Formal Aspects of Component Software [⋆]
– An Overview on Concepts and Relations of Different Theories

Zhiming Liu, Jiadong Teng and Bo Liu

School of Computer and Information Science
Southwest University, Chongqing, China

Abstract. The *International Symposium on Formal Aspects of Component Software* (FACS) was inaugurated two decades ago in response to the major software development paradigm shift from *structured development* and *object-oriented development* to *component-based software development* (CBSD) and *service-oriented architecture* (SOA). FACS is dedicated to fostering a deeper understanding of the distinctive aspects, promoting research, education, technological advancement, and the practical application of CBSD technology. On the 20th anniversary of FACS, it is appropriate to briefly recall its background and history, thereby highlighting its contributions to the community. Taking this opportunity, we focus on the discussion to elucidate the important aspects of component software that require to be and have been considered in formal theories. Leveraging the refinement of component and object-oriented systems (rCOS) as a framework, we provide an overview of these formal theories and discuss their relationships. We intend to express a vision that different theories and methods are required for different aspects in a CBSD process, and also different formal theories are required even for a particular aspect. However, ensuring their consistent application remains a major challenge and this is a main barrier to the effective industry adoption of CBSD. Furthermore, we delineate emerging challenges and prospects associated with integrating formal methods for modelling and and design human-cyber-physical systems (HCPS) — hybrid integration encompassing cyber systems, physical systems, and the mixed human and machine intelligence.

Keywords: Component-Based Software, Formal Aspects, Human-Cyber-Physical Systems, Linking Formal Methods, rCOS

1 Introduction

In general, we understand *software development* or *software engineering* as being about transforming processes carrying out out by instances of domain concepts into their programming models through activities of software *requirements analysis*, *design*, *verification*, *implementation*, *deployment* and *maintenance* (now possibly better to say *evolution*). It was at the turn of the last century and this century when a major software development paradigm shift from *structured development* and *object-oriented development*

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

to *component-based development* (CBD) and *service-oriented architecture* (SOA) was happening. By then, the academic community had gained quite a mature understanding and plenty pratice of *structured software development* and *object-oriented software development*. It was in that background that *International Symposium on Formal Aspects of Component Software* (FACS) started first as a workshop. In this section, we introduce the key ideas, concepts and aspects of component-based software systems and their development. As a paper for FACS anniversary track, however, we first give a brief summary of the 20 years of the workshops and conferences of FACS[1].

Table 1: 20 years of the FACS workshops and conferences

FACS Year [Citation]	**Venue**	**Editors of Proceedings**	**Invited speakers**	**No. accepted papers**
FACS 2022 [68]	Virtual Event	S.L.T. Tarifa and J. Proença	C. Baier, R. Neves and V. Stolz	13
FACS 2021 [65]	Virtual Event	G. Salaün and A. Wijs	R. Calinescu and C. Pasareanu	9
FACS 2020	Cancelled due to Covid Pandemic			
FACS 2019 [4]	Amsterdam, The Netherlands	F. Arbab and S-S. Jongmans	C. Ghezzi, K.G. Larsen, and W. Fokkink	14
FACS 2018 [6]	Pohang, South Korea	K. Bae and P.C. Ölveczky	E.A. Lee and G. Rosu	15
FACS 2017 [63]	Braga, Portugal	J. Proença and M. Lumpe	D. Costa and C. Palamidessi	14
FACS 2016 [30]	Besançon, France	F. Arbab and P.C. Ölveczky	J. Meseguer, J. Rushby, and K. Stølen	16
FACS 2015 [9]	Niterói, Brazil	C. Braga and P.C. Ölveczky	M. Wirsing, D. Déharbe, and R. Cerqueira	17
FACS 2014 [32]	Bertinoro, Italy	I. Lanese and E. Madelaine	H. Veith, R. de Nicola, and J-B. Stefani	22
FACS 2013 [19]	Nanchang, China	J.L. Fiadeiro, Z. Liu and J. Xue	C. Zhou, A. Legay and J. Misra	22
FACS 2012 [62]	Mountain View, CA, USA	C.S. Păsăreanu and G. Salaün	T. Bultan and S. Qadeer	16
FACS 2011 [5]	Oslo, Norway	F. Arbab and P.C. Ölveczky	J. Meseguer, J. Rushby, and K. Stølen	21
FACS 2010 [8]	Guimarães, Portugal	L.S. Barbosa and M. Lumpe	S. Seshia and L. Caires	20
FACS 2009 [57]	Eindhoven, the Netherlands	M. Sun, B. Schätz	G. Döhmen and J. Rutten	12
FACS 2008 [12]	Malaga, Spain	C. Canal, C.S. Pasareanu	Lack of data	13
FACS 2007 [52]	Sophia-Antipolis, France	M. Lumpe, E. Madelaine	C. Pasareanu and E. Zimeo	17
FACS 2006 [56]	Prague, Czech Republic	V. Mencl, F.S. de Boer	P. Van Roy and D. Caromel	16
FACS 2005 [38]	Macao, China	Z. Liu, L.S. Barbosa	F. Arbab, P. Ciancarini, J. He and R. Hennicker	22
FACS 2003 [44]	Pisa, Italy,	Z. Liu and He J.	M. Broy and T. Maibaum	11

1.1 International Symposium on Formal Aspects of Component Software

The first FACS workshop was an one-day event associated with the *12th International Symposium of Formal Methods Europe* (FME), now called *International Symposium of Formal Methods* (FM), which was held September 8–14, 2003 in Pisa of Italy. The

[1] The first author of this paper is the founder of FACS.

purpose of the workshop was to promote understanding of the software paradigm shift and to explore how formal methods can augment understanding of component-based technology, thereby encouraging further research and educational pursuits in formal methods and tools for the component-based construction of software systems. The program consisted of two invited talks by Manfred Broy and Tom Maibaum and a few regular presentations. An edited volume “Mathematical Frameworks for Component Software” [44] was then published with 11 chapters, including the refined versions of the invited talks, presentations at the workshop, and some other papers submitted in response to the call for contributions of the volume. The papers focus on mathematical models that identify the “core” concepts as their first-class modelling elements, including *interfaces*, *contracts*, connectors, and *services*. Each chapter provides a clear definition of components, articulated through a set of key aspects, and discusses challenges related to the specification and verification of these aspects. Moreover, some papers delve into issues concerning the refinement, composition, coordination, and orchestration of software components in both individual software components and broader component-based software system development.

Originally, there were no intention to establish the FACS workshop as a recurring annual event. However, the overwhelming expression of interest led to its organisation again in 2005. Consequently, FACS workshop series have been conducted annually, with proceedings being published in Springer Electronic Notes in Theoretical Computer Science (ENTCS). In 2010, the FACS workshop series evolved into an international symposium. Subsequently, the proceedings have been documented in Springer Lecture Notes in Computer Sciences, and select papers have been chosen for publication in scholarly journals, predominantly in the Science of Computer Programming (JSCP).

Over the last two decades, FACS workshops and conferences have made good contributions to advancing research, education, and application in component-based software technology. The summary presented in Table 1 alongside the papers featured in the proceedings highlight the attraction of these FACS events for a number of excellent scientists and researchers who have showcased their work. Acknowledgements are extended to colleagues who have served as members of the steering committee, event organisers, members of the program committees, the reviewers, and editors of the proceedings. The utmost appreciation is reserved for the authors for their invaluable contributions. In this paper, we endeavour to cite proceedings from FACS workshops and conferences wherever possible to demonstrate their far-reaching impact.

Throughout the past 20 years, FACS workshops and conferences FACS have made their contributions to promote the research, education, and application in component-based software technology. The summary in Table 1 and the papers in the proceedings show that the FACS events have attracted a large number of excellent scientists and researchers to present their work there. We would like to pay tributes to the colleagues who have served as members of the steering committee, organisers of the events, members of the program committees, the reviewers and editors of the proceedings, and most of all to the authors, for their contributions. In this paper, we cite proceedings of FACS workshops and conferences wherever we can with the intention to show their impact.

1.2 Component-based software development

Component-based software or *component software* generally refers to software constructed from individual components. This idea has been present since the advent of assembly programming, which is perceived as the craft of assembling instructions or ”components” to create programs. Wheeler’s subroutines, also known as Wheeler jumping routines [69], can be considered as sizable, reusable ”components” of that era[2]. Subsequent to this, more generalised abstractions of functions and procedures were implemented in high-level programming languages such as FORTRAN and PASCAL.

It is widely agreed that the idea of developing software systems by using available software components was first proposed by Douglas McIlroy in his invited talk, *mass-produced software components* [55], during the NATO conference on software engineering. The conference was held in Garmisch, Germany, in 1968, to address the problem of “software crisis”. There, he called for the development of a “software component industry” to produce components which can be used in different jobs of software development.

The philosophy of constructing large software systems from components is also foundational to *Structured Programming* [18]. It is the cornerstone of object-oriented programming [20, 58] too. Indeed, it is reasonable to affirm that every programming language incorporates some form of abstraction mechanism that facilitates the design and composition of components.

Thus, researchers in CBSD frequently find themselves addressing the question of *whether CBSD distinctly differs from structured and object-oriented development*. In other words, answering the following questions:

1. What characteristics distinguish components in CBSD from program instructions, routines, functions, classes, objects, libraries of classes or routines, and packages?
2. What distinctions exist between the ”components,” ”composition,” and ”refinement” of components in CBSD and the analogous ”components,” ”composition,” and ”refinement” of programs in structured and object-oriented programming?

In this paper, our goal is to demonstrate how research on the formal aspects of component software can elucidate the aforementioned differences and relations. To accomplish this, we employ the rCOS method as the framework to identify various aspects of software components, component-based software systems, and their development, illustrating the methods to formalise, refine, decompose, and compose them. This allows for a clear comparison with the facets of structured and object-oriented software systems. Moreover, we argue for the need for various formal theories for modelling different aspects, and even different formal theories for the same aspects. We engage in a discussion on the interoperable use of these diverse formal theories. In addition, we outline research challenges and opportunities in component-based modelling and design of human-cyber-physical systems (HCPSs), which are networked systems with mixed intelligence.

[2] At a time when when there is lack of hardware support to remember the return address of the routine.

1.3 Organisation

In Section 2, we introduce the key characteristics of software components and present the calculi of *designs* and *reactive designs* as preliminaries. These will serve as the semantic basis for our discussions in subsequent sections. Section 3 follows, defining interfaces and contracts. The main content is housed in Section 4, where we provide an overview of formal models of aspects and discuss their support for the separation of concerns in CBSE. In Section 5, we outline the processes of software component development and system development in CBSE, discussing the principal shortcomings in the current state of the art and challenges to industrial adoption. In Section 6, we propose a future research direction for CBSE—specifically, component-based design and evolution of human-cyber-physical systems (HCPSs)—and discuss related challenges. We draw conclusions at the end of this section.

2 Preliminaries

In this section, we discuss the key characteristics of software components and introduce the basics of *design calculus* [46], which will serve as the semantic basis for linking different formal theories and techniques in modelling and verification of various aspects of component-based software.

2.1 Characteristics of software components

To address the two questions raised at the end of the previous section, we consider the definition of software component given by Szyperski in his book [66]:

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

This definition characterises the requirements for a "component" to be *reusable* in different software systems, which is also the major concern expressed in McIlroy's talk [55]. As highlighted in [28], the concept of reusability defined herein contrasts with the "reusability" achieved through generalisation in object-oriented programming. The latter necessitates the "rewriting" of classes (either generalised or specialised) being reused. The discussion in paper [28] extends further to apply the above four criteria to to discern whether elements like an assembly language instruction, a library routine, a class in a library, or a package can qualify as software components. In this context, it is concluded that an assembly language instruction does not qualify as a component due to its inability to be independently deployed. Library routines and classes may be considered components, but a package, lacking an interface, does not meet the criteria for a component. However, this reasoning is somewhat unconvincing, primarily due to a lack of rigour in both syntax and semantics.

It is important to note that the four characteristics must be described in the models of different kind of requirements of components. Each kind of requirements is called an *aspect* [28]. Components of different types of software systems have different aspects -

be they sequential, concurrent and distributed, real-time, or embedded systems. Questions are raised, for instance, about the aspects that need specification in the interface contract for a component within concurrent or embedded systems.

In this paper, we take the above four basic characteristics of components as the basis and describe them in a formal model of interface contracts. An interface contract of a component is defined in the design calculus of UTP [46] and it can be factorised into specifications of different aspects. We show how different aspects embraced in an interface contract of a component can be refined in different stages of the component development, from the requirements, through the design, to coding and evolution.

2.2 Designs

We use the notion of contracts to characterise the different aspects of components. There are different theories for different aspects, and a single aspect can also be modelled using various theories. For instance, when developing reactive components, Hoare logic can be applied to specify functionality, the theory of I/O automata can model reactive behaviuor, and sequence diagram notation can represent interactions with the environment. Nonetheless, Hoare logic is often used in conjunction with traces for interactions to compose reactive components. Moreover, models or specifications, whether addressing identical or differing aspects, must maintain consistent relations. Hence, there's a requirement for a model that facilitates the definition and verification of their consistency. To provide a formal definition of contracts and formalise various aspects, we first introduce the notations of *design* and *reactive designs*. These are deployed to define the meanings of sequential and reactive programs [46, 23, 22], respectively.

Definition 1 (Design). *Given a set of state variables X, a* **design** *D over X is first-order predicate of the form $p \vdash Q$ with free variables in $X \cup \{ok\}$ and $X' = \{x' \mid x \in X\} \cup \{ok'\}$ The semantics of the design is defined as the implication $(ok \wedge p) \rightarrow Q \wedge ok'$, where*

- *a variable $x \in X$ represents the initial value of x in the state when the program starts to execute, and the $x' \in X'$ the value of x in the state when the execution terminates;*
- *p is predicate which is called the* **precondition** *and it only refers to free variables in X, and Q the* **postcondition** *and only refers to free variablesin X and X'; and*
- *ok and ok' are Boolean variables representing the observations about if the execution terminates, and the program is in a terminated state when ok is true, and the execution terminates when ok' is true.*

The set X of variables is called the **alphabet** of the design D. A *state* of D is a mapping s which assigns each variable x in X a value an *ok* a Boolean value. The precondition p and postcondition Q of a design are interpreted on the set of pairs (s, s') of states. The meaning $(ok \wedge p) \rightarrow Q \wedge ok'$ of $p \vdash Q$ says that if the execution starts well in a state s (i.e. ok is *true* in s) in which the precondition p holds, the execution will terminates in a state s' (i.e. ok' is *true* in s) such that the postcondition Q holds for (s, s').

Let $X = \{x, y, z\}$ be a set of integer variables, the design

$$x > 0 \vdash (x' = (x - 1)) \wedge (y' = x) \wedge (z' = z)$$

specifies a program such that

- from any state s in which the value of x is positive, i.e. $s(x) > 0$, its execution terminates and the execution decrements the initial value x by 1, assigns y to the initial value of x, and does not change the initial value of z, i.e. the execution terminates in a state s' such that $(s'(x) = s(x) - 1) \wedge (s'(y) = s(x)) \wedge (s'(z) = s(z))$; and
- from any state s in which the value of x is not positive, i.e. $s(x) \leq 0$, any execution is possible (acceptable), including non-terminating execution.

Program **if** $(x > 0)$ $\mathbf{then}$ $(y := x; x := x - 1)$ correctly implements this specification. Notice the difference between this design from the design below

$$x > 0 \vdash x' = (x - 1) \wedge (y' = x')$$

Note that the set of designs over X form a proper subset of first-order predicates with free variables in X, X' and $\{ok, ok'\}$. We thus require that the set of designs is closed to the common structure operations of programs, such as *assignments*, *sequencing*, *choice*, and *iteration*. To this end, we define the following primitives and operations on the set $\mathcal{P}_X = \{D \mid D \text{ is a design over } X\}$

- *skip*: $\mathbf{skip} \overset{def}{=}$ *true* $\vdash \wedge_{x \in X}(x' = x)$, that is, *program* $\mathbf{skip}$ *always terminates but its execution does not change any variables*;
- *chaotic program*: $\mathbf{chaos} \overset{def}{=} fale \vdash true$, that is, *program* $\mathbf{chaos}$ *behaves as chaos and may exhibit any behaviour*;
- **assignment**: $x := e \overset{def}{=}$ *true* $\vdash (x' = e) \wedge_{y \in X - \{x\}} (y' = y)$, namely, *this assignment always terminates, and its execution only changes the variable x to the value of expression x obtained in the initial, keeping the other variables unchanged* (it has no side effect);
- **sequencing**: $(D_1; D_2) \overset{def}{=} \exists (X_o, b).(D_1[X_o/X', b/ok'] \wedge D_1[X_o/X, b/ok])$, that is, this *sequence statement first executes D_1 and reaches a state (X_o, b) and then it executes D_2 from this state*;
- **conditional choice**: $D_1 \lhd B \rhd D_2 \overset{def}{=} (B \wedge D_1) \vee (\neg B \wedge D_2)$, that is, *this conditional choice statement behaves like D_1 if B is evaluated to true in the initial state, it behaves like D_2, otherwise;*
- **loop**: $(B * D) \overset{def}{=} \mu\mathcal{X}.(D; \mathcal{X}) \lhd B \rhd \mathbf{skip}$, that is, *the loop statement is the least fixed point of the recursion $\mathcal{X} = (D; \mathcal{X}) \lhd B \rhd$* $\mathbf{skip}$.

The above formalisation is given based on the following assumptions.

1. For the assignment, the type of x and that of e are *compatible*, and the value of e in the initial state is defined. These conditions can be explicitly expressed in precondition. We omit them for simplicity.
2. Similarly, the expression B in the conditional choice and loop is also assumed to be evaluated as a Boolean value.

For the sequence statement, b is either *true* or *false*. In former case, the execution of D_1 terminates well and execution of D_2 starts well (i.e. $ok = true$), and in the latter case the execution of D_1 is chaotic, and so is the whole sequence statement.

It is important to note that the existence of (least) fixed-point for the loop is proved based on the theorem that the partial order $(\mathcal{P}, \sqsubseteq)$, forms a *complete lattice*. In the lattice, the partial order $\sqsubseteq$ of the lattice is called the *refinement relation* among designs, and it is defined as $D_1 \sqsubseteq D_2$ if $D_2 \rightarrow D_1$ is valid. The bottom element is $\mathbf{chaos}$ and the top element is the *angelic design* defined by $true \vdash false$, denoted as $\mathbf{angel}$. Thus, per Tarski's fixed-point theorem, the least fixed point of the loop indeed exists. We proceed to present the ensuing theorem about designs [46].

Theorem 1. *The set $\mathcal{P}_X$ of designs over X is closed with respect to the program structure operations. That is, given any two design D, D_1 and D_2 and any Boolean expression B such that B only mentions variables in X, the formulas defined for* $\mathbf{skip}$*,* $\mathbf{chaos}$*, assignment, sequencing, conditional choice and loop are also designs.*

We can add the non-deterministic choice operation $(D_1 \sqcap D_2)$, which is defined by $(D_1 \vee D_2)$, for nondeterministic programming language.

We notice that while both the left zero law $((\mathbf{chaos}; D) = \mathbf{chaos})$ and the left unit law $((\mathbf{skip}, D) = D)$ hold for designs, neither the right zero law $((D; \mathbf{chaos}) = \mathbf{chaos})$ nor the right unit law $((D; \mathbf{skip}) = D)$ generally apply. We enforce these laws as *healthiness conditions* on the set of designs [46]. Furthermore, the assumption of side effect freedom for assignments in OO programming is generally not valid. In rCOS, we have developed a calculus of designs tailored for OO programming languages [23].

The design calculus is used to define the semantics of imperative (or procedural) programming languages, which are utilised for the implementation of components within the paradigm of structured software engineering. Meanwhile, the OO extension of the calculus in rCOS [23] can be employed to define the semantics of OO programming languages for implementing components within an OO software development paradigm [17]. Nonetheless, components implemented using these languages lack abstractions for concurrency and synchronisation.

2.3 Reactive designs

To model communication and synchronisation, we define the notion of *reactive design*. This is done by incorporating two fresh Boolean variables, *wait* and *wait'*, to represent *observables of synchronisation*. A design D on an alphabet X with observables $\{ok, ok', wait, wait'\}$ is called a reactive design if it satisfies the healthiness condition $\mathcal{W}(D) = D$, where the transformation $\mathcal{W}$ is defined as folows:

$$\mathcal{W}(D) \overset{def}{=} (true \vdash wait' \wedge ((X' = X) \wedge ok = ok'))) \lhd wait \rhd D$$

To specify a reactive program, it's necessary to specify the synchronisation conditions of an activity with other activities, as well as the functionality of the activities. To this end, we introduce the concept of *guarded designs* represented as $g \& D$, where D is a design, and g is a Boolean expression of the given X, called the *guard* of the design.

The semantics of $g\&D$ is defined as $D \lhd g \rhd (\textit{true} \vdash \textit{wait}' \wedge (X' = X) \wedge (ok = ok'))$, where X is the alphabet of the design D.

It is straightforward to prove that the transformation $\mathcal{W}$ is monotonic (with respect to the implication), thereby establishing that the set of reactive designs constitutes a complete lattice. Moreover, the subsequent properties are valid for reactive designs.

Theorem 2. *For a given set of variables X,*

(1) For any design D, $\mathcal{W}^2(D) = \mathcal{W}(D)$;
(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: $(\mathcal{W}(D_1) \lhd b \rhd \mathcal{W}(D_2)) = \mathcal{W}(D_1 \lhd b \rhd D_2)$.

Note that "$=$" stands for logical equivalence. Furthermore, the following properties hold for guarded designs.

Theorem 3. *For a given set of variables X and if D, D_1 and D_2 are reactive designs,*

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

These properties, together with the subsequent transformations for programming commands, enable us to define a reactive program c as reactive designs of the form $g\&\mathcal{W}(c)$, where c is defined as follows:

$$c ::= \mathbf{skip} \mid \mathbf{stop} \mid \mathbf{chaos} \mid x := e \mid c \lhd B \rhd c \mid B * c$$

And

$$\begin{aligned}
&\mathcal{W}(\mathbf{skip}) \overset{def}{=} \mathcal{W}(\textit{true} \vdash \neg \textit{wait}' \wedge (X' = X)) \\
&\mathcal{W}(\mathbf{stop}) \overset{def}{=} \mathcal{W}(\textit{true} \vdash \textit{wait'} \wedge (X' = X)) \\
&\mathcal{W}(\mathbf{chaos}) \overset{def}{=} \mathcal{W}(\textit{fale} \vdash \textit{true}) \\
&\mathcal{W}(x := e) \overset{def}{=} \mathcal{W}(\textit{true} \vdash \neg \textit{wait}' \wedge (x' = e) \wedge \bigwedge_{y \in X - \{x\}} (y' = y)) \\
&\mathcal{W}(c_1 \lhd B \rhd c_2) \overset{def}{=} \mathcal{W}(\mathcal{W}(c_1) \lhd B \rhd \mathcal{W}(c_2)) \\
&\mathcal{W}(B * c) = \mu.\mathcal{X}.(\mathcal{W}(c); \mathcal{X}) \lhd B \rhd \mathcal{W}(\mathbf{skip})
\end{aligned}$$

3 Interfaces and Contracts

We hold the view that there should be few restrictions on what qualifies as a component. A component can be, for instance, a few lines of code for a function, procedures, a service provider, a substantial component for internet search or managing databases, or a reactive system. Regardless, they should be deployable for execution and ought to have specified interfaces, which serve as the sole interaction points with the environment.

Interfaces are most important for components and their composition. Thus, we define them as the first-class model elements in our rCOS modelling theory [41, 22].

Definition 2 (Interfaces). *An* **interface** *is a triple $\mathcal{I} = (T, X, O)$, where T is a set of type definitions, X a set of state variables with their types defined in T, and O is a set of operation signatures of the form $m(T_1\ x; T_2\ y)$ with an input parameter of type T_1 and an output parameter of type T_2.*

We permit T to encompass a set of class definitions in an OO programming language, facilitating the object-oriented implementation of components. Either the input parameter x or the output parameter y, can be vectors, possibly empty. The set X of variables denotes the state encapsulated in the component and can also be empty. A component is termed stateless when X is empty and stateful otherwise. The names in an interface are designated as the *syntactic aspects* of types, states, and functionality. These names must be derived from predefined formal languages. For instance, elements of T are names of defined types within a type theory; elements in X are well-defined identifiers in a specified programming language, and elements in O are signatures from an Interface Definition Language (IDL). These stipulations ensure syntactic consistency and well-formedness checking.

It is important to note that a component can have multiple interfaces. In rCOS, we handle this requirement at the syntactic level by defining "merging" or "union" of interfaces [22]. We now give the definition of interface contracts.

Definition 3 (Contracts). *A* **contract** *is a tuple $\mathcal{C} = (\mathcal{I}, \theta, \Phi)$, where*

- *$\mathcal{I} = (T, X, O)$ is an interface,*
- *θ is a first order predicate on X which is the set of state variables of interface $\mathcal{I}$ specifying the allowable initial states of the program, called the* **initial condition***,*
- *Φ is a mapping which assigns each interface operation $m(T_1\ x; T_2 : y) \in O$ a guarded deign with input alphabet $X \cup x$ and output alphabet $X' \cup y'$.*

We observe that the contract of an interface generally defines a model for concurrent (or reactive) programs. Such models can be specified in a well-established formal theory, for instance, in the *temporal logic of actions* (TLA) [31] as a *normal form* $\theta \wedge \Box[\bigvee_{m() \in O} \Phi(m())]_X$. A well-established formal theory for concurrent programs, like TLA, facilitates the development of concurrent systems from scratch. Other renowned theories in this domain include UNITY [13], Event-B [1], and the stream calculus [10]. Such as theory can be be used for the development of concurrent systems from scratch by going through a whole process of specification, decomposition, composition and verification. However, these theories often do not provide explicit support for the separation of concerns or for *black-box* integration (whether composition or assembly). Moreover, event-based theories, such as those based on *input/output automata* [54], CSP [27, 64], CCS [59], and other process algebras [7], predominantly emphasize interaction and concurrency aspects.

We believe that these limitations play a significant role in the industry's lukewarm adoption of component-based development technologies. More effort should be dedicated to creating integrated development environments (IDEs) that support consistent use of various theories, techniques, and tools for diverse aspects. In the next section, we identify these aspects and identify the formal theories for their specification, decomposition, verification, and refinement.

4 Models of Aspects of Contracts

To address the aforementioned limitations, component-based development should ideally support the composition of components based on interface specifications in a black-box manner. Furthermore, it should enable an interface model to be divided into models addressing different aspects, thereby facilitating the separation of concerns in the development process. We will now delve into such a factorisation, drawing upon the interface model defined in the preceding section.

In general, a formal theory of each aspect is a mathematical logic system or a universal many-sorted algebra, which consists of a *formal language*, its *semantics* (interpretations), and a proof system defined by *axioms* and *inference rules*. We now discuss the aspects interface contracts and their formal theories.

4.1 Type systems

Definitions 2&3 show that an interface contract is based on a *type system*. This represents a formal theory essential for addressing the *data aspect*, ensuring consistency checks for data representations of states in X, signatures of operations, and the expressions in guards and designs. The type system for an interface contract can be distinct from the type systems in various programming languages utilised for the component implementations. Nonetheless, transformations must be defined to bridge the abstract type system at the interface level with those in the programming languages. Established theories of *abstract data types* (ADT) and *algebraic specification*, such as [11], serve this purpose effectively.

Here we highlight the rCOS method [43, 23, 72, 29], which provides a semantic theory for OO programs, encompassing an object-oriented type system. This theory extends the design calculus, offering the subsequent features:

- Variables are categorised as *public*, *protected*, or *private*. The types of variable values can either be *primitive types* or *classes*.
- An object is recursively defined as a graph structure. Nodes represent objects, while directed edges, labelled by the attribute names in the source node, denote references from one node to another (akin to a UML object diagram). Such a graph contains a unique root node, symbolising the current object.
- The system supports type casting through dynamic type binding of method invocations and type safety analysis.
- At any given execution moment, the program's state is an object graph termed the *state graph*. This represents the main program's object, implying the root is the main class's object. The object nodes within this state graph encapsulate the dynamic object types.
- Program command execution transitions from one state to another by either adding a new object to the graph (e.g., opening a new account in either a small or large bank system), altering attribute values of certain graph objects (like the *transfer()* action in a banking system), or modifying graph edges (such as enabling customer access to a shared account). Thus, the command's semantics (including method invocations) is expressed as a relation between states in UTP design form.

– To bolster incremental program development, a class declaration is also specified as a design, capturing changes in the program's static class structure. This can be seen as a textual formalisation of a UML class diagram. The class declaration is an action undertaken prior to program compilation.

Drawing from this semantic theory, *OO refinement* is articulated across three dimensions: *refinement of commands*, which encompasses method invocations, *refinement of classes* (also dubbed *OO structure refinement*), and *refinement of programs*. Class refinement also signifies *sub-typing*. Program refinement includes both the extension and modification of the program's class declarations, coupled with the refinement of the main method and other class methods. This research is elaborated in the paper [23]. The rCOS theory of OO semantics is also applicable to components within an OO programming paradigm. The soundness and (relative) completeness of the OO refinement are proven in [72].

4.2 Functionality and synchronisation behaviour

This model of interface contracts defines the requirements of a component, including both functional requirements and synchronisations conditions of each individual interface operation. From an operational perspective, we can divide this model into three parts:

– $\mathcal{R}$: Represents the *functional aspects* of the contract. It is a mapping that assigns a design (not a reactive design) to each operation $m()$.
– $\mathcal{S}$: Denotes a labelled state transition system. Using symbolic states, it characterises the reactive behaviour.
– $\mathcal{A}$: Captures the *data state aspects* of the contract. It is a mapping that associates each state s of the transition statement with a (or a set of) predicate formula $\mathcal{A}(s)$.

$\mathcal{R}$, $\mathcal{A}$ and the state transition system are related such that for any state transition $s \overset{m()}{\to} s'$ from s to s' by an operation $m()$ of the interface $\mathcal{I}$, it is required that the guard g holds in s and that design $\Phi(m())$ holds for (s, s'). Here, we say that g holds in s if $\mathcal{A}(s) \to g$, and (s, s') holds for $\Phi(m())$, if $p \to \mathcal{A}(s)$ and $\mathcal{A}(s') \to Q$, where p and Q are the precondition and postcodnition of $\Phi(m())$, respectively.

Therefore, the functional aspect is specified, analysed, refined, and verified within its own theory, be it Hoare Logic [26], the design calculus in the Unifying Theory of Programming (UTP) [46], or the rCOS OO design calculus in [23]. On the other hand, the synchronisation aspect is modelled, designed, refined and verified using its dedicated theory, such as labelled state transition systems, interface automata [2], or UML state diagrams. It is important to note that there is not an automatic method (neither an algorithm nor syntactic rules) to factor a given contract. Typically, we construct the models for these aspects using a use-case driven approach, then merge them into a contract using the syntactic rules presented in [14], followed by a consistency check (refer to Section 5).

4.3 Interaction protocols

When assembling components, checking interaction compatibility between them often benefits from a declarative protocol specification based on a set of allowable traces rather than an operational model. Consider a buffer with a capacity of one: it only permits interaction traces that alternate between *put*$()$ and *get*$()$, beginning with *put*$()$. More generally, a buffer with capacity n would accept traces that start with *put*$()$. In any prefix of such traces, the number of *get*$()$ occurrences should not exceed that of *put*$()$, and the count of *put*$()$ should not be greater than n times the number of *get*$()$. This aspect can be addressed using the theory of regular expressions, process calculi, or sequence charts.

Often, we can express the specification of an interface contract in the simplified form $(\mathcal{I}, \theta, \mathcal{R}, \mathcal{T})$, where:

- $\mathcal{I}$ represents the interface,
- θ denotes the initial condition,
- $\mathcal{R}$ assigns a design to each operation signature $m()$ for its functionality aspect, and
- $\mathcal{T}$ is a set of traces of the form $\langle ?m_1(x_1), \ldots, ?m_k(x_k)\rangle$, representing the interaction protocol between the contract and its environment. Here, $?m_i(x_{i_j})$ signifies an invocation event of service m_i with the input value x_i.

One might wonder how such a specification relates to a contract defined in Definition 3. To our knowledge, no transformation directly from such a specification to a contract has been defined. However, one can envision a design of the component by initially designing the functionality of services without synchronisation control, followed by the design of the interaction protocol for the constraints of the invocations. Subsequently, we define the semantics of the design in terms of reactive designs. Essentially, the environment should not be blocked if it invokes the provider services following any of the traces in $\mathcal{T}$, and the execution of the invocation should be correct with respect to the functionality aspect $\mathcal{R}$. Several well-established theories support the specification and design of interaction protocols, including process calculi and sequence charts.

The main theme we propose here is the separation of the functionality and synchronisation aspects of communications. This separation enables the flexible combination of a simpler theory of functional design and component development, such as the rCOS relational calculus of components [42], with theories for component coordination, e.g. [3], and component orchestration, e.g. [60], for system assembly.

4.4 Dynamic behaviour

A component interacts with its environment through its interfaces, which serve as access points. Its behaviour can be described by potential sequences of alternating events of invocations and returns of provided operations (or services). It's often necessary to specify potential failures that can occur during execution, namely, *deadlock* and *livelock* (or *divergence*). In this context, we define the *dynamic behaviour* of a contract $\mathcal{C}$, as outlined in Definition 3, as a pair of sets $\mathcal{B}_\mathcal{C} = (\mathcal{D}_\mathcal{C}, \mathcal{F}_\mathcal{C})$ representing *divergences* and *failures*. In this section, we provide only informal definitions of $\mathcal{D}_\mathcal{C}$ and $\mathcal{F}_\mathcal{C}$, and direct the reader to the paper [22] for a more formal discussion.

Definition 4 (Divergences). *Given a contract $\mathcal{C}$, the set $\mathcal{D}_\mathcal{C}$ of* **divergences** *for $\mathcal{C}$ consists of sequences of invocations $\langle ?m_1(x_1)!m_1(y_1)\ldots ?m_k(x_k)!m_k(y_k)\rangle$ of provided operations $m_i()$. These sequences end in a divergent state, meaning the execution of $\theta; m_1(x_1;y_1);\ldots;m_i(x_i;y_i)$ in a prefix of the sequence from an initial state results in ok$'$ being false. Here, x_i and y_i represent the actual input and output parameters of the invocation $m_i()$, respectively.*

Notice that each pair of events $?m_i(x_i)!m_i(y_i)$ in the sequence corresponds the execution of an method invocation $m_i(x_i;y_i)$.

Definition 5 (Failures). *Given a contract $\mathcal{C}$, the set $\mathcal{F}_\mathcal{C}$ of* **failures** *for $\mathcal{C}$ consists of pairs (tr, M). Here, tr is a finite sequence of invocations $\langle ?m_1(x_1)!m_1(y_1)\ldots\rangle$ of provided operations, and M is a set of service invocations. One of the following conditions must hold:*

(1) tr is the empty sequence, and M consists of invocations $m(v)$ for which the guards of their designs are false in the initial condition θ.
(2) tr is a trace $\langle ?m_1(x_1)!m_1(y_1)\ldots ?m_k(x_k)!m_k(y_k)\rangle$, and M includes the invocations $m(v)$ such that after executing 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, i.e., $\theta; m_1(x_1,y_1);\ldots;m_k(x_k,y_k)$ implies $\neg g'$ for the guard g of the design of $m()$.
(3) tr is a trace $\langle ?m_1(x_1)!m_1(y_1)\ldots ?m_k(x_k)\rangle$ where the execution of the invocation $m_k(v)$ has not yet delivered an output, and M includes 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 comprises all invocations.
(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 now establish the relationship between a protocol $\mathcal{T}$ defined in the previous subsection, and the dynamic behavior. This, in turn, relates to the contract defined in Definition 3. First, we define the set of traces for a contract $\mathcal{C}$ based on its failures.

$$\mathit{Trace}(\mathcal{C}) \overset{def}{=} \{tr \mid \text{there exists a } M \text{ such that } (tr, M) \in \mathcal{F}_\mathcal{C}\}$$

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

Definition 6 (Consistent Protocol of Contract). *A protocol $\mathcal{T}$ is* **consistent** *with $(\mathcal{D}_\mathcal{C}, \mathcal{F}_\mathcal{C})$ (and hence with contract $\mathcal{C}$),if executing any prefix of any invocation sequence sq in $\mathcal{T}$ does not lead to a state where all invocations to the provided services are rejected. That is, for any sq $\in \mathcal{T}$ and any $(tr, M) \in \mathcal{F}_\mathcal{C}$ such that sq $=$ tr $\downarrow_?$, we must have $M \neq \{m(v) \mid m() \in O\}$ if tr $\downarrow_?$.*

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

Theorem 4. *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 every operation $m()$ of the interface, then $\mathcal{T}_1$ is consistent with $\mathcal{C}_1$ if it is consistent with $\mathcal{C}$.

It is important to note that a *largest protocol* (or *weakest protocol*) of a contract exists and can be derived from the contract. We refer to [22] for technical details. The importance of the notion of consistency and its properties lies in ensuring the correct use of components in various interaction environments. Furthermore, this allows for the design of functional aspects and the tailoring of communication protocols (i.e., coordination and orchestration) to be separated.

In rCOS [16, 17], we demonstrated how protocols are specified using sequence diagrams, with their semantics defined in terms of CSP [35]; dynamic behaviour is modelled using UML state diagrams, and their semantics is also defined in terms of the failure-divergence semantics of CSP [64]. As a result, consistency can be automatically checked [48]. However, with UTP as its semantic foundation, other theories for different aspects can be utilised and their integration can be defined in the model of contracts defined in Definition 3. The sustainability of components is supported by the refinement calculus of contracts.

Definition 7 (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. $\mathcal{D}_{\mathcal{C}_2} \subseteq \mathcal{D}_{\mathcal{C}_1}$; and
(2) $\mathcal{C}_2$ is not more likely to block the environment, i.e. $\mathcal{F}_{\mathcal{C}_2} \subseteq \mathcal{D}_{\mathcal{C}_1}$.

There is an effective and complete method to prove that one contract refines another using *downward simulation* and *upward simulation*. We refer to our paper [22] for technical details.

4.5 Other aspects

There are other aspects of components, including real-time and security, and even power consumption. In theory, these can be incrementally integrated into the above framework. For example, time can be represented using the timed sequence in RT-UML, timed transition systems [24], or timers or clocks defined in TLA [45]. We also believe that component-based architecture is quite effective for imposing various security policies, e.g. [47]. However, we leave these topics out of this paper due to the page limit.

5 Component-Based Development

It is widely agreed that component-based system development comprises both the creation of individual components and the system development using these available components. Components are expected to be heterogeneous, indicating that they can be software developed within various paradigms (potentially by different teams using distinct programming languages). Moreover, components can be hierarchical, suggesting that they can be assembled from other components. We assume that a component developer can construct hierarchical components, but the system developer might not necessarily be aware of these hierarchies. Nevertheless, the development process of component-based systems does involve specifying the architecture for component composition or integration. It is worth noting that the composition language (e.g., Orc [60] or Reo [3]) might differ from the languages utilized in component development.

It is a common contention that the system development process should be bottom-up, with the system assembled from pre-implemented or even pre-deployed components. However, such a stance is somewhat idealistic and not always practical, given the ever-evolving nature of applications. Additionally, we assert that there is always a necessity for a system requirements specification when developing a new system, even if it is based on an existing one. Moreover, there's a need to design a system architecture model derived from this requirements specification. This design would facilitate the identification of existing components, their interfaces, and protocols, ideally within a component repository equipped with management tool support. A significant challenge lies in the identification process. This involves mapping naming schemes from the requirements model to the architecture model and subsequently to pre-implemented components, such as those in component repositories. Arguably, this challenge is the most significant barrier to the effective industrial adoption of component-based technology.

Nevertheless, rCOS offers a framework for OO component-based software development, emphasising use case-driven system requirements specifications. From these specifications, a model of the system architecture is derived. Each use case is articulated as an interface contract, and the relationships among use cases, represented in UML use-case diagrams, are formalised as dependencies between interfaces. This forms an initial model of the system architecture. We will now provide an overview of this development process. For a more detailed discussion, we refer readers to our papers [17, 37].

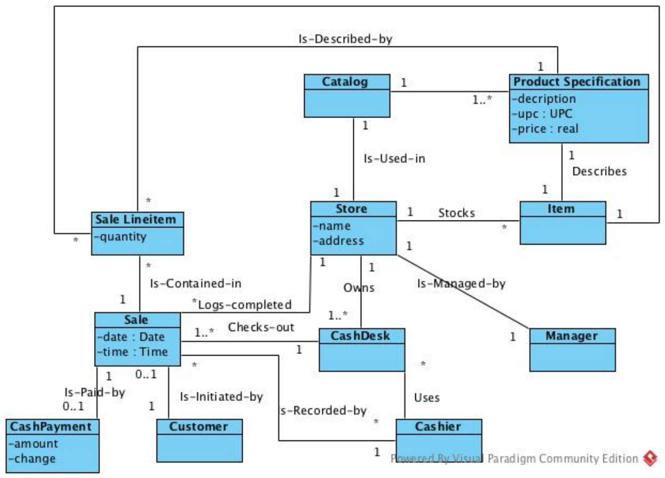

Fig. 1: An example of a conceptual class diagram

5.1 Use case as components

A requirements model comprises a set of interrelated use cases identified from the application domain. Each use case is modeled as an interface contract for a component. In the following discussion, we will briefly outline the steps to create a model of a use case as an interface contract. For technical details, readers are referred to rCOS-related papers, specifically the two cited here [15, 17].

Identify and represent the interface of a use case As demonstrated in the papers, an informal description of the use case is first provided using structured natural language. The operations of the provided interface for the component corresponding to a use case consist of interactions with the actors, and their symbolic names are then assigned. State variables serve as symbolic representations of the objects that must be recorded, checked, and updated, in accordance with the need to know policy [33, 36].

Identify classes and their associations The use case describe a domain process and the description involves domain concepts and objects, which the use cane needs to record, check, modify and and communicates. Then an initial class diagram can then be constructed by giving symbolic names to this concepts and their relations. Fig. 1 is the initial class diagram for use case "Process Sale" of the CoCoMe Example [25, 50].

Representing the interaction protocol An interaction of an actor and the component is the possible events with which the actor triggers the system for the execution of an interface action. The interaction protocol between the actors and the component is represented by a sequence diagram and it is created based on the description of the use-case. Fig. 2 is the initial sequence diagram use case "Process Sale" of the CoCoMe Example.

Specify functionality of interface operations The functionality of each interface operation is specified by a design, which consists of a pair of pre- and post-conditions. These conditions can initially be described informally and later formalised for consistency checking and validation. The preconditions emphasise the properties of existing objects that need to be checked, while the postconditions focus on describing what new objects are created, what changes are made to attributes, what new links between objects are formed. For examples informal descriptions of functionality of interface operations and their formalisation, we refer the reader to the papers [25, 50, 17].

Represent dynamic behaviour The dynamic behavior of a use case is modeled using a state diagram. This model aids in the verification of system properties, including both safety and liveness, by employing model checking techniques.

The models for the aforementioned aspects of a component, corresponding to a use case, collectively form the interface contract for that use case. Completion and consistency checks must be conducted within the framework of the rCOS interface contracts discussed in Section 3. With the interface contracts of the components of the use cases, we can create the UML component diagram as the diagrammatic representation of the system architecture at the requirements modelling phase.

5.2 Component development process

We propose that the design process for a component begins with an OO design approach and concludes by transforming the OO design models of certain components

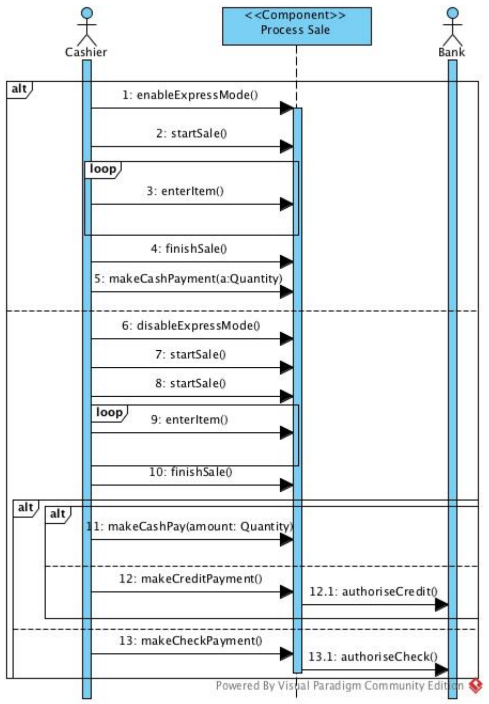

Fig. 2: An example of use-case sequence diagram

into component-based design models. Due to space limitations, we will only outline the steps of the OO design process for a component, illustrating what models should be produced:

1. Begin the process by taking each use-case component and designing each of its provided operations according to its pre- and post-conditions using the OO refinement rules, especially the four patterns of GRASP [33, 17].
2. Decompose the functionality of each use-case operation into internal object interaction and computation, refining the use-case sequence diagram into an object sequence diagram for the use-case [17, 35].
3. During the decomposition of the functionality of use-case operations to internal object interaction and computation, refine the requirements class model into a design class model. This involves adding methods and visibilities in classes based on responsibility assignments and direction of method invocations [17].
4. Select some of the objects in an object sequence diagram as candidate component controllers. These candidates should pass an automatic check ensuring they meet six given invariant properties. Following this, transform the design sequence diagram into a component-sequence diagram [34].
5. Generate a component diagram for each use-case (this can be done automatically). This diagram should depict a decomposition of the use-case component from the

requirements model into a composition of sub-components. From this, the complete component-based architecture model at the requirements level is decomposed into a component-based design architecture model [34].

6. Coding from the design architecture model is straightforward and can largely be automated [51, 70].

The model transformations involved in the development process for a use-case component are depicted in Fig. 3.

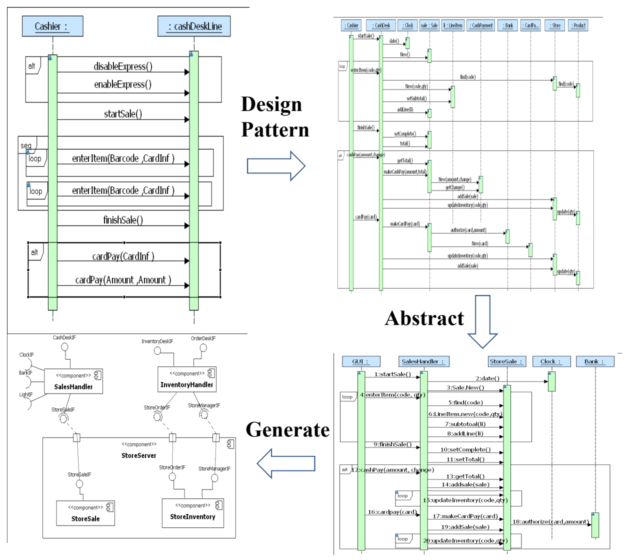

Fig. 3: Transformations from requirements to design of a component

5.3 System development

For a given application domain, we assume a repository exists containing implemented components for a multitude of use cases, their contract specifications, information on context dependencies, and possibly their sub-components [3].

Broadly speaking, system development begins with the creation of a requirements model based on contract use cases. These use case contracts are then refined and/or decomposed into component compositions to form a system architecture model. From this

[3] However, it should be noted that we are not aware of such an existing repository.

point, we search the repository for candidate components that could match a component in the architecture and verify if their contracts are refinements of the component's contract within the architecture. The checks for functional requirements and synchronization requirements can be performed separately. Additionally, they can be refined individually by adding connectors and coordinators, respectively.

It is possible that for some component contracts within the architecture, there are no suitable components that can be easily adapted for implementation. In such cases, we must use the method of component development discussed in the previous subsection. The primary features of component and system development in rCOS are shown in Fig. 4.

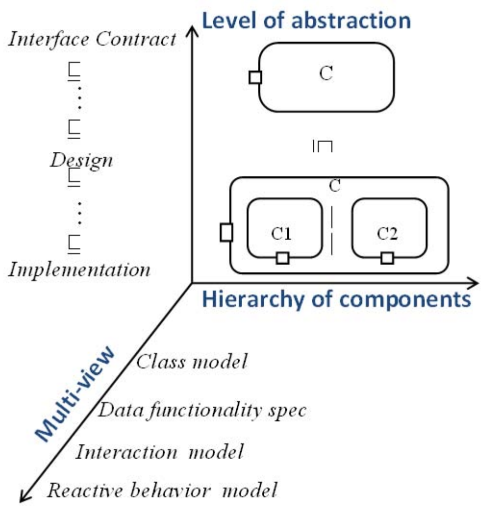

Fig. 4: Features of the rCOS modelling and development

Note that domain knowledge is crucial for providing the requirements model in terms of use cases, designing the architecture, and mapping them to components in the repository. The primary challenge in formalising the mapping and developing tool support stems from the varying naming schemes used in the requirements models, the architectural design, and the representations of the component models in the repository. In our opinion, significant effort is required in this area.

6 Future Development and Conclusion

From the discussions in the previous sections, it's evident that the modelling theories and design methods for transitional component-based software development are relatively mature. However, there are significant shortcomings in bridging the models in requirements, design, and those utilised in component repositories. These gaps are primary sources of challenges for industry adoption.

At the same time, computer systems are rapidly evolving to be more networked, hybrid, and larger in scale. Consequently, their software systems also become more intricate. Consider scenarios where components can be cyber-physical system (CPS) devices, artificial neural networks, or even humans. These systems are widely called *Human-Cyber-Physical Systems* (HCPS).

6.1 Extend rCOS for model human-cyber-physical systems

We are currently working on a project titled *Theory of Human-Cyber-Physical Computing and Software Defined Methodology*. Our aim is to extend the rCOS component-based modelling notation to model software architectures of HCPS. We view the architecture of an HCPS as comprising *cyber systems*, *communication networks*, *physical processes*, *human processes*, and *interfaces*. Specifically:

- Physical processes can encompass various operations, such as mechanical, electrical, and chemical processes.
- Cyber systems (or information systems) are computing entities. Within this:
  - Some are dedicated to data collection and processing.
  - Others, termed controlling systems, are responsible for making control decisions based on the data provided by the aforementioned systems to manage physical processes.
- Human processes involve making control decisions based on information received from information systems to guide physical processes.
- Interfaces serve as middleware between physical systems, cyber systems, and humans. These include sensors, actuators, A/C and C/A converters, among others.
- Sensors detect the physical processes, gather data about the behavior of these processes, and relay this data to the information systems via the network.
- Control decisions made by both computers and humans are dispatched as commands through the network to the appropriate actuators, which then execute the corresponding control actions.

There is a need for system software to coordinate and orchestrate the behaviors of the component systems, as well as for scheduling the physical, network, hardware, human, and software resources. Specifically, components that can facilitate the switching of control between human and computer controllers are crucial.

Our primary challenge is to extend the rCOS contracts of interfaces to accommodate *cyber-physical interfaces* (CP-interfaces) or *hybrid interfaces*. A CP-interface incorporates field variables that include both *signals*—representing information about the states of physical processes—and program variables. It encompasses both program operations and signals for interaction with the component's environment. Additionally, a signal can be either discrete or continuous. A contract for a CP-interface comprises a provided CP interface, a required CP interface, and a specification detailing the functionality of the program operations as well as the behaviour (expressed through differential or difference equations) of the signals in the interfaces.

We suggest defining the dynamic behavior of such a contract using two hybrid input/output automata [53]. These include one for the provided CP interface and another for the required CP interface. They should be specifiable in Hybrid CSP [21] and analysable using Hybrid Hoare logic [73]. Our preliminary ideas on this extension can be found in [40, 39, 49], with a proof-of-concept example provided in [61].

A significant challenge in modelling HCPS is the absence of a computational model and theory for human interactions with cyber and physical systems. We propose a model of *human-cyber-physical automata* (HCPA). In this model, human behaviour is represented by a neural network, and the controller responsible for control switching between human and machine is depicted as an oracle with an associated learning model. Importantly, we are not aiming to model generalised human intelligence but rather the behaviour of a human in a specific application when executing their tasks. Researching a comprehensive theory will require addressing the complexities of integrating traditional computational models with AI models. We have given an initial definition to HCPA that includes only one human process to control a physical process in tandem with digital controllers. This model, along with a proof-of-concept case study, is presented in the extended abstract of the invited talk [71]. A full and extended version of this work is now avialbe in the paper [67]. For further exploration of the research challenges in this project, we direct readers to the editorial paper [49] and the lecture notes available at [39]. The research will significantly encompass the controllability and composability of AI systems, their integration with traditional computational systems, and the reliability of these hybrid systems.

6.2 Conclusions

To commemorate the 20th anniversary of FACS, we aim to elucidate the meaning of ”formal aspects of component software.” In doing so, we provide an overview of the formal models and methods that can be used for different aspects of software components and component-based software systems. The central theme we wish to emphasise is, however, that the engineering principles of separation of concerns and divide and conquer necessitate the consistent application of various theories and methods tailored to distinct aspects. Using the rCOS framework as an example, we demonstrate these theories and methods, illustrating how and when they are applied consistently throughout development. From our discussion, it is evident that while the concepts and theories are robust, there remains a gap in the evolution of engineering techniques and tool support. Moreover, we recognise that research in CBSE must advance in tandem with developments in computer-based systems, notably in areas like human-cyber-physical systems (HCPS).

References

1. Abrial, J.R.: Modeling in Event-B: System and Software Engineering. Cambridge University Press (2010)
2. de Alfaro, L., Henzinger, T.A.: Interface automata. SIGSOFT Software Engineering Notes **26**(5), 109–120 (2001)

3. Arbab, F.: Coordinated composition of software components. In: Liu, Z., He, J. (eds.) Mathematical Frameworks for Component Software, pp. 35–68. World Scientific (2006)
4. Arbab, F., Jongmans, S. (eds.): Formal Aspects of Component Software - 16th International Conference, FACS 2019, Amsterdam, The Netherlands, October 23-25, 2019, Proceedings, Lecture Notes in Computer Science, vol. 12018. Springer (2020). https://doi.org/10.1007/978-3-030-40914-2, https://doi.org/10.1007/978-3-030-40914-2
5. Arbab, F., Ölveczky, P.C. (eds.): Formal Aspects of Component Software - 8th International Symposium, FACS 2011, Oslo, Norway, September 14-16, 2011, Revised Selected Papers, Lecture Notes in Computer Science, vol. 7253. Springer (2012). https://doi.org/10.1007/978-3-642-35743-5, https://doi.org/10.1007/978-3-642-35743-5
6. Bae, K., Ölveczky, P.C. (eds.): Formal Aspects of Component Software - 15th International Conference, FACS 2018, Pohang, South Korea, October 10-12, 2018, Proceedings, Lecture Notes in Computer Science, vol. 11222. Springer (2018). https://doi.org/10.1007/978-3-030-02146-7, https://doi.org/10.1007/978-3-030-02146-7
7. 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)
8. Barbosa, L.S., Lumpe, M. (eds.): Formal Aspects of Component Software - 7th International Workshop, FACS 2010, Guimarães, Portugal, October 14-16, 2010, Revised Selected Papers, Lecture Notes in Computer Science, vol. 6921. Springer (2012). https://doi.org/10.1007/978-3-642-27269-1, https://doi.org/10.1007/978-3-642-27269-1
9. Braga, C., Ölveczky, P.C. (eds.): Formal Aspects of Component Software - 12th International Conference, FACS 2015, Niterói, Brazil, October 14-16, 2015, Revised Selected Papers, Lecture Notes in Computer Science, vol. 9539. Springer (2016). https://doi.org/10.1007/978-3-319-28934-2, https://doi.org/10.1007/978-3-319-28934-2
10. Broy, M.: A theory for requirements specifications and architecture design. In: Liu, Z., He, J. (eds.) Mathematical Frameworks for Component Software, pp. 119–154. World Scientific (2006)
11. Broy, M., Wirsing, M.: On the algebraic extensions of abstract data types. In: Díaz, J., Ramos, I. (eds.) Formalization of Programming Concepts, International Colloquium, Peniscola, Spain, April 19-25, 1981, Proceedings. Lecture Notes in Computer Science, vol. 107, pp. 244–251. Springer (1981)
12. Canal, C., Pasareanu, C.S. (eds.): Proceedings of the 5th International Workshop on Formal Aspects of Component Software, FACS 2008, Malaga, Spain, September 10-12, 2008, Electronic Notes in Theoretical Computer Science, vol. 260. Elsevier (2010), https://www.sciencedirect.com/journal/electronic-notes-in-theoretical-computer-science/vol/260/suppl/C
13. Chandy, K.M., Misra, J.: Parallel Program Design: A Foundation. Addison-Wesley, Reading (1988)
14. 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., Plasil, F. (eds.) Proceedings of 33rd Conference on Current Trends in Theory and Practice of Computer Science, volume 4362 of Lecture Notes in Computer Science. pp. 819–831. Springer (2007)
15. Chen, Z., Hannousse, A.H., Hung, D.V., Knoll, I., Li, X., Liu, Y., Liu, Z., Nan, Q., Okika, J.C., Ravn, A.P., Stolz, V., Yang, L., Zhan, N.: Modelling with relational calculus of object and component systems–rCOS. In: Rausch, A., Reussner, R., Mirandola, R., Plasil, F. (eds.) The Common Component Modeling Example, volume 5153 of Lecture Notes in Computer Science, chap. 3, pp. 116–145. Springer, Berlin (2008)

16. 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. Lecture Notes in Computer Science, vol. 4700, pp. 83–114. Springer (2007)
17. Chen, Z., Liu, Z., Ravn, A.P., Stolz, V., Zhan, N.: Refinement and verification in component-based model driven design. Science of Computer Programming **74**(4), 168–196 (2009)
18. Dahl, O., Dijkstra, E., Hoare, C.: Structured Programming. Academic Press (1972)
19. Fiadeiro, J.L., Liu, Z., Xue, J. (eds.): Formal Aspects of Component Software - 10th International Symposium, FACS 2013, Nanchang, China, October 27-29, 2013, Revised Selected Papers, Lecture Notes in Computer Science, vol. 8348. Springer (2014). https://doi.org/10.1007/978-3-319-07602-7, `https://doi.org/10.1007/978-3-319-07602-7`
20. Goldberg, A., Robson, D.: Smalltalk-80: the language and its implementation. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1983)
21. He, J.: From CSP to hybrid systems. In: The Proc. of A Classical Mind: Essays in Honour of C. A. R. Hoare. Prentice-Hall (1994)
22. He, J., Li, X., Liu, Z.: A theory of reactive components. Electr. Notes Theor. Comput. Sci. **160**, 173–195 (2006)
23. He, J., Liu, Z., Li, X.: rCOS: A refinement calculus of object systems. Theoretical Computer Science **365**(1-2), 109–142 (2006)
24. Henzinger, T.A., Manna, Z., Pnueli, A.: Temporal proof methodologies for timed transition systems. Inf. Comput. **112**(2), 273–337 (1994). https://doi.org/10.1006/inco.1994.1060, `https://doi.org/10.1006/inco.1994.1060`
25. Herold, S., Klus, H., Welsch, Y., Deiters, C., Rausch, A., Reussner, R., Krogmann, K., Koziolek, H., Mirandola, R., Hummel, B., Meisinger, M., Pfaller, C.: The common component modeling example. In: Rausch, A., Reussner, R., Mirandola, R., Plášil, F. (eds.) The Common Component Modeling Example, volume 5153 of Lecture Notes in Computer Science, chap. 1, pp. 16–53. Springer-Verlag, Berlin, Heidelberg (2008)
26. Hoare, C.A.R.: An axiomatic basis for computer programming. Communications of the ACM **12**(10), 576–580 (1969)
27. Hoare, C.A.R.: Communicating sequential processes. Communications of the ACM **21**(8), 666–677 (1978)
28. Holmegaard, J.P., Knudsen, J., Makowski, P., Ravn, A.P.: Formalisization in component-based developmen. In: Liu, Z., He, J. (eds.) Mathematical Frameworks for Component Software, pp. 271–295. World Scientific (2006)
29. Ke, W., Liu, Z., Wang, S., Zhao, L.: A graph-based generic type system for object-oriented programs. Frontiers Comput. Sci. **7**(1), 109–134 (2013). https://doi.org/10.1007/s11704-012-1307-8, `https://doi.org/10.1007/s11704-012-1307-8`
30. Kouchnarenko, O., Khosravi, R. (eds.): Formal Aspects of Component Software - 13th International Conference, FACS 2016, Besançon, France, October 19-21, 2016, Revised Selected Papers, Lecture Notes in Computer Science, vol. 10231 (2017). https://doi.org/10.1007/978-3-319-57666-4, `https://doi.org/10.1007/978-3-319-57666-4`
31. Lamport, L.: The temporal logic of actions. ACM Transactions on Programming Languages and Systems **16**(3), 872–923 (1994)
32. Lanese, I., Madelaine, E. (eds.): Formal Aspects of Component Software - 11th International Symposium, FACS 2014, Bertinoro, Italy, September 10-12, 2014, Revised Selected Papers, Lecture Notes in Computer Science, vol. 8997. Springer (2015). https://doi.org/10.1007/978-3-319-15317-9, `https://doi.org/10.1007/978-3-319-15317-9`
33. Larman, C.: Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process. Prentice-Hall, Upper Saddle River, 2nd edn. (2001)

34. 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.) Formal Aspects of Component Software – 8th International Symposium, FACS 2011, Oslo, Norway, September 14-16, 2011, Revised Selected Papers. Lecture Notes in Computer Science, vol. 7253, pp. 97–114. Springer (2011)
35. Li, X., Liu, Z., He, J.: A formal semantics of UML sequence diagram. In: 15th Australian Software Engineering Conference (ASWEC 2004), 13-16 April 2004, Melbourne, Australia. pp. 168–177. IEEE Computer Society (2004)
36. Liu, Z.: Software development with UML. Tech. Rep. 259, IIST, United Nations University, P.O. Box 3058, Macao (2002)
37. Liu, Z.: Linking formal methods in software development - a reflection on the development of rCOS. In: Bowen, J., Li, Q., Xu, Q. (eds.) Theories of Programming and Formal Methods, Lecture Notes in Computer Science, vol. 14080, pp. 52–84. Springer (2023)
38. Liu, Z., Barbosa, L.S. (eds.): Proceedings of the International Workshop on Formal Aspects of Component Software, FACS 2005, Macao, October 24-25, 2005, Electronic Notes in Theoretical Computer Science, vol. 160. Elsevier (2006), `https://www.sciencedirect.com/journal/electronic-notes-in-theoretical-computer-science/vol/160/suppl/C`
39. Liu, Z., Bowen, J.P., Liu, B., Tyszberowicz, S.S., Zhang, T.: Software abstractions and human-cyber-physical systems architecture modelling. In: Bowen, J.P., Liu, Z., Zhang, Z. (eds.) Engineering Trustworthy Software Systems - 5th International School, SETSS 2019, Chongqing, China, April 21-27, 2019, Tutorial Lectures. Lecture Notes in Computer Science, vol. 12154, pp. 159–219. Springer (2019)
40. Liu, Z., Chen, X.: Interface-driven design in evolving component-based architectures. In: Hinchey, M.G., Bowen, J.P., Olderog, E. (eds.) Provably Correct Systems, pp. 121–148. NASA Monographs in Systems and Software Engineering, Springer (2017)
41. Liu, Z., He, J., Li, X.: Contract oriented development of component software. In: Lévy, J., Mayr, E.W., Mitchell, J.C. (eds.) Exploring New Frontiers of Theoretical Informatics, IFIP 18th World Computer Congress, TC1 3rd International Conference on Theoretical Computer Science (TCS2004), 22-27 August 2004, Toulouse, France. IFIP, vol. 155, pp. 349–366. Kluwer/Springer (2004)
42. Liu, Z., He, J., Li, X.: rCOS: A relational calculus for components. In: Liu, Z., He, J. (eds.) Mathematical Frameworks for Component Software, pp. 207–238. World Scientific (2006)
43. Liu, Z., He, J., Li, X., Chen, Y.: A relational model for formal object-oriented requirement analysis in uml. In: 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, pp. 641–664. Springer (2003)
44. Liu, Z., Jifeng, H. (eds.): Mathematical fremworks for component software. World Scientific (2006)
45. Liu, Z., Joseph, M.: Specification and verification of fault-tolerance, timing, and scheduling. ACM Transactions on Programming Languages and Systems **21**(1), 46–89 (1999)
46. Liu, Z., Kang, E., Zhan, N.: Composition and refinement of components. In: Butterfield, A. (ed.) Post Event Proceedings of UTP08, volume 5713 of Lecture Notes in Computer Science. Springer, Berlin (2009)
47. Liu, Z., Morisset, C., Stolz, V.: A component-based access control monitor. In: Margaria, T., Steffen, B. (eds.) Leveraging Applications of Formal Methods, Verification and Validation, Third International Symposium, ISoLA 2008, Porto Sani, Greece, October 13-15, 2008. Proceedings. Communications in Computer and Information Science, vol. 17,

pp. 339–353. Springer (2008). https://doi.org/10.1007/978-3-540-88479-8_24, https://doi.org/10.1007/978-3-540-88479-8_24

48. Liu, Z., Stolz, V.: The rCOS method in a nutshell. In: Fitzgerald, J., Larsen, P.G., Sahara, S. (eds.) Modelling and Analysis in VDM: Proceedings of the Fourth VDM/Overture Workshop. No. CS-TR-1099 in Technical Report Series, Newcastle University (2008)
49. Liu, Z., Wang, J.: Human-cyber-physical systems: concepts, challenges, and research opportunities. Frontiers Inf. Technol. Electron. Eng. **21**(11), 1535–1553 (2020)
50. Liu, Z., Zhang, Z. (eds.): Engineering Trustworthy Software Systems - First International School, SETSS 2014, Chongqing, China, September 8-13, 2014. Tutorial Lectures, Lecture Notes in Computer Science, vol. 9506. Springer (2016)
51. 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)
52. Lumpe, M., Madelaine, E. (eds.): Proceedings of the 4th International Workshop on Formal Aspects of Component Software, FACS 2007, Sophia-Antipolis, France, September 19-21, 2007, Electronic Notes in Theoretical Computer Science, vol. 215. Elsevier (2008), https://www.sciencedirect.com/journal/electronic-notes-in-theoretical-computer-science/vol/215/suppl/C
53. Lynch, N., Segala, R., Vaandrager, F.: Hybrid I/O automata. Information and Computation **185** (2003)
54. Lynch, N.A., Tuttle, M.R.: An introduction to input/output automata. CWI Quarterly **2**(3), 219–246 (1989)
55. McIlroy, M.D.: Mass produced software components. In: Software Engineering: Report of a conference sponsored by the NATO Science Committee, Garmisch, Germany, 7-11 Oct. 1968. Scientific Affairs Division, NATO (1969)
56. Mencl, V., de Boer, F.S. (eds.): Proceedings of the Third International Workshop on Formal Aspects of Component Software, FACS 2006, Prague, Czech Republic, September 20-22, 2006, Electronic Notes in Theoretical Computer Science, vol. 182. Elsevier (2007), https://www.sciencedirect.com/journal/electronic-notes-in-theoretical-computer-science/vol/182/suppl/C
57. Meng, S., Schätz, B. (eds.): Proceedings of the 6th International Workshop on Formal Aspects of Component Software, FACS@FMWeek 2009, Eindhoven, The Netherlands, November 2-3, 2009, Electronic Notes in Theoretical Computer Science, vol. 263. Elsevier (2010), https://www.sciencedirect.com/journal/electronic-notes-in-theoretical-computer-science/vol/263/suppl/C
58. Meyer, B.: Object-oriented software construction. Prentice Hall, second edition (1997)
59. Milner, R.: Communication and concurrency. Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1989)
60. Misra, J.: Orchestration. In: Fiadeiro, J.L., Liu, Z., Xue, J. (eds.) Formal Aspects of Component Software - 10th International Symposium, FACS 2013, Nanchang, China, October 27-29, 2013, Revised Selected Papers. Lecture Notes in Computer Science, vol. 8348, pp. 5–12. Springer (2013). https://doi.org/10.1007/978-3-319-07602-7_2, https://doi.org/10.1007/978-3-319-07602-7_2
61. 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, https://doi.org/10.3390/s16111810

62. Pasareanu, C.S., Salaün, G. (eds.): Formal Aspects of Component Software, 9th International Symposium, FACS 2012, Mountain View, CA, USA, September 12-14, 2012. Revised Selected Papers, Lecture Notes in Computer Science, vol. 7684. Springer (2013). https://doi.org/10.1007/978-3-642-35861-6, https://doi.org/10.1007/978-3-642-35861-6
63. Proença, J., Lumpe, M. (eds.): Formal Aspects of Component Software - 14th International Conference, FACS 2017, Braga, Portugal, October 10-13, 2017, Proceedings, Lecture Notes in Computer Science, vol. 10487. Springer (2017). https://doi.org/10.1007/978-3-319-68034-7, https://doi.org/10.1007/978-3-319-68034-7
64. Roscoe, A.W.: Theory and Practice of Concurrency. Prentice-Hall, Upper Saddle River (1997)
65. Salaün, G., Wijs, A. (eds.): Formal Aspects of Component Software - 17th International Conference, FACS 2021, Virtual Event, October 28-29, 2021, Proceedings, Lecture Notes in Computer Science, vol. 13077. Springer (2021). https://doi.org/10.1007/978-3-030-90636-8, https://doi.org/10.1007/978-3-030-90636-8
66. Szyperski, C.: Component Software: Beyond Object-Oriented Programming. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edn. (2002)
67. Tang, X., Zhang, M., Liu, W., Du, B., Liu, Z.: Towards a model of human-cyber–physical automata and a synthesis framework for control policies. Journal of Systems Architecture **144** (2023)
68. Tarifa, S.L.T., Proença, J. (eds.): Formal Aspects of Component Software - 18th International Conference, FACS 2022, Virtual Event, November 10-11, 2022, Proceedings, Lecture Notes in Computer Science, vol. 13712. Springer (2022). https://doi.org/10.1007/978-3-031-20872-0, https://doi.org/10.1007/978-3-031-20872-0
69. Wheeler, D.J.: The use of sub-routines in programmes. In: Proceedings of the 1952 ACM national meeting. p. 235. ACM, Pittsburgh, USA (1952)
70. Yang, Y., Li, X., Ke, W., Liu, Z.: Automated prototype generation from formal requirements model. IEEE Trans. Reliab. **69**(2), 632–656 (2020)
71. 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.) Theoretical Aspects of Computing - ICTAC 2022 - 19th International Colloquium, Tbilisi, Georgia, September 27-29, 2022, Proceedings. Lecture Notes in Computer Science, vol. 13572, pp. 36–41. Springer (2022)
72. Zhao, L., Liu, X., Liu, Z., Qiu, Z.: Graph transformations for object-oriented refinement. Formal Aspects of Computing **21**(1-2), 103–131 (2009)
73. 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, September 29 - Oct. 4, 2013. pp. 9:1–9:10. IEEE (2013). https://doi.org/10.1109/EMSOFT.2013.6658587, https://doi.org/10.1109/EMSOFT.2013.6658587