of the program is to change the initialised object to a final object, each step of the execution may create a new object, modify the values of an attribute of primitive type, replace an edge with another one (swings an edge, say by executing $le.a := self.b$).

The formal definition of an object is by finite paths by the graph, thus having ruled out infinitely looping in the graph. Therefore, a design $P \vdash Q$ in the OO theory specifies a relation on state graphs. The refinement relation is also defined as logic implication. In the paper [50], an operational semantics for the OO language is actually defined using graphs, and a set sound and relative complete refinement rules are proven to actually form a refinement calculus of OO programs.

It is easy to realise that the class declarations of an OPL program can be represented as a UML class diagram. On the other hand, a UML class diagram also represents a list of OPL class declarations plus constraints, such as those specified by association multiplicities and textual comments. There is no difficulty in extending the OPL syntax to allow the specification of these constraints on classes and attributes, which can be written in the *Object Constraint Language* (OCL) or formal predicate logic.

We refer to UML diagrams and OPL class declaration sections with object constraints as *class models*. Therefore, a class model defines a set of UML object diagrams, with each object in an object diagram corresponding to an object graph. This enables us to utilise the rCOS object-oriented semantic theory as a formal semantics for UML class models, as well we the type system of rCOS OPL programs.

3.2 Model of Interface Contracts of Components

According to Szyperski [46], 'A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software com-

ponent can be deployed independently and is subject to composition by third parties.' In the semantic theory, we specifically focus on 'contractually specified interfaces and explicit context dependencies only' and the idea of being 'subject to composition by third parties.' Therefore, *interfaces* are considered first-class syntactic model elements, and *interface contracts* represent the first-class semantic modelling concept.

Interfaces. An *interface* $\mathcal{I} = \langle \mathcal{T}, \mathcal{V}, \mathcal{M} \rangle$ consists of a list $\mathcal{T}$ of type declarations, a list $\mathcal{V}$ of state variables (or attributes) with types defined in $\mathcal{T}$, and a list $\mathcal{M}$ of method declarations with parameters having types from $\mathcal{T}$. $\mathcal{T}$ can include class declarations, and both $\mathcal{V}$ and $\mathcal{M}$ follow the syntax of the rCOS OPL language. The state variables in $\mathcal{V}$ are encapsulated within the component and can only be accessed from outside the component through specified methods. The implementation details of the interface methods are hidden inside the component, but their functionalities must be specified and published to enable proper use.

Example 1 (Interface of Memory). A memory provides services for writing values to it and reading values from it from it with an interface

$$\mathit{M.I} = \langle \{int, array[int]\} \{array[int]\ m\}, \{w(\textit{int}\ v), r(; \textit{int}\ u)\} \rangle$$

where m is a variable for the content of memory. Later we simply omit the types in example when they are clear, and use a program template to write interfaces.

Reactive Designs. Components in component-based systems are distributedly deployed and run concurrently, often requiring to be reactive. To specify the reactive behaviour of components, we introduce the concept of *reactive designs*. Similar to the approach in Sect. 3, where the relational theory $\mathbf{R}$ is extended to the design theory $\mathbf{D}$, we extend the theory $\mathbf{D}$ to the theory $\mathbf{C}$ by introducing two new Boolean observables, *wait*, *wait*$'$, and defining that a design D is considered *reactive* if it is a fixed point of the lifting mapping $\mathcal{W}$, i.e., $\mathcal{D}(D) = D$, where $\mathcal{W}$ is defined as follows:

$$\mathcal{W}(D) \stackrel{def}{=} (\textit{true} \vdash (\textit{wait}' \wedge (in\alpha' = in\alpha) \wedge (ok' = ok)) \lhd \textit{wait} \rhd D$$

Here $in\alpha$ is the input alphabet of D without *ok* and *wait* being in it. Hence, when *wait* is true the execution of D is blocked.

To specify both the functionality and synchronisation conditions of interface methods, we introduce the concept of 'guarded design' in the form $g\&D$, where g is a Boolean expression based on program variables (namely, *ok* and *wait*), and D is a reactive design in $\mathbf{C}$. The meaning of $g\&D$ is defined as:

$$D \lhd g \rhd (\textbf{true} \vdash (in\alpha' = in\alpha) \wedge (ok' = ok))$$

It can be easily proven that $\mathcal{W}$ is monotonic with respect to implication, thus the set $\mathbf{C}$ of reactive designs. forms a complete lattice. This, along with the following two theorems, is necessary for $\mathbf{C}$ to be used as a theory of reactive programming.

Theorem 1. *For any designs D, D_1 and D_2 of* $\mathbf{D}$*, we have*

(1) $\mathcal{W}$ *is idempotent* $\mathcal{W}^2(D) = \mathcal{W}(D)$*;*
(2) $\mathcal{W}$ *is nearly closed for sequencing:* $(\mathcal{W}(D_1); \mathcal{W}(D-2)) = \mathcal{W}(D_1, \mathcal{W}(D_2))$*;*
(3) $\mathcal{W}$ *is closed for non-deterministic choice:* $(\mathcal{W}(D_1) \vee \mathcal{W}(D_2)) = \mathcal{W}(D_1 \vee D_2)$*;*
(4) $\mathcal{W}$ *is closed for conditional choice:* $(\mathcal{W}(D_1) \lhd b \rhd \mathcal{W}(D_2)) = \mathcal{W}(D_1 \lhd b \rhd D_2)$.

Here, "=" denotes logical equivalence, and the following properties hold for guarded designs, which ensure that they can be used as commands in reactive programming.

Theorem 2. *For any react designs* D*,* D_1 *and* D_2 *of* $\mathbf{D}$*, we have*

(1) $g\&D$ *is reactive;*
(2) $(g\&D_1 \vee g\&D_2) = g\&(D_1 \vee D_2)$
(3) $(g_1\&D_1 \lhd b \rhd g_2\&D_2) = (g_1 \lhd bg2)\&(D1 \lhd B \rhd D_2)$*;*
(4) $(g_1\&D_1; g_2\&D_2) = g_1\&(D_1; g_2\&D_2)$.

It is important to note that from Theorem 1(1) a(1), D in a guarded design $g\&D$ can be a design in $\mathbf{D}$ too, and the top level interface specification only uses guarded designs of this kind. We need to define programs in a programming language, such as one in Subsect. 2.2 and the OPL in Subsect. 3.1, as reactive designs. The way to do this is to define the primitive commands directly as reactive design and the apply the above two theorems for composite composite commands. We take the following primitive commands as examples.

$$\mathbf{skip} \stackrel{def}{=} \mathcal{W}(\mathit{true} \vdash \neg \mathit{wait}' \wedge in\alpha' = in\alpha), \quad \mathbf{choas} \stackrel{def}{=} \mathcal{W}(\mathit{false} \vdash \mathit{true})$$
$$\mathbf{stop} \stackrel{def}{=} \mathcal{W}(\mathit{true} \vdash \mathit{wait}' \wedge in\alpha' = in\alpha)$$
$$x := e \stackrel{def}{=} \mathcal{W}(\mathit{true} \vdash (x' = e) \wedge \neg \mathit{wait}' \wedge (in\alpha - \{x\})' = (in\alpha - \{x\})$$

We will write $\mathit{true}\&D$ as D.

Contracts of Interfaces. An *interface* of a component is a point through which the component interact with its environment, that is to provide services or require services. Therefore, a component can have a number of interfaces, some of them are *provided* and some *required services*, but their semantics is defined in the same way, as *contracts*.

Definition 1 (Contract of Interface). *Given an interface* $\mathcal{I} = \langle \mathcal{T}, \mathcal{V}, \mathcal{M} \rangle$*, a* **contract** $\mathcal{C}$ *for* $\mathcal{I}$ *is a pair* (θ, Φ)*, where*

- θ *is a predicate with free variables in* $\mathcal{V}$*, called the initial condition;*
- *and* θ *is a mapping from* $\mathcal{M}$ *to* $\mathbf{C}$ *such that each* $m(in; out)$ *is assigned with a guarded design* $g\&D$ *over input variables* $\mathcal{V} \cup in$ *and output variables* $\mathcal{V}' \cup out'$*.*

A contract $\mathcal{C}$ *is denoted by a tripe* $\mathcal{C} = \langle \mathcal{I}, \theta, \Phi \rangle$.

Example 2 (Contracts for memory interface). We give two contracts for *M.I*. The first $\mathcal{C}_1 = \langle \mathit{M.I}, \mathit{true}, \Phi_1 \rangle$, where Φ_1 is defined by giving the designs as bodies of the corresponding methods as follows

$$w(\mathit{int}\ v)\{\neg \mathit{isFull}(m) \vdash m' = \mathit{append}(a, m)\};$$
$$r(; \mathit{int}\ u)\{\neg \mathit{isEmpty}(m) \vdash (u' = \mathit{head}(m) \wedge m' = \mathit{tail}(m))\}$$

Here, operations on data *isEmpty*, *isFull*, *append*(), *head*(), and *tail*() are supposed to have been defined in the definitions of the types.

The second contracts $\mathcal{C}_2 = \langle \mathit{M.I}, \mathit{isEmpty} = false, \Phi_2 \rangle$ imposes controls to the invocations to the interface methods with gourds, where

$$w(\mathit{int}\ v)\{\neg \mathit{isFull}(m) \& (\neg \mathit{isFull}(m) \vdash m' = \mathit{append}(a, m))\};$$
$$r(; \mathit{int}\ u)\{\neg \mathit{isEmpty} \& (\neg \mathit{isEmpty}(m) \vdash u' = \mathit{head}(m) \wedge m' = \mathit{tail}(m))\}$$

Thus, the first invocation must be write, and then the user cal write into the memory as long it is not full, and can read from it as long as it is not not empty.

Dynamic Behaviours of Contracts. In an operational view, a contract defines the behaviour of an *labelled state transition system* (LSTS) in such a way that its states include elements from $\mathcal{V} \cup \mathit{ok}, \mathit{wait}$, where the labels of transitions represent invocations $?m(v)$ and returns $!m(u)$ with actual input parameters v and actual output parameters u. An execution of the LSTS diverges once it enters a state in which *ok* is *false*, and it deadlocks when it reaches a state where *wait* is *true*. However, when we define the 'refinement relation' between contracts, a denotational model becomes more convenient. The dynamic behaviour of a contract $\mathcal{C} = \langle \mathcal{I}, \theta, \Phi \rangle$ is defined in terms of its *divergences*, *failures*, and *traces*, which captures the key characteristics of concurrent programs.

Definition 2 (Divergences). *For a contract $\mathcal{C} = \langle \mathcal{I}, \theta, \Phi \rangle$, the set Div($\mathcal{C}$) of* **divergences** *of $\mathcal{C}$ consists of sequences of invocations and returns of methods from the interface $\langle ?m_1(v_1)!m_1(u_1) \ldots ?m_k(v_k)!m_k(u_k) \rangle$ whose executions end in a divergent state. In other words, the execution of $\theta; m_1(v_1; u_1); \ldots; m_j(v_j; u_j)$ as a prefix of the sequence, starting from an initial state, results in ok$'$ being false, where v_i and u_i are the actual input and output parameters of the invocation $m_i()$.*

Definition 3 (Failures). *Given a contract $\mathcal{C}$, a* **failure** *of $\mathcal{C}$ is pair (tr, M), where tr is a finite trace of method invocations and returns $\langle ?m_1(v_1)!m_1(u_1) \ldots \rangle$ of the interface, and M a set of method invocations, such that one of the following conditions holds*

(1) tr is the empty sequence, and M is the set of invocations $?m(v)$ with their guards being false in the initial states;
(2) tr is a trace $\langle ?m_1(v_1)!m_1(u_1) \ldots ?m_k(v_k)!m_k(u_k) \rangle$ and M consists of the invocations $?m(v)$ that after the executions of the invocations $m_1(x_1, y_1) \ldots m_k(x_k, y_k)$ from an initial state the guard of $m(v)$ is false;
(3) tr is a trace $\langle ?m_1(x_1)!m_1(y_1) \ldots ?m_k(x_k) \rangle$ and the execution of the invocation $m_k(v)$ is yet to deliver an output, and M contains all invocations;
(4) tr is a trace $\langle ?m_1(x_1)!m_1(y_1) \ldots ?m_k(x_k) \rangle$ and the execution of the invocation $m_k(v)$ has entered a wait state, and M contains all invocations; or
(5) tr is a divergence in $\mathcal{D}_\mathcal{C}$, and M contains all invocations (all invocations can be refused after the execution diverges).

We use Fail($\mathcal{C}$) to represent set of failures of contract $\mathcal{C}$.

It is worth noting that some guards are more complex than others, depending on whether they involve only control states, both control and data states, or a combination of input parameters, control, and data states. For example, consider to change the guard of $r()$ in Example 2 to $\neg isEmpty \wedge Even(v) \wedge count(m) \leq 4$. In general, changing preconditions of designs affects the divergence set and failures. Similarly, altering guards influences failures and divergences, potentially leading to invocations that violate the design's precondition. These properties are characterised by the concept of contract refinement and its associated theorem.

Definition 4 (Contract refinement). *A contract $\mathcal{C}_1$ is* **refined** *by a contract $\mathcal{C}_2$, denoted as $\mathcal{C}_1 \sqsubseteq \mathcal{C}_2$, if they have same interface, and*

(1) $\mathcal{C}_2$ is not more likely to diverge, i.e. $Div(\mathcal{C}_2) \subseteq Div(\mathcal{C}_1)$; and
(2) $\mathcal{C}_2$ is not more likely to block the environment, i.e. $Fail(\mathcal{C}_2) \subseteq Fail(\mathcal{C}_1)$.

The effective way to establish contract refinement is by *upward simulation* and *downward simulation* [17].

The above discussion shows how we can extract models of failures and divergences from the models of contracts defined by reactive designs. This establishes a connection between the theory of input and automata, action systems, and process calculi. It is often simpler to specify communication requirements in terms of communication traces or protocols. We first define the set of traces for a contract $\mathcal{C}$ from its failures

$$Trace(\mathcal{C}) \stackrel{def}{=} \{tr | \text{there exists an } M \text{ such that } (tr, M) \in Fail(\mathcal{C})\}$$

Then, a protocol $\mathcal{P}$ is a subset of $Prot(\mathcal{C}) \stackrel{def}{=} \{tr \downarrow_? \mid tr \in Trace(\mathcal{C})\}$.

Definition 5 (Consistency of Protocol with Contract). *A protocol $\mathcal{T}$ is* **consistent** *with $(\mathcal{D}_\mathcal{C}, \mathcal{F}_\mathcal{C})$ (thus with contract $\mathcal{C}$), if the execution of any prefix of any invocation sequence sq in $\mathcal{T}$ does not enter a state in which all invocations to the provided services are refused, i.e. for any $sq \in \mathcal{T}$ and any $(tr, M) \in \mathcal{F}_\mathcal{C}$ such that $sq = tr \downarrow_?$, $M \neq \{m(v) \mid m() \in O\}$ if $tr \downarrow_?$.*

We have the following theorem for the consistency between protocols and contracts.

Theorem 3. *Given a contract $\mathcal{C}$ and its protocols $\mathcal{T}_1$ and $\mathcal{T}_2$, we have*

(1) If $\mathcal{T}_1$ is consistent with $\mathcal{C}$ and $\mathcal{T}_2 \subseteq \mathcal{T}_1$, $\mathcal{T}_2$ is consistent with $\mathcal{C}$.
(2) If both $\mathcal{T}_1$ and $\mathcal{T}_2$ are consistent with $\mathcal{C}$, so is $\mathcal{T}_1 \cup \mathcal{T}_2$.
(3) If $\mathcal{C}_1 = (I, \theta_1, \Phi_1)$ is another contract of interface I and $\theta \sqsubseteq \theta_1$ and $\Phi(m()) \sqsubseteq \Phi_1(m())$ for any operation $m()$ of the interface, $\mathcal{T}_1$ is consistent with $\mathcal{C}_1$ if it is consistent with $\mathcal{C}$.

The concept and theorems discussed above regarding consistency are essential for the correct use of components in different interaction environments. It also enables the separation of designing functional aspects from tailoring communication protocols. In an incremental development process, the specification of a component's interface can first be provided as a pair $(\mathcal{D}, \mathcal{P})$ of designs in OPL (not guarded designs) along with a protocol $\mathcal{P}$. This initial specification can then be further developed into a fully specified contract as required. Another advantage of separating the specification of functionality using designs and interaction protocols is that, as shown in Example 2, it allows for different interaction protocols for the same functionality specification, and vice versa.

4 rCOS Support to Separation of Concerns in MDE

We have developed an understanding of the unification of data, local functionality, and dynamic behaviour. However, the purpose of this unification is to ensure consistent integration of models of these different aspects. This is particularly crucial for Model-Driven Engineering (MDE) and integrated development platforms. Now, let us discuss how this is put into practice in MDE. In rCOS, we propose to support a use-case-driven incremental development process, also known as the Rational Unified Process (RUP). However, we place a strong emphasis on component-based architectural models and refinement.

4.1 Use Case Driven Requirements Model Building

A requirements model consists of a set of interrelated use cases identified from the application domain. Each use case is modelled as an interface contract of a component in the following steps:

1. The operations provided by the component's interface corresponding to a use case consist of the interactions with the actors.
2. The classes are represented in conceptual class diagrams that the use case involves, and the state variables are names for objects that the use case needs to know (i.e., to record, check, modify, and communicate).
3. The interaction protocol of the use case represents the possible events of the interactions between the actors and the system to carry out the use case, and they are modelled by sequence diagrams.
4. The dynamic behaviour of a use case is modelled by a state diagram of the use case.
5. 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.
6. The requirements architecture is modelled by UML component-based diagrams that reflect the relations among use cases in the use case diagram.

The models of above aspects of a component for a use case consisting of interactions, dynamic behaviour, types and functionality. More systematic presentation of the method can be found in [11, 13], but with less formal support.

4.2 Component Development Process

A OO design process for components consists of the following modelling steps as shown in Fig. 1:

1. It takes each use-case component and designs each of its provided operations according to its pre- and post-conditions using the object-oriented refinement rules, with a focus on the four patterns of GRASP, in particular [11,27].
2. This decomposes the functionality of each use case operation into internal object interactions and computations, refining the use case sequence diagram into an object sequence diagram of the use case [11].

3. During the decomposition of the functionality of use-case operations into internal object interactions and computations, the requirements class model is refined into a design class model by adding methods and visibilities in classes based on responsibility assignments and method invocation directions [11].
4. Select some of the objects in an object sequence diagram and consider them as candidate component controllers if they satisfy six given invariant properties through automatic checks. Then, transform the design sequence diagram into a component-sequence diagram [30].
5. Generate a component diagram for each use case from the component sequence diagram obtained in the previous step, automatically decomposing the use-case component in the requirements model into a composition of sub-components. This leads to the decomposition of the entire component-based architecture model at the requirements level into a component-based design architecture model [30].
6. The coding from the design architecture model is not particularly difficult and can largely be automated [37,49].

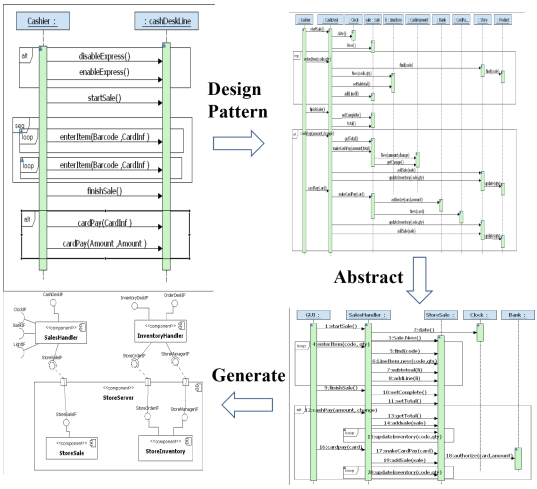

Fig. 1. Transformations from requirements to design of a component

4.3 System Development

For a given application domain, we assume a repository of implemented components for 'plenty' of use cases, their contract specifications, information on context dependencies, and (possibly) their sub-components[1].

Roughly speaking, the system development begins with the development of a requirements model in the form of use case contracts. The use case contracts are then

[1] We do not know such as an existing repository.

refined and/or decomposed into compositions of components to create a model of the system architecture. Subsequently, we search for candidate components in the repository that match a component in the architecture and check if their contracts are refinements of those in the architecture. The verification of functional requirements and synchronisation requirements can be conducted separately, and they can be refined independently by adding connectors and coordinators, respectively.

It is possible that for some contracts of components in the architecture, there are no appropriate components that can be easily adapted for implementation. In such cases, we have to develop them using the component development method discussed in the previous subsection. The main features of component and system development in rCOS are shown in Fig. 2.

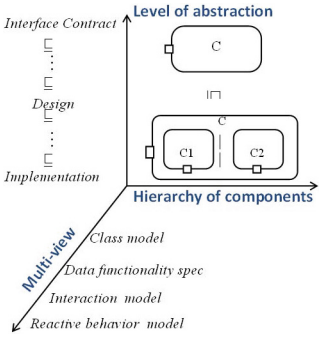

Fig. 2. Features of the rCOS modelling and development

To gain a deeper understanding of the processes described above, we recommend referring to the paper in [10], which reports the application of rCOS to the CoCoMe common case study [19]. Please note that domain knowledge is essential for providing the requirements model in terms of use cases, for designing the architecture, and for mapping them to components in the repository. The primary challenge in formalizing the mapping and developing tool support lies in the different naming schemes used in the requirements models, the architecture design, and the representations of the models of the components in the repository. In our opinion, significant effort is required in this area.

5 Conclusions

Abstractions and decompositions with respect different aspects are, which are essential for reuse and modification. We are skeptical that there exists a tool capable of automatic modelling and decomposition that can seamlessly achieve this separation of concerns. It predominantly requires human intelligence and effort. Nonetheless, a formal modelling method should offer multi-dimensional decomposition beyond just component-based methods, and multi-dimensional abstraction surpassing mere focus on a single

aspect of the system. Additionally, when it comes to model refinement, there should be a broader spectrum of transformation options than just those transitioning from platform-independent models (PIMs) to platform-specific models (PSMs).

We have highlighted the significance and challenges associated with the development and application of formal methods that systematically aid in addressing abstraction and decomposition. We have introduced the rCOS framework to delineate the dimensions of models for different aspects and to discuss issues related to their decomposition, refinement, and consistent composition. While our discussion primarily centres on the formalisms and their capabilities, we acknowledge that there is limited coverage of tool implementation and support. However, for rCOS tools, we refer to the work in [29,34,49], though they are still in the form of proof of concepts.

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. Andersen, E.P., Reenskaug, T.: System design by composing structures of interacting objects. In: Madsen, O.L. (ed.) ECOOP 1992. LNCS, vol. 615, pp. 133–152. Springer, Heidelberg (1992). https://doi.org/10.1007/BFb0053034
4. 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
5. Bell, M.: Service-Oriented Modeling: Service Analysis, Design, and Architecture. Wiley, Hoboken (2008)
6. Booch, G.: Object-Oriented Analysis and Design with Applications. Addison-Wesley, Boston (1994)
7. Brooks, F.P.: No silver bullet: essence and accidents of software engineering. IEEE Comput. **20**(4), 10–19 (1987)
8. Brooks, F.P.: The mythical man-month: after 20 years. IEEE Softw. **12**(5), 57–60 (1995)
9. Chandy, K.M., Misra, J.: Parallel Program Design: A Foundation. Addison-Wesley, Reading (1988)
10. Chen, Z., et al.: Modelling with relational calculus of object and component systems - rCOS. In: Rausch, A., Reussner, R., Mirandola, R., Plášil, F. (eds.) The Common Component Modeling Example. LNCS, vol. 5153, pp. 116–145. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-85289-6_6
11. 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)
12. Dijkstra, E.W.: A Discipline of Programming. Prentice-Hall, Englewood Cliffs (1976)
13. Dong, J.S., Woodcock, J. (eds.): Formal Methods and Software Engineering, 5th International Conference on Formal Engineering Methods, ICFEM 2003, Singapore, November 5–7, 2003, Proceedings. Lecture Notes in Computer Science, vol. 2885. Springer, Heidelberg (2003). https://doi.org/10.1007/b94115
14. Gauthier, R., Pont, S.: Designing Systems Programs. Prentice-Hall, Englewood Cliffs (1970)
15. Goguen, A.J., Burstall, R.M.: Institutions: abstract model theory for specification and programming. J. ACM **39**(1), 95–146 (1992)
16. Haan, J.D.: 8 reasons why model-driven approaches (will) fail, infoQ. https://www.infoq.com/articles/8-reasons-why-MDE-fails/

17. He, J., Li, X., Liu, Z.: A theory of reactive components. Electr. Notes Theor. Comput. Sci. **160**, 173–195 (2006)
18. He, J., Liu, Z., Li, X.: rCOS: a refinement calculus of object systems. Theoret. Comput. Sci. **365**(1–2), 109–142 (2006)
19. Herold, S., et al.: The common component modeling example. In: Rausch, A., Reussner, R., Mirandola, R., Plášil, F. (eds.) The Common Component Modeling Example. Lecture Notes in Computer Science, chap. 1, , vol. 5153, pp. 16–53. Springer, Heidelberg (2008)
20. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM **12**(10), 576–580 (1969)
21. Hoare, C.A.R.: Communicating sequential processes. Commun. ACM **21**(8), 666–677 (1978)
22. Hoare, C.A.R., He, J.: Unifying Theories of Programming. Prentice-Hall, Upper Saddle River (1998)
23. Jones, C.B.: Systematic Software Development using VDM. Prentice Hall, Upper Saddle River (1990)
24. Kent, S.: Model driven engineering. In: Butler, M., Petre, L., Sere, K. (eds.) IFM 2002. LNCS, vol. 2335, pp. 286–298. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-47884-1_16
25. Lamport, L.: The temporal logic of actions. ACM Trans. Program. Lang. Syst. **16**(3), 872–923 (1994)
26. Lamport, L.: Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers. Addison-Wesley, Boston (2002)
27. 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)
28. Leavens, G.T., Baker, A.L.: Enhancing the pre- and postcondition technique for more expressive specifications. In: Wing, J.M., Woodcock, J., Davies, J. (eds.) FM 1999. LNCS, vol. 1709, pp. 1087–1106. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-48118-4_8
29. Li, D., Li, X., Liu, J., Liu, Z.: Validation of requirements models by automatic prototyping. J. Innov. Syst. Softw. Eng. **4**(3), 241–248 (2008)
30. Li, D., Li, X., Liu, Z., Stolz, V.: Interactive transformations from object-oriented models to component-based models. In: Arbab, F., Ölveczky, P.C. (eds.) FACS 2011. LNCS, vol. 7253, pp. 97–114. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-35743-5_7
31. Liu, Z.: Linking formal methods in software development - a reflection on the development of rCOS. In: Bowen, J.P., Li, Q., Xu, Q. (eds.) Theories of Programming and Formal Methods. LNCS, vol. 14080, pp. 52–84. Springer, Cham (2023). https://doi.org/10.1007/978-3-031-40436-8_3
32. 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
33. Liu, Z., Joseph, M.: Specification and verification of fault-tolerance, timing, and scheduling. ACM Trans. Program. Lang. Syst. **21**(1), 46–89 (1999)
34. 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)
35. 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
36. Liu, Z., Wang, J.: Human-cyber-physical systems: concepts, challenges, and research opportunities. Frontiers Inf. Technol. Electron. Eng. **21**(11), 1535–1553 (2020)

37. Long, Q., Liu, Z., Li, X., He, J.: Consistent code generation from UML models. In: 16th Australian Software Engineering Conference (ASWEC 2005), 31 March–1 April 2005, Brisbane, Australia, pp. 23–30. IEEE Computer Society (2005). https://doi.org/10.1109/ASWEC.2005.17
38. Milner, R.: Communication and Concurrency. Prentice-Hall Inc., Upper Saddle River (1989)
39. Milner, R.: A Calculus of Communicating Systems. Springer, Heidelberg (1980). https://doi.org/10.1007/3-540-10235-3
40. Parnas, D.L.: On the criteria to be used in decomposing systems into modules. Commun. ACM **15**(12), 1053–1058 (1972)
41. Parnas, D.L.: A technique for software module specification with examples. Commun. ACM **15**, 330–336 (1972)
42. Petri, C.A., Reisig, W.: Petri net. Scholarpedia **3**(4) (2008)
43. Roscoe, A.W.: Theory and Practice of Concurrency. Prentice-Hall, Upper Saddle River (1997)
44. Smith, G.: The Object-Z Specification Language. Springer, Heidelberg (2000). https://doi.org/10.1007/978-1-4615-5265-9
45. Spivey, J.M.: The Z Notation, A Reference Manual. International Series in Computer Science, 2nd edn. Prentice Hall, Upper Saddle River (1992)
46. Szyperski, C.: Component Software: Beyond Object-Oriented Programming, 2nd edn. Addison-Wesley Longman Publishing Co., Inc., Boston (2002)
47. Tarr, P., Ossher, H., Harrison, W., Sutton, S.M.: N degrees of separation: multi-dimensional separation of concerns. In: Proceedings of the 1999 International Conference on Software Engineering, pp. 107–119. IEEE (1999)
48. Wang, J., Zhan, N., Feng, X., Feng, Liu, Z.: Overview of formal methods (in Chinese). Ruan Jian Xue Bao/J. Softw. **30**(1), 33–61 (2019)
49. Yang, Y., Li, X., Ke, W., Liu, Z.: Automated prototype generation from formal requirements model. IEEE Trans. Reliab. **69**(2), 632–656 (2020)
50. Zhao, L., Liu, X., Liu, Z., Qiu, Z.: Graph transformations for object-oriented refinement. Formal Aspects Comput. **21**(1–2), 103–131 (2009)