Towards Interface-Driven Design of Evolving Component-Based Architectures

Xin Chen and Zhiming Liu

Abstract The sustainable development of most economies and the quality of life of their citizens largely depend on the development and application of *evolutionary digital ecosystems*. The characteristic features of these systems are reflected in the so called *Internet of Things* (*IoT*), *Smart Cities* and *Cyber-Physical Systems* (*CPS*). Compared to the challenges in ICT applications that the ProCoS project used to face 25 years ago, we today deal with systems with the complexity of ever evolving architectures of networked digital components, physical components, together with sensors and devices controlled and coordinated by software. The architectural components, also called subsystems, are designed with different technologies, run on different platforms and interact through different communication technologies. However, the ProCoS project goal remains valid and the critical requirements of applications of these systems should not be compromised, and thus critical components need to be "*provably correct*". This chapter is in a form of a summary and position paper to discuss how software design for complex evolving systems can be supported by an extension of interface-driven rCOS method that we have recently been developing. We show the need for an *interface theory* to underpin development of techniques and tools. We demonstrate the need of multi-modelling notations for the description of multi-viewpoints of designs to help mastering system complexity, and their theoretical foundation in the nature of Unifying Theories of Programming proposed by Sir Professor Tony Hoare and Professor He Jifeng, as part of the outcome of the ProCoS project.

Zhiming Liu—The work is funded by the project SWU 116007, and China NSF Grant 61672435.

X. Chen
State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China
e-mail: chenxin@nju.edu.cn

Z. Liu (✉)
Centre for Software Research and Innovation, Southwest University, 2 Tiansheng Rd, Beibei, Chongqing 400715, China
e-mail: zhimingliu88@swu.edu.cn

M.G. Hinchey et al. (eds.), *Provably Correct Systems*, NASA Monographs in Systems and Software Engineering, DOI 10.1007/978-3-319-48628-4_6

1 Introduction

In the post-industry era, the challenges of the global concern of sustainable development depend on innovation application *digital ecosystems*. Such a system exists in the form of a distributed network of smart devices, program controlled physical systems (such as machines in future manufacturing factories and devices in hospitals), digital computing systems and services on the Web (or clouds). The digital components and physical objects with embedded electronics, software and sensors, which interact and collaborate through different communication networks and protocols. Such a system is open and evolving from both of

1. the key feature of the system that allows to plug-and-play new system components and services, and allows legacy components to be adapted, upgraded or replaced, and
2. the key feature of the business, social and knowledge communities it supports that are ever changing and growing.

The generally known Internet of Things (IoT) [26], Smart Cities [35] and Cyber-Physical Systems [20] are different forms of digital ecosystems. They are becoming major networks of infrastructures for development of applications in all economic and social areas such as healthcare, environment management, transport, enterprises, manufacturing, agriculture, governance, culture, societies and home automation. These applications share a common model of architectures and involve different communication technologies and protocols among the architectural components. The research and applications thus require collaborations among experts with expertise in a variety of disciplines and various skills in software systems development.

The openness of the architecture, heterogeneity of components and the scale (or complexity) of both functionality and interactions impose challenges beyond the capacity of the state of the art of software engineering. One of the most fundamental problems is that either the traditional top–down or the bottom–up development strategy, or any combination of both kinds, cannot be readily used to the development and maintenance of digital ecosystems. Therefore, there exist no methods and tools to support systematic development of digital ecosystems and their front-end applications. Ad-hoc development using tailored existing methods and tools is far from meeting the following essential requirements:

- safe and secure integration of new digital and cyber-physical components;
- maintenance and healthy evolution of legacy components and services;
- consistent adaptation of existing Internet and cloud services and applications to new and special-purpose services/devices;
- development of new applications and services from existing services/devices;
- data collection from different sources with different components, interoperably communicating among different components for processing, analytics and support of decision making.

To advance beyond the state of the art of software engineering, we need a model that captures the ever-evolving nature of the system architectures, allowing dynamically

integration and replacement of different devices, services and components. We need to develop software engineering techniques and their tool support for

1. incrementally building the model of the evolving architecture,
2. interface-based development of new components and front end applications, and their integration into an existing architecture,
3. interface-based adaption and reuse of legacy components in an existing architecture, and
4. validation and verification of components and systems by using integrated tools of simulation, testing and formal verification of trustworthiness (safety, security, privacy and dependability).

The architectural model should also support the design of fault-tolerance [10, 27, 36] with techniques of runtime monitoring and recovery [17]. Simulation with large amount of data is also needed in building models, where the data are either known or collected in the model building process, say through sensors.

In what follows, we discuss, in Sect. 2, the characteristic of complexity of digital ecosystems to clarify the challenges stated above and to give a background motivation to the interface-driven approach to health system evolution. In Sect. 3, we introduce the basics of the rCOS formal model-driven method of component and object system. We give an example in Sect. 4 to show how rCOS supports incremental and interface-driven design. In Sect. 5, we propose an extension of rCOS to modelling cyber-physical component systems.

2 Complex Evolving Systems

Software engineering was born with the aim to deal with the inherent complexity of software development, and its vision was that complexity should be mastered through the use of models, techniques and tools developed based on the types of theoretical foundations and practical disciplines that have been established in the traditional branches of engineering [28, 33]. The directions and contents of software engineering and their advances are defined and driven by the following fundamental attributes of software complexity [1–3]:

1. the complexity of the domain application,
2. the difficulty of managing the development process,
3. the flexibility possible to offer through software, and
4. the problem of characterising the behaviour of software systems.

The first attribute is the source of the challenges in software requirements *gathering, specification, analysis* and *validation*, that are the main topics of *software requirements engineering*. The second attribute, driving the development of *software project management*, concerns the difficulty to define and manage a development process to deal with complex and changing requirements of software projects that involve a

large team of software engineers and domain experts. The process has to identify the software technologies and tools that support collaboration of the team in working on shared software artifacts. The third attribute concerns the difficulties in the design of software architecture, and the design and reuse of software components, algorithms and platforms. The final attribute of software complexity pinpoints the challenges in modelling, analysis, validation and verification of the behaviour of the software.

2.1 Chronic Complexity of Digital Ecosystems

The fundamental attributes of software complexity are all reflected in software of digital ecosystems, but their extensions are becoming increasingly wider, due to the increasing power of these systems, here we quote

> "The major cause of the software crisis is that the machines have become several orders of magnitude more powerful. To put it quite bluntly: as long as there were no machines, programming was no problem at all; when we had a few weak computers, programming became a mild problem, and now we have gigantic computers, programming has become an equally gigantic problem."
>
> — Edsger Dijkstra
>
> The Humble Programmer, Communications of the ACM [9]

Now not only we have gigantic computers, but also networked computers of all scales of power from micro devices, through systems with multi-cores and multiprocessing units, to supercomputers. They execute programs anywhere and any time, which share data and communicate and collaborate with each other. These digital ecosystems are represented by the popular Internet of Things (IoT), Smart Cities, Data Centres and Cyber-Physical Systems (CPS). There exist not much agreed or clear characteristic descriptions of these systems, and a variety of viewpoints and classification exist for them. In fact, it is reasonable not to distinguish them [11, 18], especially when we are interested in system modelling, design, verification and validation. They all share the following attributes of these complex and evolving systems

1. They bring together computation, physical objects and processes, electronics, and networking communication to seamless integration of and close interaction between the physical world and computer-based systems.
2. The actions of these systems, as well as the objects, are monitored, coordinated, controlled and integrated by computing systems and existing network infrastructures.
3. These system are constantly evolving, such that new digital systems, embedded devices and physical processes keep being integrated into the system, and legacy digital systems, devices and physical processes keep being removed, modified and reconfigured.

We consider systems with the above characteristics which have *component-based* or *system of systems architectures*. Some researchers intend to distinguish

component-based systems from systems of systems and say that the latter have *emergent behaviour*. We interchange these two terms as there is no clear definition on what emergent behaviour of CPS is. Complex evolving systems exhibit the following features that are the causes of major challenges in their modelling, analysis and design:

1. Different components of these systems can have different data models, such as patients' records in healthcare systems. This feature implies the requirements of interoperable communication and information sharing.
2. Such a system has multi-stakeholders and multi-endusers who have different viewpoints of the system and whose applications use different computing, data and network and physical resources and services of the system.
3. The composition and coordination of distributed computations and services also support collaborative workflows involving multi-users.
4. Diversity of requirements of safety, security, privacy, timing, and fault-tolerance.

2.2 An Application Examples

The example as shown in Fig. 1, is a smart grid, taken from the presentation at an UK Innovate event [31]. Such a system includes smart metering and advanced metering infrastructure that provides intriguing opportunities to embrace new sustainable services for the whole energy value chain [8, 38].

A network of smart meters can also be part of the grid to provide real-time pricing for all types of users and so encourage individual consumers to reduce their power consumption at peak times. To this end, consumers can adjust their own individual

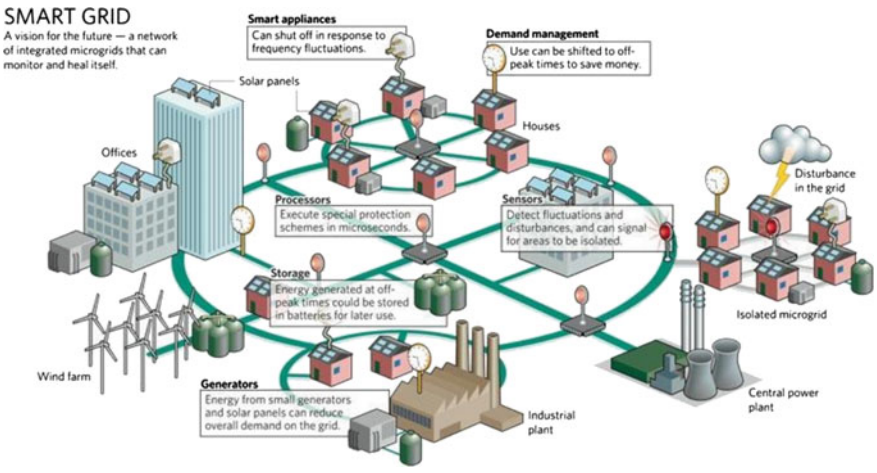


Fig. 1 Smart grid

load according to the time-differentiated prices. Furthermore, smart meters, software and communication together also enable consumers to cooperate aiming at achieving energy-aware consumption patterns, in order to realise for example, the demand-side management, demand response and Direct Load Control programmes. For illustration, imagine a smart community that autonomously adapts its energy consumption by means of enabling a limited number of household smart meters to share real-time neighbourhood information cooperatively. Users therefore cooperate with each other and with data collectors, thus facilitating the integration of energy consumption information into a common view. We will propose to develop a model of an evolving network of smart meters in Sect. 5. As in branches of transitional engineering, handling the above challenges involves the best practice of the fundamental principles of *separation of concerns*, *divide and conquer*, and *use of abstraction* through information hiding (in different design stages).

3 Interfaces and Component-Based Architectures

We now introduce the model of component interfaces that we have developed in the rCOS methods - Refinement of Component and Object Systems. The work on the rCOS framework includes formal semantics of an OO specifications, an OO refinement calculus, a unified model of component-based and OO model, that are available in a number of publications, e.g. [5, 7, 12, 13]. This chapter provides a summary and linkages among these models and theories without going into formal details. We have also published work about a UML profile for rCOS and tool support to model constructions and transformations based on the profile [21, 23, 25]. Therefore, the UML diagrams used in this chapter all have formal semantics in rCOS.

The rCOS method intends to support model-driven design (MDD) of complex evolving system. This is characterised by *letting system design be carried out in a process through building system models* to gain confidence in requirements and designs. The process of model construction in MDD emphasises on

- the use of *abstraction* for information hiding so as to be well-focused and problem oriented;
- the use of the engineering principles of *decomposition* and *separation of concerns* for *divide and conquer* and *incremental development* and evolution; and
- the use of *formalisation* to make the *process repeatable* and *artefacts (models) analysable*.

3.1 Key Features of rCOS

Main differences of the rCOS method from other model-based formal frameworks, such as Circus [4, 29], are rather in philosophic principles and intentions, instead of

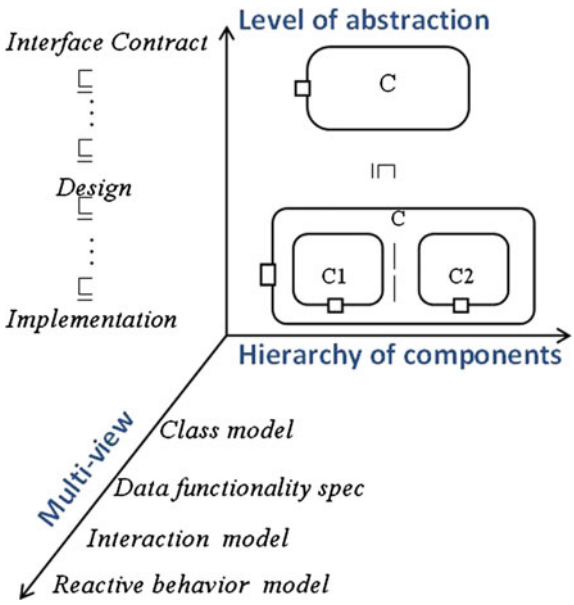


Fig. 2 rCOS modelling approach

expressive power. For example, the rCOS method makes components and interfaces as first class modelling concepts and elements, and explicitly and systematically supports separation of concerns with its multi-dimensional modelling approach to component-based architecture modelling, as shown in Fig. 2.

- First, it allows models of a component at different levels of abstraction, from the top level models of *interface contracts* of components, through models produced at different design stages including *platform independent models* (PIM) and *platform specific models* (PSM), to models of deployment and implementations.
- At each level of abstraction, a component has models of different viewpoints, including the *class model* (or *data model*), the specification of *static data functionality* (i.e. changes of data states), the *model of interaction protocol* with the environment (i.e. actors) of the components, and the *model of reactive behaviour*. These models of different viewpoints support the understanding of different aspects of the components and support different techniques of analysis, design and verification of different kinds of properties.
- A model of a component is hierarchical and composed from models of 'smaller' components that interact and collaborate with each other through their interfaces. Some components can also control, monitor or coordinate other components.

The significant advantage is that it allows the model of a component or a system at a level of abstraction is synthesised from the models of the data model, functionality and architecture, while these individual models can be refined in separation to preserve their consistency. More distinguished features of rCOS include

- direct object-oriented abstraction, instead of coding classes, objects and polymorphism in process-oriented models with unstructured states [6, 13];
- fully supported by a sound and complete object-oriented refinement calculus [37];
- direct formulation of OO design patterns as refinement rules [32, 37];
- provision of model transformations from component-based models of architecture of requirements to OO models of design architecture [6], and from OO models of design architectures to component-based models of design architectures [21];
- the provision of a well defined UML profile so that models can be constructed using the subset of UML defined by the profile and automatically translated to into rCOS models [24, 25].

The feature in the last bullet point allows us to use UML to represent models in the rest of the chapter.

3.2 Components and Their Interfaces

Components are service providers - including computing devices realising functions, processes that coordinate and control components through interactions and connectors. We intend to have different types of interfaces for different interaction mechanisms and protocols. Here, we only use a running example to show the rCOS modelling notation and method.

To ease the understanding and practice, we divide the definition of component into its syntactic description and semantic specification that we call the *contract* of the component. A (**syntactic**) **component** is represented by tuple $C = \langle X, IF, A \rangle$, where

- X is a finite set (possibly empty) of state variables.
- IF is the (**provided**) **interface** defining a finite set of operation signatures of the form $m(\overline{x}; \overline{y})$ with a finite number of input parameters and a finite number of return parameters. Each operation represents service provided to users.
- A is a finite set (possibly empty) of **internal actions**, each of which is represented as a parameterless method a. An internal action is automatous and does not have parameters.

For example, a memory can be modelled as component that provide a write operation and read operation to its user, e.g. a processor.

Component M
  $Z\ d$;
  provided interface MIF {
    $W(Z\ v)$; $R(; Z\ v)$
  }

A faulty memory can be modelled as a component below, that provide write and read operations to the user (e.g. a processor) but its content can be corrupted by an internal ‘fault action’.

component fM
$Z\ d$;
provided interface MIF {
$W(Z\ v)$; $R(;Z\ v)$;
}
actions{//fault modelling corruption
$fault$
}

The syntactic interface defines the static type of the component, but it does not specify the behaviour of the interface. The behaviour of an interface is specified by a contract. For incremental understanding, we first define a *service contract* of an interface, which specifies the state change of an execution of interface operation, provided, required or internal operations.

A **service contract** $\mathcal{C}$ of a syntactic component C specifies

- an **initial condition** defining the allowed possible initial states of the variables X by a state predicate $\mathcal{C}.init$ on X, called the **initial condition**;
- a **state transition relation** $\mathcal{C}.next$ that specifies each operation $m(\overline{x};\overline{y})$ in the provided interface IF by a pair $P \vdash R$ of a precondition P and a postondition R, where
  - P is a predicate over $X \cup \overline{x}$,
  - R is a predicate over $X \cup \overline{x} \cup X' \cup \overline{y'}$, and X' and $\overline{y'}$ are the sets of the primed version of the variables in X and $\overline{y}$.

  The meaning of $P \vdash R$ is that from a state s of X with the input parameters $\overline{x}$ satisfying precondition P, the execution of $m()$ will change the state s of X into a state s' (in which the value of x is represented by x') with the return values $\overline{y}'$ such that $((s,\overline{x}),(s',\overline{y}'))$ holds for R.
- the **state transition relation** $\mathcal{C}.iNext$ that specifies each internal operation a in A by pair $P \vdash R$ of a precondition P and a postondition R, where
  - P is a predicate over $X \cup \overline{x}$,
  - R is a predicate over $X \cup X'$.

In general, it is proven in UTP [16] that all programming statements in traditional structured programming languages can be defined by designs. In particular, an assignment $x := e$ is defined as design $\{x\} : true \vdash x' = e$, meaning that the state is changed from a state s to a new state s' in which only the value of x is changed to the evaluation of e in s, keeping other variables unchanged. The following specification combines the syntax and the service contract of a memory component offering the environment a write operation and a read operation.

Component M
$Z\ d$;
provided interface MIF {
$W(Z\ v)\{d := v\}$; $R(;Z\ v)\{v := d\}$
}

The design calculus in UTP [16] is extended to object-oriented designs in [13, 37].

A service contract only specifies the functionalities of the component in terms of a *contract* between the *assumption* on the current state and input parameters and the *guarantee* on the change of the state and return values. However, a component is in general *reactive*, thus also controls its interaction protocol with the environment and the dependency (or causality) relation between its operations. The flow of control and interaction are specified by the *guards* of the operations:

- the **guard** of an operation $m()$ in the interface IF or the international action set A is a predicate on X such that $m()$ can be executed in a state only when its guard holds in the state and the action is disabled in the state otherwise.[1]

Thus, a (guarded) contract $\mathcal{C}$ of a component actually defines a *labeled state transition system*, but the states combine both control and data together, and the labels are the interface operations and internal operations. $\mathcal{C}$ specifies each operation $m()$ by a triple of a guard g, a precondition P and a post condition R, denoted by $g\&(P \vdash R)$, called a **guarded design**. A transition from a state s to a state s' by an operation $m()$, provided, required or internal, is possible only if its guard, denoted by $\mathcal{C}.guard(m)$, holds in s. And when it is possible

- if the precondition P of $m()$ holds in s, then R holds for the pair (s, s') of states together with relation between the input and return parameters if $m()$ is a provided or required interface operation; and
- if the precondition P of $m()$ does not hold in s, s' can be any state.

When we separate the control states from the data states in the state transition system of $\mathcal{C}$, we obtain an automaton with the control states and the interface signatures as the alphabet. This allows us to use the language defined by the automaton, a regular expression when the automaton is of finite states, to express the interaction protocols.

We propose a textual specification of components in a format similar to Java, that allows us to declare multiple interfaces. In the corresponding abstract definition of components, the provided interface IF is the union of the declared interfaces. We take a few simple examples to illustrate the concepts of components. For example, the following reactive component specifies a memory that controls the order in which the write and read operations are invoked.

Component B
  $Z\ d, Bool\ w = 1;$
  provided Interface BIF {
    $W(Z\ v)\{w\&\{d, w\} : true \vdash d' = v \wedge w' = \neg w\};$
    $R(; Z\ r)\{\neg w\&\{v, w\} : true \vdash v' = d \wedge w' = \neg w\}$
  }
}

This memory also behaves like a one-place buffer.

[1] In general, the guard can contain input parameters, and even the primed version y' of return parameters y in $\overline{y}$, especially when advanced security assurance is required. We do not consider this general case as we have no semantics yet to handle them.

3.3 Composition and Orchestration

We can easily see that the one-place buffer B can be built by coordinating the uncontrolled memory M

Component B **requires** M
  $Bool\ w = 1;$
  provided Interface BIF {
    $W(Z\ v)\{w\&(M.W(z);\ w := 0)\};$
    $R(;\ Z\ r)\{\neg w\&(M.R(;\ r);\ w := 1)\}$
  }
}

We use regular expression to specify the protocol of control, obtaining the following equivalent specification

Component B **requires** M
  provided Interface BIF {
    $W(Z\ v)\{M.W(z)\};$
    $R(;\ Z\ r)\{M.R(;\ r)\}$
    protocol $\{(WR)^* + (WR^*)W\}$
  }
}

Thus, a coordination mainly changes the interaction protocol of a component, such as M, without changing the data functionality of the component. Later in Sect. 3.4, we will see a visual model the protocol can be represented as state machine diagram in the rCOS UML profile [23].

With given components, we can construct new components with connectors and through orchestration of the provided operations in the given components. For example, taking $B_i = B[W_i/W, R_i/W]$ is obtained by the connector that renames the write $W()$ and read $R()$ operations of B to $W_i()$ and read $R_i()$, respectively, for $i = 1, 2$, we can have

component M_2 **requires** B_1, B_2 {
  $Z\ y;$
  provided interface M_2IF {
    $move()\{B_1.R_1(;\ y);\ B_2.W_1(y)\};$
  }

This component provides the newly added $move()$ and the operations that B_1 and B_2 provide minus those that are called in the body of $move()$. And the protocol is defined by the guard conditions of B_1 and B_2. In general, we can extend a given set of components to form new components by defining additional provided operations using structured programming constructs. we can also use the *internalising* connector to make a provided operation, such as $move()$, internal. for example $Buff2 = M_2 \backslash move()$ behaves as

component *Buff2* **requires** B_1, B_2
  Z y;
  actions A {
    $move()\{B_1.R_1(; y); B_2.W_1(y)\}$;
  }

Component *Buff2* behaves like M_2, except for *move*() will be executed internally and autonomously when it is enabled, without the need to be called from the environment. Thus, it behaves like a two-place buffer.

Now we give an specification of the faulty memory, in which an interaction protocol is specified using an regular expression that can be coded as guards of the interface operations.

component *fM*
  Z d;
  provided interface *MIF* {
    $W(Z\ v)\ \{d := v\}$; $R(; Z\ v)\ \{v := d\}$;
  protocol $\{(WR)^* + (WR)^*W$// protocol of $C\}$
  }
  actions{//fault modelling corruption
    $fault\ \{true| - d' <> d\}$
  }

We use the renaming operators on the (provided) interface of fM and obtain three faulty memory components $fM_i \mathrel{\widehat{=}} fM[fM_i.W/W, fM_i.R/R]$, for $i = 1, 2, 3$. We now specify the following component.

component V **requires** fM_1, fM_2, fM_3 {
  provided interface *VIF*{
    $W(Z\ v)\ \{fM1.W(v); fM2.W(v); fM3.W(v)\}$;
    $R(; Z\ v)\ \{v := vote(fM1.R(v), fM2.R(v), fM3.R(v))\}$;
    protocol $\{(WR)^* + (WR)^*W\}$
  }

We can prove the proposition that *the composition of* V *is refinement of the perfect component* $B = C||M$ *if it is assumed at any time at most one of the* fM_i *is in faulty state* [27, 36]. The component-based architecture is shown in Fig. 3.

3.4 Separation of Concerns

When the data model for the variables, interface interaction protocols and the dynamic behaviour of component become complex, models of different viewpoints for different design concerns are needed. To this end, we have a UML profile for rCOS [24]. This allows that for object-oriented design of component-based modelling and design of finite state components, we use

- UML *class models* for the representation of the data models at different levels of abstraction, specially *conceptual class model* for requirements and *design class models* for object-oriented design of components;

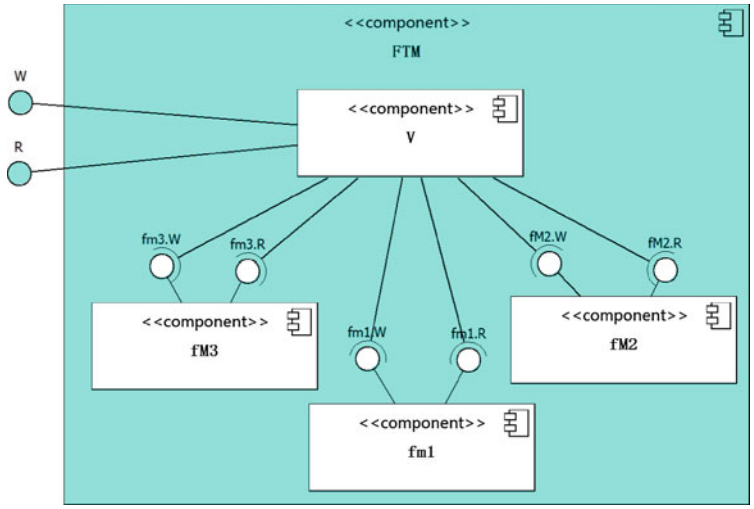


Fig. 3 Component-based architecture of a fault-tolerant memory

- (extended) UML *sequence diagrams* for modelling interactions among components and between components and actors (component sequence diagrams), and for interaction among objects of a design of a component (*object sequence diagrams*); and
- (extended) UML state machine diagrams for modelling the dynamic behaviour of a component.

The extended sequence diagrams, together with the textual specification of pre- and post-conditions of the methods, generate the rCOS functionality definitions of the participating components, such as V, and the state diagrams of the components define the protocols that are corresponding to the guards of the methods in the components. Thus, the contracts of the interfaces can be divided into the **contracts of static functionality** and the **contracts of dynamic behaviour**. The former are given by the unguarded design of interface operations that are specified only by their pre- and post-conditions, and the latter by the state machine diagram of the components. With a UML profile defined for rCOS, these models of different views points can be automatically integrated into rCOS textual specification [23, 25].

The sequence diagrams and state machine diagrams of different viewpoints of f the fault-tolerant memory are shown in Fig. 4, and we will discuss more examples in the next section.

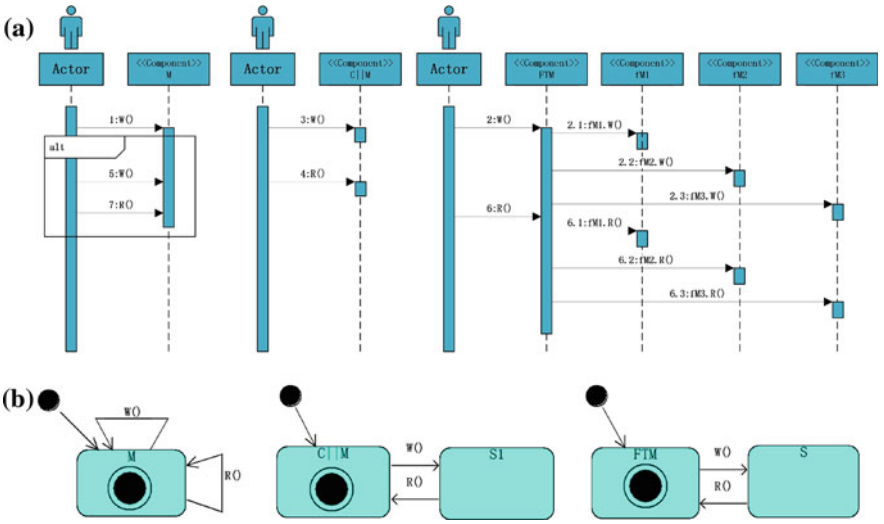


Fig. 4 UML models of interactions and dynamic behaviour

4 Incremental Design of an Enterprise Application

Incremental/evolutionary modelling and design has been practised in empirical and ad hoc software development. This section, however, demonstrates how rCOS supports an incremental/evolutionary modelling and design of the case study of a computerised trading system of an enterprise of supermarkets. It was used as the Common Component Modelling Example (CoCoME) [6, 14]. It is an extension of the Point of Sale (POST) example used in Larman's textbook [19]. The case study was described in terms of the use cases related to *process sales*, *manage inventory*, *prepare for product orders*, *process deliveries of ordered products*, and *exchange products among different stores*, etc.

The evolutionary nature of the system is determined by the development of the enterprise. The business may just start from a single store and the store requires a computerised system to improve the automation of the use case process sales to speed up customer checkout and record the sales. Also, at the early stage of the business, only one checkout "cash desk" is enough, or the system development can start with considering only one checkout cash desk.

4.1 Requirements Modelling

The requirements gathering and analysis starts from describing use cases, and any described use case explicitly or implicitly implies restrictions on the functionality

either due to the stage of the business development or consideration for a simplification to start with. For example, we start with the use case *process sale with cash payment* briefly described below.

Overview: A customer arrives at the Cash Desk with the product items to purchase with cash payment. The sale and the payment are recorded in the system. Involved Actors includes Customer and Cashier.

Process: The normal courses of interactions between the actors and the system are described as follows.

1. When a Customer comes to the Cash Desk with her items, the Cashier initiates a new sale. The system creates a new sale.
2. The Cashier enters each item, either by scanning in the bar code or by some other means; if there is more than one of the same item, the Cashier can enter the quantity. The system records each item and its quantity and calculates the subtotal.
3. When there are no more items, the Cashier indicates to the system the end of entry. The total of the sale is calculated. The Cashier tells the Customer the total and asks her to pay.
4. The Customer gives the Casher cash and the Cashier enters the amount received. The system records the cash payment amount and calculates the change. Then the completed sale is logged.

Alternative courses of events: There are exceptional or alternative courses of interactions, e.g., if the entered bar code is not known in the system, the Customer does not have enough money for a cash payment. A system needs to provide means of handling these exceptional cases, such as cancel the sale.

At the requirements stage, *we model a use case as a component* by a **conceptual class model**, a **component sequence diagram**, **state machine diagram**, and the **contract of static functionality** of the interface operations. For the use case process sale with cash payment, we have the class model in Fig. 5, sequence diagram in Fig. 6a, and state machine diagram in Fig. 6b.

The operations that actor Cashier calls in Fig. 6a form the provided interface of component *ProcessSale*, and the state machine diagram in Fig. 6b defines its contracts of dynamic behaviour. Their consistency can be checked by FDR [34] after being translated into processes of CSP [15, 34]. The contract of static functionality of *ProcessSale* is specified by the pre- and post-conditions of the interface operations.

The precondition of *startSale*() requires the existence of the *Store*, the *CashDesk*, the *Catalog* and the *Product Specifications*. The postcondition of *startSale*() is to create a new sale. Thus, the state variables of *ProcessSale* include *Store store*, *CashDesk cashdesk*, *Catalog cat*, and *Sale sale*. The contract of *startSale*() can be specified as

$$\{store \neq nil \wedge cashdesk \neq nil \wedge cat \neq nil\}\ startSale()\ \{sale' = new\ Sale\}$$

Similarly, we can specify the contracts of the other operations. For example,

$$\left\{\begin{array}{l}\wedge store \neq nil\\ \wedge cashdesk \neq nil\\ \wedge cat \neq nil\\ \wedge sale.isComplete\end{array}\right\} makeCashPayment(a) \left\{\begin{array}{l}\wedge cashPay'{=}new\ CashPayment\\ \wedge cashPay'.amount = a\\ \wedge Is\text{-}Pay\text{-}by(sale, cashPay')\end{array}\right\}$$

The semantics of OO contracts of operations are derived from OO designs in [13].

4.2 OO Design of Components

In practical but informal OO development, the design stage is to decompose the functionality and responsibility of each interface operation (informally) described by it pre- and post-conditions and assign the sub-responsibilities to "appropriate" objects of the component. The decomposition and assignment of the responsibilities are carried out using GRASP design patterns [19]. These patterns are proven to be rCOS refinement rules [13, 37]. Therefore, the following design steps can actually be formally justified in rCOS.

For a requirements model of a component, such as that of *ProcessSale* given in the previous subsection, we design each interface operation according to its contract. This is done by using the formalised GRASP design patterns and refactoring rules that are formally proven in the OO refinement calculus [37]. In particular, by Controller pattern, we can decide to implement the provided interface of *ProcessSale* by class

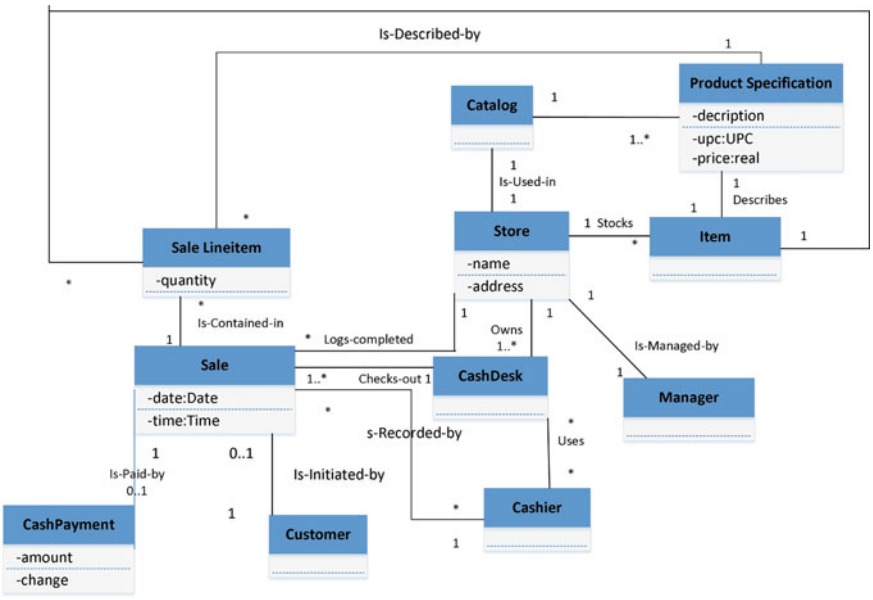


Fig. 5 Conceptual class diagram of *ProcessSale*

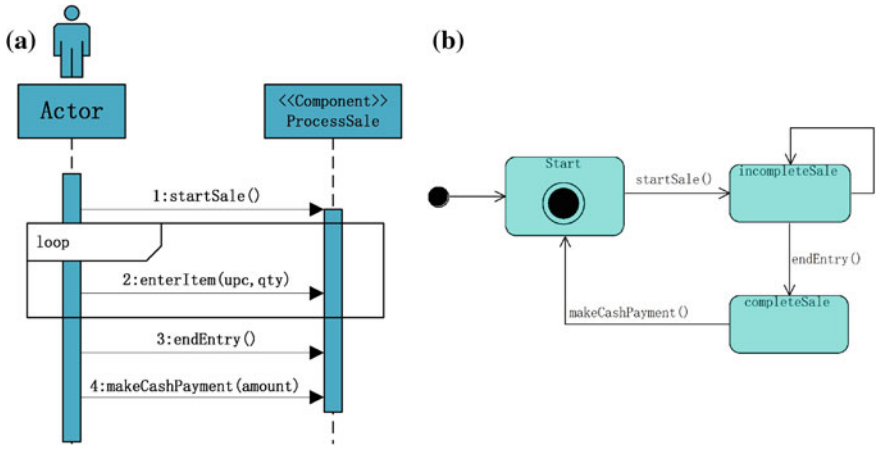


Fig. 6 Sequence diagram and state machine diagram of *ProcessSale*

$CashDesk$, and the design of each operation is represented by an **object sequence diagram**. For example, the design of $makeCashPayment()$ is given in Fig. 7.

With the model transformation tool of rCOS [22], we can check that the objects : *CashDesk*, *sale: Sale* and *:CashPayment* form a component $HandleSale$ with the interface object: *CashDesk*; and the objects *:Store* and the container object $\langle\langle Set\rangle\rangle Sale$ form another component *StoreManagement* with the interface object : *Store*. The tool then automatically transforms the OO design in Fig. 7 to a component-based design in Fig. 8b.

The design proceeds with OO design of the other provided interface operations of *ProcessSale*, followed by decomposition into provided interface operations, *startSale*(), $enterItem(upc, qty)$, and *endEntry*() of *HandleSale*, and required interface operations of *HandleSale* for *checking* the validity of *upc*, and *extracting* the product specification from the *Catalog* object continued in the *Store* object. Therefore, the *upc* checking operation $check(upc)$ and specification *extracting* operation $find(upc; spec)$ are provided interface *ManageStore*. We then obtain a component-based decomposition of

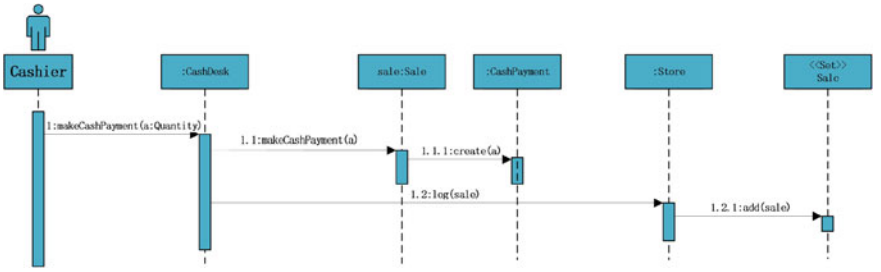


Fig. 7 OO design of $makeCashPayment()$

component *ProcessSale* shown in Fig. 8a. The rCOS transformation tool [22] also automatically generates the *static component diagrams* shown in Fig. 9 corresponding to the transformation from the OO design in Fig. 7 to the component-based design in Fig. 8.

4.3 Incremental Development and System Evolution

Component *ProcessSale* designed in the previous subsection assumes some restrictions on the functionality. For example, among other restrictions, it deals cash payment only and has no inventory update when the completed sale is logged. In general, in each cycle of Rational Unified Development Process, components and their individual operations are designed for restricted functionalities. Further development is to relax the assumptions to extend their functionalities, and to design new components. The rCOS method also put such incremental and evolutionary design into its formal refinement calculus so as to ensure rigorous correctness. We informally show such incremental design by singling out the process of handling cash payment as a use case by itself, denoted by *HandleCashPayment*.

We take the operations represented by messages 1–3 in Fig. 6a to form a component, denoted by *HandleSale*. Component *HandleCashPayment* itself can be designed as a component with the provided interface operation *makeCashPayment*(). Its OO design is the same as that in Fig. 7, and the component-based decomposition is the same as that in Fig. 8b, but with a new component name *HandleCashPayment*. In a new development cycle, we can follow the same way in which component *HandleCashPayment* is modelled to design a model of component *HandleCreditPayment*. It provides an operation *makeCreditPayment*(). Before a *CreditPayment* is created, *HandleCreditPayment*

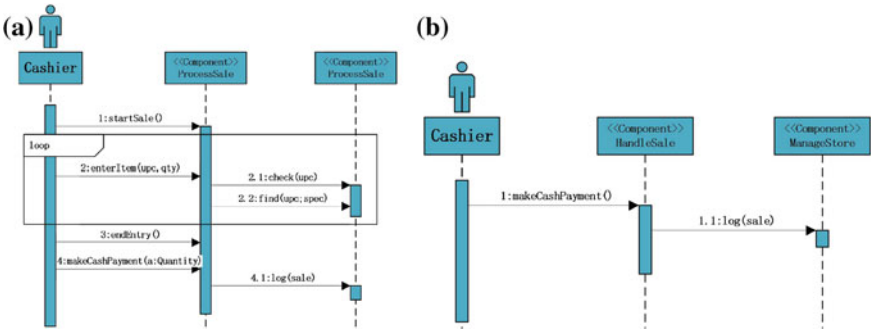


Fig. 8 Component sequence diagram of component *ProcessSale*

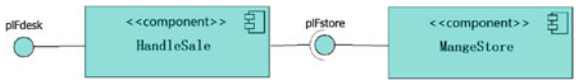


Fig. 9 A component diagram

calls the service from actor *Bank* for the authorisation of the credit payment. Therefore, *HandleCashPayment* requires to call an operation of the Bank, that we denote by *authoriseCredit*(*cardInfo*, *amount*). After authorisation, the *CreditPayment* is created, and the completed sale is logged to *Store*. In the same way, we design a component *HandleCheckPayment*.

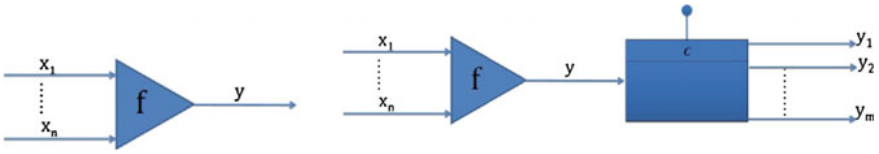
Assume that a system that only supports process sale with cash payment is already developed. In its system evolution, a new component *HandleCreditPayment* can be specified through investigation of the original architecture that consists *HandleSale* and *ManageStore*. This new component can then be designed and integrated into the legacy architecture to support processing credit payment.

With the architecture models in Figs. 8 and 9, we can extend the provided interface of *ManageStore* with more product management operations, such as those for changing the price of a product, increasing and deducting the inventory of a product (after more items are ordered and sold). We can then upgrade component *HandleProcessSale* so that after the complete sale is logged to the *Store*, the product items of the sale are removed from stock using the inventory deduction operation, say *deInventory*(upc, qty). This can be realised by *aspect oriented* design and the interface operation *makeCashPayment* (and *makeCreditPayment*()) first executes its original body and then calls the method *decInventory*(cpu, qty) of *ManageStore* repeatedly for each item in the sale. This is an "*after*" advice in aspect oriented design. An aspect oriented architecture modification like this is modelled as a connector component that changes the original component by modifying the execution of the interface operation according to the advices in the aspect.

Further system evolution can go from one checkout cash desk to a number of them in a store, from an one-store business to an enterprise of a store chain. Also, further extension to the system can be developed to support online shopping. The model of component-based architecture and interfaces contracts are also imported for analysis of safety, security and performance vulnerabilities and deficiencies so that architecture modifications and changes of interaction protocols can be designed to improve the safety, security and performance.

5 Towards Modelling Cyber-Physical Component Systems

The components in the previous section are digital components. We now propose to extend the models to *physical interfaces* and *cyber-physical components*, using the evolutionary development of a smart meter network demand response (DR) programme [30].



(a) Physical interface (b) A Cyber-physical component

Fig. 10 Cyber-physical component

5.1 Physical Interfaces and Cyber-Physical Components

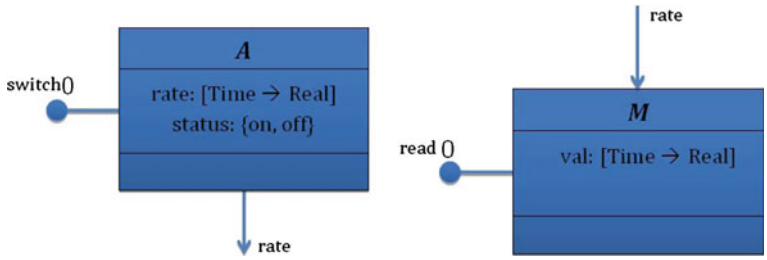
We extend the model of components with variables, called *physical variables*, whose behaviour are functions from time to real number, depending on conditions of digital states. The trajectories of the physical variables are specified by differential equations. For example, the rate of electricity consumption of an electrical appliance are different when the appliance is in different states, say when it is "on", "off", or in the "energy-saving" state. We model a *physical interface* as function $f(x_1, \ldots, x_n; y)$ with one or more *incoming signals* $x_1, \ldots, x_n$ that are continuous variables, and one[2] *outgoing signal* y, as shown in Fig. 10a. The incoming signals of an interface are also called *requiring signals*. A component also provides (or outputs) signals to the environment, such as $y_1, \ldots, y_n$ in cyber-physical component shown in Fig. 10b. There, the function f defined in the component is part of hybrid behaviour of the component, and the solid circle represents the provided *digital* (or cyber) interface. The definition of operations in the provided cyber operations may rely on operations to be provided by other operations, called the *required cyber interface* and represented by the half circle in the diagram. The composition of components is also extended by linking provided signals of a component to incoming signals of interfaces of another component.

5.2 Model the Evolution of a Smart Meter Network

The system in this case study consists of three kinds of components.

- **Consumer**: is a household equipped with one or more smart meters that is connected to the power line, electrical appliances, and to a communication network.
- **Data Collector**: is in charge of the data aggregation process. According to the resource allocation algorithm, this process is modelled as a centralised coordinator, but a distributed approach can be implemented securely.
- **Utility**: is a set of energy suppliers shared by customers. We assume utilities to implement distributed generation

[2] In general, there can be more than one.



(a) Model of an appliance (b) Model of a meter

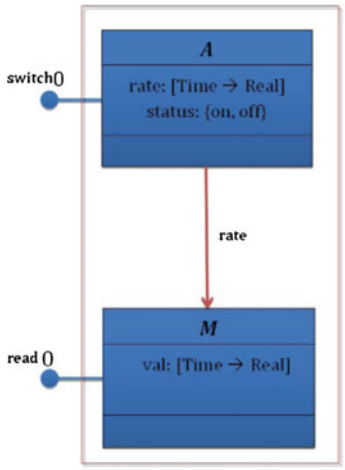
Fig. 11 Appliance and meter

We mainly demonstrate the evolutionary nature of the system and show how our modelling approach scales up. We first consider a single appliance A of a single household. An appliance, as shown in Fig. 11a, has a digital state *Status* which takes a value *on* or *off*, and it is changed by the *digital interface* operation *switch*(). The appliance has an observable signal *rate* representing the electricity consumption rate. It is a function from "Time" to real numbers, that (presumedly) can be obtained from manufacturer of the appliance. The signal "rate" is useless if the householder only observes the "rate" and switches on the appliance when needed.

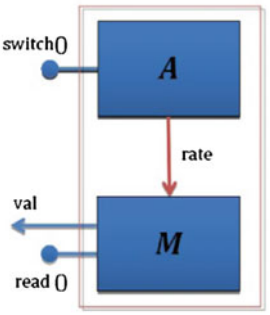
If the householder wants to know better about his daily use of electricity and to plan his use of the appliance in order to reduce their electricity bill, an electronic meter M can be introduced as shown in Fig. 11b. Meter M records the accumulated consumption of energy of an appliance A. Its provided interface $M.pIF$ provides a digital operation *read*() and its required interface $M.rIF$ consists of a single signal *rate*. The interface behaviour of M (i.e., the return of *read*()) is a discretised value of the internal signal *val* that is a timed function dependent on the required signal rate. For example, it can be defined as $val(t) = \int_0^t rate\,dx$. In general, the the trajectories of the continuous variables of a component C are specified as timed functions of the form $\gamma C = F(\beta C, \gamma C, rW)$, where feedbacks loops are possible. If we compose the appliance A and the meter M, we have the component shown in Fig. 12a.

There are alternative models. For example, a meter can include a sensor that observes the *rate*. Then *val* would be discretised and represented as a step function. Then *read*() directly returns the value of the internal discrete variable *val*. In this model, the sensor is actually represented as part of the physical interface. Also, a meter can be modelled as a component with a required signal rate and a provided digital operation *val*() to the meter component. The advantage of the component-based modelling with explicit interface contracts is exactly to allow different models and support comparative analysis.

At this stage of system evolution, the *read*() and *switch*() are still only manually operated by the householder. A further desire of home automation is to introduce a control component P, called a *control pad*. For accuracy and fault-tolerance, we make the internal signal *val* of $A \| M$ external, and denote this cyber-physical component as



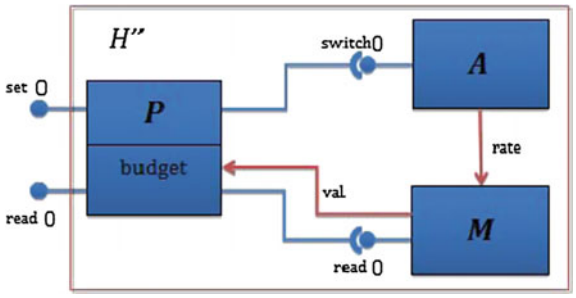
(a) Composition $A \| M$ of an appliance and a meter



(b) A cyber-physical model of the composition MA

Fig. 12 Models of composition of an appliance and a meter

Fig. 13 Automatically controlled appliance



MA, as shown in Fig. 12b. We now compose the control pad P and component *MA*, and it is shown in Fig. 13. Now the householder can use *set* and *read*() to program daily use of the appliance, according to a daily budget.

Home automation The evolution continues and a household can have a number of appliances. Then more meters or a meter with an open number of required input signals can be used for the design of a control pad. The overall control pad can either be designed using the existing individual control pads or the individual control pads are replaced with a centralised control pad. In either case, the design models of the individual control pads can be reused. The advantage of the proposed framework is that a household with a number of appliance can be treated in the same ways as if the household has a single appliance.

$$A_i \mathrel{\widehat{=}} A[switch_i/switch, rate_i/rate], \qquad A \mathrel{\widehat{=}} A_1 \parallel \ldots \parallel A_n$$
$$M_i \mathrel{\widehat{=}} A[read_i/read, val_i/val], \qquad M \mathrel{\widehat{=}} M_1 \parallel \ldots \parallel M_n$$
$$P_i \mathrel{\widehat{=}} [set_i/set, read_i/read, val_i/val, switch_i/switch], \quad P \mathrel{\widehat{=}} P_1 \parallel \ldots \parallel P_n$$

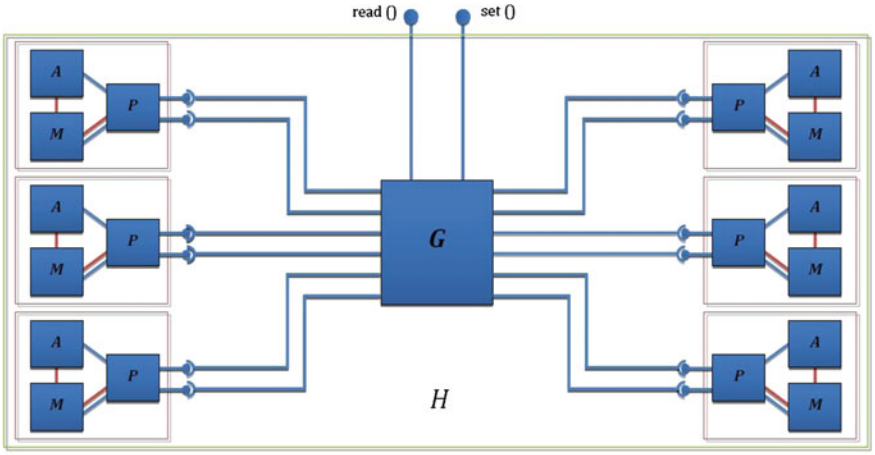
Here, renaming of interface operations and signals are used. We add a global controller P for planning and schedule of a household, and thus obtain an automated household $H \mathrel{\widehat{=}} G \| P \| M \| A$. This is shown in Fig. 14a. This system is closed inside the house and thus there are no security threats to it (unless a burglary happens). However, a further step of evolution can introduce a controller operated through a mobile phone, as shown in Fig. 14. We denote this automated home by *MH*. Then, an open mobile phone communication network is used, and security threats are introduced too. Therefore, interface-driven component-based architectures are essential to identify system safety vulnerabilities, security threats, and performance deficiencies, so as to make architecture modifications to enhance safety, security, availability and fault-tolerance.

Network evolution The designs of a household can be abstractly described as follows.

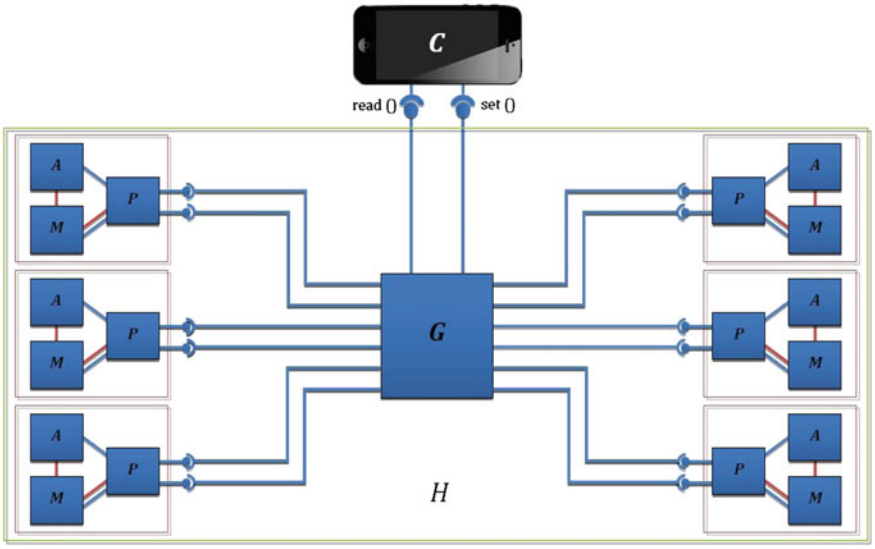
```
Component H {
    attributes: fD, vD: Real;//fixed and variable
                            //energy demands of the community
    signal: val: Real;
    provided interface:
         Rf(;x:Real), Rv(;y:Real);
         Wf(x:Real), Wv(y:Real);
         setUp() /** set up budget and policy /** by householder;
         val/ ** provided signal

     Functionality:
     Rf(;x){x:=fD}; Rv(; y){y:=vD};
     Wf(x){fD:=x}; Wv(y){vD:=y}
     ----
   }
```

We define a network of households $H \mathrel{\widehat{=}} H_1 \parallel \ldots \parallel H_k$ for a community of residence. Assume the component *Utility* provides a operations *requestF*(*x:Real*; *u: Real*) and *requestV*(*y:Real*; *v: Real*) for supply of fixed energy and variable energy, respectively. When it is called, the method returns the amount of committed supply for the day through the return parameter. Consider a *Coordnator* component which periodically calls the interface operations $Rf_i()$ and $Rv_i()$ and makes a request to *Utility* through *request*(). After it receives notification from *Utility* about the committed supply, it "negotiates" with the households (through communication interfaces that we omit in this chapter) and reallocates budgets to the households through $Wf_i()$ and $Wv_i()$. This gives a network system $H \| Coordinator \| Utility$, as shown in Fig. 15.

Except for the "negotiation" of the *Coordinator* with individual households, the composition H of the households behaves exactly the same as one household. Similarly, we can imagine that a network of utilities works in collaboration to provide a power supply. Once they reach an agreement among themselves on how they share the supply to the request from the collector, they interface with the collector in the



(a) Automated household



(b) A mobile controlled home

Fig. 14 Home automation

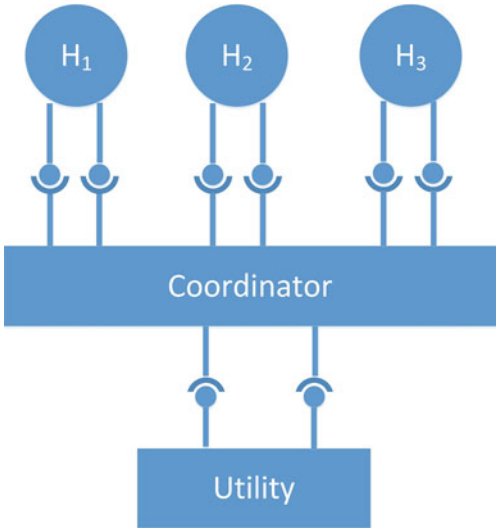


Fig. 15 A smart grid

same manner as a single utility. Furthermore, the centralised collector can be transformed into a distributed implementation so that the "negotiation" can be performed among households themselves.

6 Conclusions

This chapter has argued the importance of component-based (or system of systems) architectures and contracts of interfaces for healthy evolution of digital ecosystems. We proposed an extension to the rCOS model of digital components and interfaces to cyber-physical components. This makes the notion of interfaces very general. For example, a piece of wall or a window can be modelled interfaces between the temperatures outside and inside a room. Even the "air" between two sections of a room can modelled as an interface that transforms the temperature of one section to that of another. However, this general notion of interfaces poses a number of challenges, for example

1. How to develop a model of contracts of such interfaces, as it is often the case that there is no known physical laws or functions for defining these interfaces?
2. How to define the formal semantics and the refinement relation between cyber-physical interface contracts?

These are the first significant questions to ask when developing a semantic theory for these CPS components and their compositions. Further challenges include

1. how to develop design techniques and tools,

2. how to combine David Parnas's Four-Variable Model, Michael Jackson's Problem Frames Model, and the Rational Unified Process (RUP) of the use case driven approach systematically into the continuous evolutionary integration system development process?

We believe that our model-driven approach is again promising, and techniques and tools of simulation with rich data and machine learning would become increasingly important in building the correct models.

Acknowledgements We acknowledge the contribution to the development of the rCOS method from Zhenbang Chen, Ruzheng Dong, He Jifeng, Wei Ke, Dan Li, Xiaoshan Li, Jing Liu, Charles Morisset, Anders Ravn, Volker Stolz, Shuling Wang, Jing Yang, Liang Zhao, and Naijun Zhan. We also thank Jonathan Bowen, Xiaohong Chen, Sabita Maharjan, Esther Palomar and Yan Zhang for the collaboration on Component-Based Modelling for Sustainable and Scalable Smart Meter Networks [30].

References

1. Booch, G.: Object-Oriented Analysis and Design with Applications. Addison-Wesley, Boston (1994)
2. Brooks, F.P.: No silver bullet: essence and accidents of software engineering. IEEE Comput. **20**(4), 10–19 (1987)
3. Brooks, F.P.: The mythical man-month: after 20 years. IEEE Softw. **12**(5), 57–60 (1995)
4. Cavalcanti, A., Sampaio, A., Woodcock, J.: A refinement strategy for circus. Form. Asp. Comput. **15**(2–3), 146–181 (2003). http://dx.doi.org/10.1007/s00165-003-0006-5
5. Chen, X., He, J., Liu, Z., Zhan, N.: A model of component-based programming. In: Arbab, F., Sirjani, M. (eds.) International Symposium on Fundamentals of Software Engineering. Lecture Notes in Computer Science, vol. 4767, pp. 191–206. Springer, Berlin (2007)
6. Chen, Z., Hannousse, A.H., Hung, D.V., Knoll, I., Li, X., Liu, Y., Liu, Z., Nan, Q., Okika, J.C., Ravn, A.P., Stolz, V., Yang, L., Zhan, N.: Modelling with relational calculus of object and component systems–rCOS. In: Rausch, A., Reussner, R., Mirandola, R., Plasil, F. (eds.) The Common Component Modeling Example. Lecture Notes in Computer Science, chap. 3, vol. 5153, pp. 116–145. Springer, Berlin (2008)
7. 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). Feb
8. Darby, S.: Smart metering: what potential for householder engagement? Build. Res. Inf. **38**(5), 442–457 (2010)
9. Dijkstra, E.W.: The humble programmer. Commun. ACM **15**(10), 859–866 (1972). An ACM Turing Award lecture
10. Fischer, C.: Fault-tolerant programming by transformations. Ph.D. thesis, University of Warwick (1991)
11. Gunes, V., Peter, S., Givargis, T., Vahid, F.: A survey on concepts, applications, and challenges in cyber-physical systems. Trans. Internet Inf. Syst. **8**(12), 4242–4268 (2014)
12. He, J., Li, X., Liu, Z.: A theory of reactive components. Electr. Notes Theor. Comput. Sci. **160**, 173–195 (2006)
13. He, J., Liu, Z., Li, X.: rCOS: a refinement calculus of object systems. Theor. Comput. Sci. **365**(1–2), 109–142 (2006)
14. 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. Lecture Notes in Computer Science, chap. 1, vol. 5153, pp. 16–53. Springer, Berlin (2008)
15. Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall, Upper Saddle River (1985)
16. Hoare, A., He, J.: Unifying Theories of Programming. Prentice Hall, New York (1988)
17. Kim, M., Viswanathan, M., Lee, I., Ben-Abdellah, H., Kannan, S., Sokolsky, O.: Formally specified monitoring of temporal properties. In: Proceedings of the European Conference on Real-Time Systems (1999)
18. Koubaa, A., Andersson, B.: A vision of cyber-physical internet. In: Proceedings of the Workshop of Real-Time Networks (RTN 2009), Satellite Workshop of ECRTS 2009 (2009)
19. 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)
20. Lee, E.: Cyber physical systems: design challenges. Technical Report No. UCB/EECS-2008-8, University of California, Berkeley (2008)
21. 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). http://dx.doi.org/10.1007/978-3-642-35743-5_7
22. Li, D., Li, X., Liu, Z., Stolz, V.: Interactive transformations from object-oriented models to component-based models. In: 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)
23. Li, D., Li, X., Liu, Z., Stolz, V.: Support formal component-based development with UML profile. In: 22nd Australian Conference on Software Engineering (ASWEC 2013), 4–7 June 2013, Melbourne, Victoria, Australia. pp. 191–200 (2013)
24. Li, D., Li, X., Liu, Z., Stolz, V.: Support formal component-based development with UML profile. In: 22nd Australian Conference on Software Engineering (ASWEC 2013), 4–7 June 2013, Melbourne, Victoria, Australia. pp. 191–200. IEEE Computer Society (2013). http://dx.doi.org/10.1109/ASWEC.2013.31
25. Li, D., Li, X., Liu, Z., Stolz, V.: Automated transformations from UML behavior models to contracts. SCI. CHINA Inf. Sci. **57**(12), 1–17 (2014). http://dx.doi.org/10.1007/s11432-014-5159-8
26. Li, X., Lu, R., Liang, X., Shen, X., Chen, J., Lin, X.: Smart community: an internet of things application. Commun. Mag. **49**(11), 68–75 (2011)
27. Liu, Z., Joseph, M.: Specification and verification of fault-tolerance, timing, and scheduling. ACM Trans. Program. Lang. Syst. **21**(1), 46–89 (1999)
28. Naur, P., Randell, B. (eds.): Software Engineering: Report of a Conference Sponsored by the NATO Science Committee, Garmisch, Germany, 7–11 Oct. 1968, Brussels, Scientific Affairs Division, NATO (1969)
29. Oliveira, M., Cavalcanti, A., Woodcock, J.: Formal development of industrial-scale systems in *Circus*. ISSE **1**(2), 125–146 (2005). http://dx.doi.org/10.1007/s11334-005-0014-0
30. Palomar, E., Liu, Z., Bowen, J.P., Zhang, Y., Maharjan, S.: Component-based modelling for sustainable and scalable smart meter networks. In: Proceeding of IEEE International Symposium on a World of Wireless, Mobile and Multimedia Networks, WoWMoM 2014, Sydney, Australia, June 19, 2014. pp. 1–6 (2014)
31. Pronios, N.B.: Software verification & validation for complex systems, presentation at Technical Feasibility Studies Competition Information Event, Innovate UK
32. Quan, L., Qiu, Z., Liu, Z.: Formal use of design patterns and refactoring. 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. 323–338. Springer (2008). http://dx.doi.org/10.1007/978-3-540-88479-8_23

33. Randell, B., Buxton, J. (eds.): Software Engineering: Report of a Conference Sponsored by the NATO Science Committee, Rome, Italy, 27–31 Oct. 1969, Brussels, Scientific Affairs Division, NATO (1969)
34. Roscoe, A.W.: Theory and Practice of Concurrency. Prentice-Hall, Upper Saddle River (1997)
35. Shapiro, M.: Smart cities: quality of life, productivity, and the growth effects of human capital. Rev. Econ. Stat. **88**, 324–335 (2006). May
36. Zhang, M., Liu, Z., Morisset, C., Ravn, A.P.: Design and verification of fault-tolerant components. In: Butler, M., Jones, C., Romanovsky, A., Troubitsyna, E. (eds.) Methods, Models and Tools for Fault Tolerance. Lecture Notes in Computer Science, vol. 5454, pp. 57–84. Springer, Berlin (2009)
37. Zhao, L., Liu, X., Liu, Z., Qiu, Z.: Graph transformations for object-oriented refinement. Formal Aspects Comput. **21**(1–2), 103–131 (2009). Feb
38. Zhu, J., Pecen, R.: A novel automatic utility data collection system using ieee 802.15.4-compliant wireless mesh networks. In: Proceedings of IAJCIJME International Conference (2008)