Software Abstractions and Human-Cyber-Physical Systems Architecture Modelling

Zhiming Liu[1], Jonathan P. Bowen[1,2(✉)], Bo Liu[1], Shmuel Tyszberowicz[1,3], and Tingting Zhang[1]

[1] RISE – Centre for Research and Innovation in Software Engineering, Southwest University, Chongqing, China
{zhimingliu88,liubocq}@swu.edu.cn
[2] School of Engineering, London South Bank University, London, UK
jonathan.bowen@lsbu.ac.uk
[3] Afeka Academic College of Engineering, Tel Aviv, Israel
tyshbe@tau.ac.il

Abstract. It is over fifty years since the subject discipline of *software engineering* and more than forty years from when the area of *formal methods* have been established. During this period, the academic community has accomplished extensive research in foundations and methods of software engineering, as well as developing and teaching a large body of software engineering knowledge and techniques. At the same time, the IT industry has produced larger, more complex, and better (in many aspects) software systems. Yet, these large projects are largely developed using a trial and error approach, without systematic use of the developed software engineering methods and tools. The cost of these projects is high, the percentage of project delay and cancellation is significant, and the dependability of the systems is low in many requirements. The most serious problem of this ad hoc development approach is that the development process is not repeatable and the systems developed are not well evolvable. This problem is particularly crucial for the design and implementation of modern networked distributed software systems, known as *Human-Cyber-Physical Systems* (HCPS).

In this tutorial paper, we reflect the development of software engineering through software abstractions and show that these abstractions are integral in the notion of software system architectures. We discuss the importance of architecture modelling and argue for a seamless combination of informal and formal activities in the modelling and design of the architecture. A point that we make is that it is important to engineer systems using formal methods in relation to the definition and management of development processes, and how a model of the software architecture, with rich semantics and refinement relations, plays an important role in this process. We consider development of two typical types of software components and use examples to discuss the traditional processes for their *domain modelling* and *software requirements modelling*. We then

J. P. Bowen et al. (Eds.): SETSS 2019, LNCS 12154, pp. 159–219, 2020.
https://doi.org/10.1007/978-3-030-55089-9_5

propose to combine these modelling approaches and this naturally leads to a *unified modelling process* for HCPS architecture modelling, design, and evolution. Based on the unified processes, we outline a framework in engineering formal methods for HCPS modelling, including the mapping of the system architecture to the technology architecture and organization of the development team with the expertise required, and decide the appropriate formal methods and tools to be used.

Keywords: Formal methods · Human-Computer-Physical System · Abstraction · Architecture modelling · Conceptual integrity · System evolution

1 Introduction

Although the term was used earlier, the notion of "Software Engineering" with its *intention* and *extension*, was first proposed at the world's first conference on software engineering that was held in 1968, sponsored and facilitated by NATO [77]. The challenges which it intended to address were characterized by "*software crisis*". Symptoms of the crisis had been that large software projects resulted in a high percentage of cancellations, late deliveries, over-spending of the budget, and systems that often failed to meet critical requirements. The cause of the crisis was regarded to be due to the growing power of computing machines [30], and as a consequence increase in software demand, complexity, and challenges, yet without changing the methods and tools being used.

From the viewpoint with respect to software construction, however, the fundamental problem of the software crisis was that it lacked systematic engineering methods for the production of computer programs. Unlike well-established branches of engineering, where the construction of a complex product was undertaken using well defined *engineering processes* employing proper *methods* and *tools*, software production was carried in an ad hoc manner, without the systematic use of standard methods and tools. Here, by methods and tools, we mean approaches that have been developed based on scientific and sound mathematical theories for modelling, validation, and verification. An improvised development process with best-effort methods is more likely to fail and is not repeatable. Neither the process nor the artefacts can be validated and the final product is not maintainable. This is why in the days of the software crisis, "various large software projects were almost all one-off projects, developed for specific customers, and all too many of the largest projects were characterized by underestimates and overexpectations" as Randell recalls [90].

The participants of NATO's Conference set the intent of software engineering to provide the construction of software systems with a sound mathematical foundation and based on this foundation to develop *systematic methods*, *tools*, and *standards* for software *design*, *analysis*, *implementation*, *verification*, and *validation*—like the traditional branches of engineering disciplines [77].

The report on the NATO conference demonstrates that it was very stimulating and history also proves it created great impact. Brian Randell pointed out in

his keynote presentation at the 2018 International Conference on Software Engineering (ICSE) [90], among other software engineering subject areas, some people who attended NATO's conference were motivated by the problem of how to create programs that were mathematically-proven to be free from errors, becoming pioneers of *formal methods* (e.g., Edsger Dijkstra [29]), and some started the area of *fault-tolerant programming*, concerning how to design programs that could be usefully relied upon even if it was admitted that they still contained yet-to-be found bugs (e.g., Randell himself [88]). The first author of the present paper has worked on formal techniques in fault-tolerant programming [62,67].

In the half-century since the NATO conference, as well as these aspects of software engineering, intensive research has been conducted on "all aspects of software production from the early stages of system specification through to maintaining the system after it has gone into use" [99]. A rich body of knowledge, principles, and methods, including development processes, architecture styles, design patterns, programming paradigms, as well as techniques and tools for verification, validation, program construction, synthesis, etc., have been proposed and developed [45]. Research on formal methods [10] has also advanced with the development of comprehensive theories of formal semantics of languages, specification and verification techniques, and related tools [104].

At the same time, the software industry is still producing increasingly complex as well as larger software systems. However, it appears that there is a common concern about the degree of systematic adoption of the concepts, principles, and methods developed by the research community. Software practitioners do not necessarily see that research software engineering methods solve their problems. Common complains are that methods developed by the research community, and in particular formal methods, are difficult to understand and hard to scale up, and therefore they cannot meet their market requirements.

Hence, large systems which are constructed in an ad hoc way are costly, likely to fail or are not delivered on time. The construction process is not repeatable and have poor system reliability. Our understanding is that the main reasons for the gap between systematic software engineering methods and industry practice include, among some other issues, that theories, techniques, and tools developed in software engineering fundamental research:

(a) generally lack a seamless and coherent combination of theories, techniques, and tools for the various dimensions of the multi-scale design space, such as behaviour, quality of service (QoS), space and time, and thus
(b) usually consider only local solutions of methodology, tools, and models that ease part of the development, rather than the definition, management, and execution of the entire process, and especially
(c) they do not effectively address project management problems.

These problems are especially true for existing formal methods. Each formal method is based on an abstract model, which only considers one or at least very few aspects of the system, thus not addressing problems concerning all aspects of the design. In addition, there no systematic, sound, and coherent combination

of different formal methods to solve combined issues in the design. This is also true to some extend for informal software engineering methods (or *empirical methods*), and informal methods themselves are less systematic because they are prone to ambiguity and different interpretations.

For two decades, we have believed that model-driven development [63] is the key to closing the gap, as it naturally provide the linkage between the concepts and methods in empirical methods and those in formal methods. In principle, model-driven development can be used to define the development process, the technology architecture that consists of methods, techniques and tools needed in the process, and the (models of) the system architecture of the system under development. This means that use model-driven method as meta-method to *engineering formal methods* in defining a development process and technology architecture for a system construction (or for solving a software system problem in its life cycle).

In this chapter, we recall the history of principles of abstraction, in programming languages in particular, which have driven the increase of software productivity and at the same, together with increasing power of computers, have led to the development of ever-larger software systems. We argue that these abstractions are integrated and reflected in software system architecture. The increasing size and complexity of software systems requires the use of engineering disciplines, processes, and modelling. Effectively modelling a complex system always needs certain degree of formality.

Software modelling involves both informal and formal activities and they need to be managed carefully for a seamless combination. Therefore, we propose the idea of engineering formal methods where systematic and effective use is possible. We consider the main concepts, the theoretical challenges, and technical challenges, involved in establishing the framework of engineering formal methods together. The process and techniques for system architecture modelling are the core of the framework. We study the traditional separation of application domain modelling and the software requirements modelling and show how to combine them into a unified modelling of *Human-Computer-Physical System* (HCPS) architecture modelling. From that point, the approach is intended to allow a smooth transition into discussions of problems in HCPS modelling and to provide an outline framework for engineering formal methods for HCPS development.

The remainder of this tutorial paper is organized as follows. Section 2 provides a discussion on the nature of software and software system development, as well as the difference between programming and software. Section 3 summarizes some major ideas of abstraction, which mark milestones in the history of software engineering development. With this discussion, we demonstrate the importance of abstraction in a systematic approach. This provides the background for the discussion in Sect. 4 about the development processes, the architecture of software systems, and technology architectures for their development. The point we wish to emphasize is that the architecture of the software systems as produced through the development process from the requirements for the system. The architecture plays a determining role in the definition and management of the development process.

In Sect. 5, we provides a review of formal methods, mainly to show that there are a large number of formal methods. There is no single one formal method can solve all the problems in a system's development, and there can be more than one formal method used for a particular design problem in the development. Therefore, there is need for the architecture model and development process to consider the use of formal methods and tools. We propose an approach with a combination of application domain modelling and requirements modelling for software systems. Further, in Sect. 6, we present a component-based, multi-view and multi-level approach to domain modelling, including a definition of the notion of *domain architecture*. Then we continue with a discussion in Sect. 7 concerning how to produce a model for the requirements of a software system, based on a model of the application domain architecture. The combination of requirements model and domain model naturally forms a general model of an HCPS. Based on this conceptual approach, in Sect. 8 we discuss the evolution of the concept of HCPS during the last 15 years, the new challenges in modelling and design based on the *unified domain and software requirements modelling* in Sect. 6 and Sect. 7, identifying the shortcomings in the theories, techniques, and tools of existing formal methods. A conceptual architecture model is proposed in this section, and based on it, we propose a framework for engineering formal methods for the development and evolution of HCPS. Section 9 summarizes this tutorial paper.

This tutorial paper is based on a number of lectures and talks on concepts, abstractions, principles, and challenges, in software engineering, in relation to HCPS.

2 Software Development Is Different from Programming

We often wonder about the definition of software and what is the difference between programs and software. In fact, the notion of software evolved from that of a program. Here we quote some software engineering pioneers.

> A program is something that I write and I use. Software is something that I write and you use. This requires more work. It has to be generalized, it has be tested, it has to be documented, and usually it has to be maintained.
>
> Fred Brooks
> ICSE 2018 Keynote [14].

Note that the key words in Brooks talk are *generalization*, *testing*, *documentation*, and *maintenance*, and these are about the term *program* and *software (product)* in the early 1950s. Brooks went on to introduce the software entity *software systems*.

> The next kind of entity ... is a software system—a system of many separate programs working together. And this requires more work because you have to define the interfaces, and many many system debugging troubles are because my understanding of the at least the connotations of the

> interface and your understanding of the unit connotations of the interface are different, and they don't work, so now we have to do set of system integration and test ...
>
> Fred Brooks
> ICSE 2018 Keynote [14].

These are Brooks' notes about software in the late 1950s. The form of software was software systems, in which many systems were integrated through their interfaces. However, the interfaces then were not defined with rigorous semantics, which was the cause of integration problems. Margret Hamlton, who is regarded to be the first to coin term "software engineering", went even further when she talked about the insight gained in her Apollo experiences [35] to understand software [38]:

> With multi-programming, shared responsibilities and more interfaces within and between every mission phase (8 tasks based on timing, 7 jobs based on priority); man-in-the-loop multi-processing within the overall system of systems.
>
> Margaret Hamilton
> ICSE 2018 Keynote [14].

What Hamilton emphasized, in addition to the integration of many programs, was the relation between software and hardware and human interaction, with the aim of better system reliability, more parallelism, and improved reuse and evolution.

Brian Randell, a participant and editor of the report of the 1968 and 1969 NATO Software Engineering Conferences [77,91] drew "a harsh distinction" between "bespoke" and "off-the-peg software" [89,90]:

> Since [the NATO conferences] not just one, but rather many, types of software industry have come into existence, in particular those that design or tailor "bespoke" software for particular clients and environments, and those that produce "off-the-peg" software packages that are sold to thousands or even millions of customers. The first type is a recognisable successor to the software activities of the '60s. In the second, very different, type of software industry economies of scale, and Darwinian-style evolution, have a large impact on what sorts of software get implemented, and how such implementation is undertaken, e.g. involving getting hundreds or thousands of users involved, willingly or unwillingly, to help with software validation and refinement ...
> Any reasonable account of how far we've come since the late '60s (and where we have got to) has to treat these two types of software and software industry very differently. The first type of software industry has gone on to attempt ever larger and more complex tasks. But it is still subject to many of the same challenges concerning implementation cost, project schedule,

> performance and (especially) dependability that so exercised the NATO conference participants. The second now provides a wonderful marketplace of usable and useful software systems, utilities and applications that has utterly transformed society's utilisation and perception of computers. But technical monoculturalism, allied to the growth of computer networking, has led to this industry and its customers also suffering from all sorts of malicious, indeed criminal, activities that were not in any way foreseen in the discussions at the NATO conferences.
>
> Brian Randell
> Keynote at COMPSAC 2008
> and reiterated in a Keynote at ICSE 2018 [90].

With our understanding of the above quotes by the software engineering pioneers, we confirm the definition of software system which we gave in the lectures at SETSS 2014 [63]:

> We thus define a *software system* to consist of set of architected programs and data that tell a set interrelated computers what to do and how to it. Computers include all devices with programmable processing capacity, all kinds of "smart devices" as well as "computers", that now affects all aspects of daily life.
>
> Zhiming Liu
> Lecture at SETSS 2014 [63].

In this section, we had a detailed discussion on some different views concerning software and software systems, before we summarized our views in the above definition. This definition gives an explicit emphasize on *the architectural aspects of system of systems and the significance of data*. In most cases, a software system is still an engineering product or artefact. Therefore, it must be delivered with enough documentation and subsequent maintenance is required after it has become operational. Later in Sect. 4, we will see the open architecture that we propose used to combine continuous system evolution with system construction, operation, and maintenance, with data used as a resource for system evolution.

3 A History of Abstractions in Software Engineering

We now consider the historical development of software engineering based on notes given by some software engineering pioneers, literature, and Internet sources. We emphasis on the abstractions proposed historically, which had led to significant advances in software technology. In any other engineering discipline, the construction (or development) of a complex system or product is done in a well defined *process* by employing proper methods and tools. These methods and tools are developed with applications of mathematical and scientific theories and/or experiments, and preferably standardized. The requirement for a defined process and the use of proper (and standard) methods and tools is for

overall trustworthiness of the product, including the demand that the process is repeatable and certifiable, and that the product can be verified and validated according to the requirements of the product.

3.1 The Motivation and Aims of Software Engineering

The notion of software engineering, as well as issues discussed at and the aims of the 1968 NATO conference on software engineering [77], implies that construction of complex software systems should be done systematically, in a repeatable and certifiable process using sound methods and tools, like other traditional engineering branches. The consequence of increasing engineering in software production is improved quality, dependability and productivity, and deduced cost and development time. This understanding is consistent with the following definition given in Ian Sommerville's widely-used software engineering textbook [99]:

> Software engineering is an engineering discipline that is concerned with all aspects of software production from the early stages of system specification through to maintaining the system after it has gone into use.

In general, the theoretical foundations for the development of methods and tools that are used for general and systematic construction of engineering products are often based on ideas of *abstractions*. This seems even more true for the construction of a software product, since it is purely made of logical mathematical objects, unlike engineering products such as a building houses or bridges, where the functionality is mostly determined by their structure and the materials used. A software product does not exhibit physical characteristics and its functional behaviour is more complex to understand [8,12,13]. Note that a software system behaves very differently when its environment changes, which often happens. Indeed, the development of software engineering theories and methods has been driven by the idea of abstraction. We give a brief account of this below.

Abstraction in Computer Architecture. In the early 1940s, when the first digital electronic computers were designed, the commands to operate the computers were wired with the hardware [58]. It was soon realized that this design made programming these machines inflexible, and the idea of *stored program architecture* was developed, evolving to *von Neumann architecture* [103]. This was an important idea of abstraction to separate "software" from "hardware", to deal with the complexity of computing, enabling common principles to be used flexibly to program different machines.

3.2 Abstraction in Programming and Programming Languages

Along with the abstraction in the design of computer architecture, abstractions also developed in programming data structures and languages for programming the computers in the 1950s. The computers then were bare machines, and programs in the early 1940s were written, using binary (or octal) machine codes

for instructions and explicit reference to absolute address represented in binary code too. The programs written on paper had to be transferred to punched paper tapes [14]. This way of programming was very slow and error-prone. It was hard to find and correct errors in programs, as when the result was wrong there was little feedback about when and how the program execution went wrong.

In the late 1940s, low-level *assembly language* was introduced to eliminate much of the error-prone, tedious, and time-consuming programming in machine instructions. Then a program called an *assembler* was responsible for translating assembly language to machine language. The introduction of this *layer of indirection* was very important in freeing programmers from tedium such as remembering binary or octal codes and calculating addresses. Early hand-written programs were used by those who wrote them. Assembly language enabled much faster program writing and programs could be distributed to be used by others as well as the original programmers. A software product was possible, to be used by customers or clients other than the program authors. Thus, the notion of *software product* emerged (in the early 1950s) and a program could be delivered to many customers for their use. For this, documentation, testing, and ongoing maintenance are required. These became possible with programs in assembly language.

With the support of assembly language, functionalities that needed to repeatedly execute in a program or commonly needed in several programs were defined and implemented as *subroutines* [72,106,107]. From the middle of the 1950s, *high-level programming languages* became avaiable. The first widely used high-level programming languages were Fortran (1956 by John Backus), Cobol (1959), and Algol (1958–1960). These languages (or their descendant) are still in use today to a greater or lesser extent, partly due to legacy code produced over the years. The main idea of high-level programming language appeared much earlier due to Konrad Zuse in 1943–1945 [33]. A high-level programming language uses natural language word elements (such "**if** ... **then** ..." and " **while** ... **do** ...") together with mathematical expressions. It provides programmers with *strong abstraction from the details of the computer*.

Compared to low-level assembly language, high-level programming languages provided another *layer of indirection*. This layer was realized through an *interpreter* or a *compiler*, or a combination of both. The interpreter of a high-level programming language is a program that follows the program flow, reads each program statement, and then executes it by interpreting it directly as a series of machine code instruction. A compiler translates the syntax of a program into an *executable form* ("object code") before running it, and the executable form can be either machine code or an intermediate representation.

With high-level languages, more programs structures were defined and implemented as programming language facilities (or mechanisms), e.g., *functions*, *procedures*, *data types*, etc. This allowed programmers to identify and design significant algorithms, a major challenge in programming at that time. It also provided better standardization, generalization, and thus reuse in many aspects of programming thinking, program design, program implementation, program operation, and its maintenance.

Apart from algorithm design, another major concern of programming in the 1960s was data structures. Data types were introduced in high-level languages like Fortran and Algol. A data type defines a set of permissible data and the permissible operations on these data. Thus, data types liberated programmers from error-prone ad hoc design and use of data in programs.

While the level of abstraction in programming languages was increasing, software systems such as input and output converters, symbolic assemblers, and compilers, became available, together with new hardware devices. The notion of a *software system*, a system of many programs working together, evolved. However, new problems were raised with this new layer of abstraction such as interface definition, system integration, system testing, and system debugging. This was due to the lack of a means for rigorous definition and understanding the semantics of program interfaces. Different people's understanding of the connotations of the interfaces can easily vary if formal definitions are lacking. In fact, the notion of software system was first practiced in building the first known operating system, GM-NAA I/O [95]. This was an integration of a number of component programs, including an input translator, an output converter, a SHARE Assembly program, and a compute monitor program. Operating systems in general were yet another layer of abstraction, realized as a software system to manage the computer hardware and software resources, providing common functionalities for computer programs.

Advances in programming languages and tools can therefore be characterized as having been gradually increasing the abstraction level, by introducing high-level language constructs that represent common concepts and design patterns of software designers.

3.3 Abstractions in Software Development

Compared to low-level languages, high-level programming languages greatly improved productivity for software systems in the 1960s. Increasingly large and complex systems were produced. Meanwhile, computers were becoming more powerful, especially those with hardware designed for operating systems, e.g., *multiprogramming and time sharing operating systems* [23], which ended the era of bare machines with no operating system. Therefore, requirements for large and complex software systems were rapidly increasing. Software system development worldwide started to encounter issues. The growth in complexity due to combinatorial possibiities for interaction was nonlinear and this caused major concerns in project management. With this background, computer scientists and software development practitioners started to consider the notion of writing programs as an engineering discipline and the 1968 NATO conference [77] recognized the problem of the software crisis, discussed it, and called for establishing *engineering foundations, principles and methods* for the production of software systems. Many of the fundamental theories and principles and methods were then developed with important ideas concerning abstraction, including those outlined below.

Objects, Classes, and Inheritance. The concepts of *objects*, *classes*, and *inheritance* were introduced in the Simula programming language by Kristen Nygaard and Ole-Johan Dahl in 1967 [79]; Simula is regarded as the first object-oriented programming language. The important concept of class, and thus the essential notion of object-orientation, was to separate the implementation from the interface, representing the data layer and the control layer.

Information Hiding, Modularity and Encapsulation. The ideas of subroutines, procedures, functions, and data types, were further developed to cover the notion of *modules for information hiding* [80]. This was also introduced into programming language design and resulted in languages with *modularity* [59] to provide protection for related procedures and data structures, through separation of the use of functionality and data from the implementation. The concept of modularity was then further used to reflect the general engineering principles of *divide and conquer*, *separation of concern*, and *reuse*, such that the overall design problem of a large software system should be decomposed into the design of a set of modules of logically discrete functions for subproblems.

The *interfaces* of modules aught to be well-defined with respect to legitimate inputs, expected outputs, and the associated valid operations. How the functionality was to be done (i.e., the implementation) inside the module should not be visible from the outside of the module. Therefore, the overall design problem of a large software system could be divided into subproblems. These formed the basis for a *structured analysis and design technique* approach or *structured software development paradigm*. In this paradigm, *software architecture* was defined as a set of modules interacting with interfaces between them.

Theoretical study of the separation of the specification from the implementation of modules led to the establishment of the theory of *abstract data types* (ADT) [60]. In ADTs, the notion of algebraic specification of data types was defined as axiomatic (using a many-sorted algebra) and as a composition of data types. ADTs were then generalized from data type specifications to program algebraic specifications [16]. ADTs allowed software engineers to consider the requirements for data structures and specify them in early stages of the software development before the programming stage.

Information hiding and modularity were also introduced in object-oriented programming languages as *encapsulation* with *private fields* of *classes* (and *private classes*). In the object-orientated approach, the access to the functionality or the data from outside the object can only be done through the object interface, i.e., its public methods. Language facilities were provided for different levels of information hiding (or encapsulation). The notion of encapsulation was also used to raise the design level of classes and packages above the level of the programming language to *object-oriented requirements analysis and design* [8].

4 Software Development Processes and Software Architecture

In all traditional engineering disciplines, complex systems are produced in well-defined processes to improve the design, product management, and project management, in order to ensure the quality of the engineering product. Also, any serious engineering system construction is undertaken with the system architecture in mind. When software systems become more complex, the implementation requires a development process and an architecture.

4.1 Software Development Process

With software systems increasing in size, it was natural to consider the problems of planning, budgeting, and management of their development. This has to consider the development activities with their corresponding requirements, proposed techniques, and available expertise. Therefore, the notion of *software development process* emerged, dividing the development into distinct phases, identifying when and what to do at each phase, who (i.e., people) are needed to do it, and what methods, techniques, and tools should be used [61]. Notice that this definition emphasizes the relationships between development jobs, people, and expertise. This includes the definition of software artefacts to be produced at each phase. We believe that these relations form the basis for principles and techniques to improve the design, product management, and project management in large software system development [11].

The earliest ideas for the principles associated with the development process were the *top-down incremental build* [74] and stepwise refinement of Niklaus Wirth [108]. The earliest widely cited process model is the Waterfall Model [94]. Further well-known extended models include the Spiral Model [7] and the V-Model [32]. These models refine and extend the Waterfall approach with an emphasis on risks analysis for the former and verification and validation for the latter. They both emphasize the iterative and incremental nature of the development of large software systems. With the popularity of the Unified Modeling Language (UML), the *(Rational) Unified Process* (RUP) [52] emerged in the early 2000s as a combination of the Spiral process and the V-model, with more activities and the production of executable models using tools.

Software development process models are defined in terms of activities in workflows of software development. However, they are described with principles, techniques, and conditions for quality and risk control/management. This is because the development of software system activities in any nontrivial case must be done using design, construction, and verification and validation techniques/tools, as for any engineering system development. The body of principles and techniques required and availability is extremely large, as is the design space. Considering, for example, the issue of selecting a programming language (taking into account the expertise of the programmers in the development team), there are no less than 8,945 different programming languages [26,90]. Thus, it is extremely difficult to define a concrete development process for a particular

project before the project starts and to decide the activities with mappings to artefacts and techniques. We believe that this is the main reason that software development models often attract criticisms for being unrealistic or impractical. However, we take a more positive view on the importance of the notion of a software development process and accept the essential characteristics of software system development due to its inherent multi-dimensional complexity described and reiterated in the literature [8,11–14,99]. We quote Brooks:

> There is no single development in technology or management which alone promises a 10× gain in 10 years – is again true 30 years later.
>
> Fred Brooks
> ICSE 2018 Keynote [14].

We also the quote Randall from Ian Sommerville's book [99]:

> There are no universal software engineering methods and techniques that are suitable for all systems and all companies. Rather, a diverse set of software engineering methods and tools has evolved over the past 50 years.
>
> Brian Randall
> ICSE 2018 Keynote [90].

In this tutorial paper we propose, or more precisely we promote, a unified and evolutionary development processe that should accommodate *agile methodologies* [2], to allow flexible changes during system development and evolution.

4.2 Software Architecture

Routines, modules, data types, classes, and packages in high-level programming languages made software systems development increasingly systematic and disciplined, thus enabling collaborative development processes by teams of developers. However, they were still very much at the programming level for the implementation and reuse of algorithms and data structures. The interactions among the modules were mainly local invocations of functionalities.

With advances of computer systems from a single processor running a program one at a time, through *multiprogramming systems* and *multitasking systems*, to *multi-processors computers and networks of computers*, with increasingly programmable input, output, and communication devices, the size of software systems for the evermore powerful and complex computers were increasing rapidly and the interactions of their components were becoming more complicated. Designers of software systems with this level of complexity started to realize and consider design problems related to the system's overall structural organization at a level of abstraction above the algorithms and data structures. This was then the notion of software architecture design, developed from intuitions of hardware and network architecture, as well as the classical computer architecture [82,83].

Roughly speaking, software architecture is the fundamental structure of a software system, comprising *software components*, *connections* among the components, and *properties* or *constraints* concerning the components and connections. Software architecture design reflects the engineering principles of decomposition and separation of concern of the overall system requirements, beyond the level of algorithms and data structures. Its concerns cover the following issues. The software architecture should:

- reflect the architecture of the computer system on which the software is to run;
- reflect the structure of the domain business organization, business processes, and their interactions;
- represent the decomposition of the functional requirements, including its behaviour or functionality, data flows, interaction protocols, synchronization, etc., through the system [46];
- relate to the software system quality of service (QoS) attributes such as fault-tolerance, extensibility, reliability, maintainability, availability, security, usability, and other *architecturally significant requirements*, sometimes called the "ilities" [18].

Therefore, software architecture can and should serve the following purposes:

- It is the vehicle to carry the defined requirements of the software system.
- It is used as the map for identification of principles, techniques, languages, models, and tools to be employed in the further design and implementation of the software system. This means there is *a correspondence relation* between the software architecture of the system and the technology architecture for its development.
- It serves as the basis for defining and managing the project development process:
  - organizing the development teams and assigning jobs to team members according to their expertise;
  - guiding the communication and collaboration between members of the development team in their development activities, as well as their management and coordination;
  - planning the development job and estimating the costs;
  - helping in risk identification and management.
- Its design and documentation facilitates and guides communication between stakeholders, captures early decisions about the high-level design, and allows reuse of design components between projects.

Though the importance of software architecture is commonly recognized in the software engineering community, research on software architecture has been largely dispersed and there is no established theory of software architecture. Effort in software architecture research has been mostly devoted to *architectural styles*. A software architecture style or an *architectural pattern* is a general, reusable solution to a commonly occurring problem in software architecture within a given context or a kind of software system.

There are a significant number of architectural styles, described in different levels of formality, from descriptive terms used informally to describe systems, through those with precisely define components and connectors, to some that are more carefully documented as industry and scientific standards. Early and well-known architecture styles include *Blackboard*, *Repository*, *Pipes and filters*, *Client-Server*, *Layered*, etc., and model architecture styles including *Object-Oriented Architecture*, *Component-Based Architecture*, *Service-Oriented Architecture*, and *Microservice Architecture*.

A conclusion of our discussion in the previous and the current sections is that systematic engineering design of software systems has been developed through a regular increase of abstraction level in programming language design and software architecture. The establishment of software architecture and software development processes gives software design better engineering characteristics. However, our observation shows that software development has not harvested enough from software architecture and development processes, in the application of formal methods in particular. In subsequent sections, we will re-emphasize the importance of software architecture and development processes, especially their relationship, in the development and evolution of software systems for emerging networked systems, including cloud-based services systems, *Internet of Things* (IoT), and *Human-Cyber-Physical Systems* (HCPS).

5 A Review of Formal Methods

While software systems were becoming larger and more complex, their verification and validation for correctness and quality of service were becoming extremely challenging. Techniques and tools for testing and code inspection based on the semantics of programming languages could not be systematic enough or comprehensive. Without a sound mathematical model of program languages, fully systematic and comprehensive program correctness verification was not be possible. No matter what could be done, some bugs would always remain in any realistically-sized system. 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", *CACM* [30].

In this section, we present a review of the development of formal methods [10]. Unlike earlier more comprehensive surveys with a focus on the state of the art (e.g. [104,109]), we instead discuss the development of the important ideas, the common theoretical roots, and relationship with technology developments. Our main purpose is to consider the selection and integratation of different techniques into a development process.

5.1 Formal Semantics of Programming Languages

The idea that we can prove the correctness of programs was widely established in the late 1960s, with Floyd's paper on *Assigning Meanings to Programs* [31] and Sir Tony Hoare's paper on *An Axiomatic Basis for Computer Programs* (known as *Hoare Logic*) [42]. Both showed how proof of program (functional) correctness based on an *abstract* and *formal* semantics defined for the programming language. We say that semantics is "abstract" because it defines an abstraction model of the execution of each program in the programming language, instead of execution of the machine code of the program generated by the language compiler. It is formal as the model is defined precisely using mathematics from the formally defined syntax of the language, and this rules out any ambiguity. The ideas and importance of program formal verification and formal semantics of programming languages were proposed earlier, e.g., Turing's idea of "checking a large routine" [102] and McCarthy's talk on "mathematical science of computation" [73]. Thus, formal semantics of programming languages established a precise abstraction of their execution. Both Floyd and Hoare used formal first-order logic to define the semantics and the associated proofs. Later, the expressiveness, soundness, and completeness to the completeness of Hoare Logic were proved [4].

The notion of formal semantics of programming languages therefore provides a level of abstraction of program execution beyond the semantics defined by a compiler in terms of direct machine code execution. Later, formal semantics at different levels of abstraction were defined, which are mainly represented by Scott-Strachey *denotational semantics* [98,101] and Plotkin's *structural operational semantics* (SOS) [85]. Formal semantics theories are usually classified into four kinds and they are *operational semantics* as such Plotkin's structural operational semantics, *denotational semantics* such as Scott-Strachey semantics, *algebraic semantics* such as abstract data types (ADT), and *axiomatic semantics* such as Hoare Logic, in increasing levels of abstraction (but algebraic semantics and axiomatic semantics are at about the same level). These different kinds of semantics are defined for various programming languages of different programming paradigms, including structured programming languages, concurrent programming languages with shared variables, concurrent and communicating programs, object-oriented and service-oriented programming languages. A theory of unification of these different semantic theories is best studied in *Unifying Theories of Programming* (UTP) of Hoare in He [44].

Since the 1968 NATO software conference [77], theories of formal semantics became the foundation for the development of formal methods, including formal *specification*, *verification*, and *refinement*, forming a subfield of software engineering. As surveyed in [104], over a dozen Turing Award laureates have made pioneering contributions to the development of formal methods, as shown in Table 1.

Table 1. Turing Award Laureates in formal methods

Year of award	Name of Laureate	Area of contribution
1971	John McCarthy	Computational theory, semantics of LISP
1972	Edsger W. Dijkstra	Calculus of predicate transformers
1976	Dana Scott	Denotational semantics and modal logic
1978	Robert Floyd	Axiomatic semantics and verification
1980	Tony Hoare	Axiomatic semantics and CSP
1984	Niklaus Wirth	Programming language formal specification
1991	Robin Milner	CCS, bisimulation, LCF, ML
1996	Amir Pnueli	Temporal logic and verification
2007	Edmund Clarke E. Allen Emerson Joseph Sifakis	Model checking
2008	Barbara Liskov	Abstract data types, Larch
2013	Leslie Lamport	Temporal logic of actions

5.2 Formal Specification and Models

The rigorous study and analysis of program correctness based on formal semantics necessitates the requirements of the program to be specified formally, describing *what* the program should do rather than *how* the program does it. This implies that the language used for requirements specification is at a level of abstraction above the programming languages to be specified. Since the 1970s, a large number of specification languages have been developed.

There are mainly two kinds of specification languages. The first kind of languages are for specification of whole system behaviour, including data functionality, control flow, and data flow. A specification language of this kind has a well-defined semantics and a specification in the language defines a model of the software system under design. The specifications can be at different levels of abstraction and similarly for their models, formally related by partial orders (*refinement*) between specifications and *refinement* between models. We will discuss this further in the next subsection. Table 2 gives a list of major specification languages of this kind.[1]

All the specification languages in Table 2 have denotational or axiomatic semantics (or algebraic semantics), and allow specifications of different levels of abstraction. They also have the notion and rules (though in different degrees of formality) of specification refinement and thus support the top-down derivation of a program from a specification and bottom-up software systems integration.

[1] The years and features of the methods in the table are not guaranteed to be accurate or comprehensive.

Table 2. A list of software system formal specification languages

Year	Name of language	Features	Key Originators
1972	VDM [50]	Denotational semantics derivative design	D. Bjørner C. B. Jones
1974	Z [9,100]	Axiomatic semantics refinement	J.-R. Abrial
1975	Guarded commands [28] Action Systems	Logic based refinement [3]	E. W. Dijkstra R.-J. Back
1987	OBJ [34]	Algebraic semantics	J. A. Goguen
1988	B-Method [1]	Abstract machine semantics refinement	J.-R. Abrial
1988	UNITY [17]	Axiomatic semantics	K. M. Chandy, J. Misra
1990	TLA [53]	Temporal logic	L. Lamport
1992	Larch [36]	Algebraic semantics	J. Guttag, S. Garland, J. Wing, *et al.*
1999	JML [56]	Contract-based design	G. T. Leavens
2001	Stream Calculus [15]	Algebraic semantics	M. Broy
2004	rCOS [40,41,64]	Contract-based design component-based development (CBD), OO, refinement	Z. Liu, J. He, X. Li, *et al.*

Therefore, formal methods based on these languages provide formalization of notions of a software system development process and principles. There is the yet to be developed capability of supporting the definition and management of the entire processes and not to consider only point solutions of methodology, tools, and models that ease part of the design.

The other kind of specification languages and theories focuses on the abstraction of specific aspects or design problems of software execution, including *control flow*, *data*, *concurrency*, and *synchronization*. These theories mainly concern *concurrent software systems* (although the theories in Table 2 can deal with concurrency too). In Table 3, we list some well-known theories of this kind. There are further similar theories or extended versions of them. Compared to the theories in Table 2, these theories mainly deal with interaction, communication, and synchronization, with or without real-time aspects.

CSP and CCS both provide theories for algebraic reasoning of equivalence and refinement or simulation of software systems (although they can model and reason about hardware behaviour). The automata or state transition system-based models (i.e., I/O automata, Statecharts, and Uppaal) support algorithm-based verification, i.e., *model checking*. The synchronous languages Lustre and Esterel are for real-time control and monitoring systems. They have tool support for code generation and they are the basis for the implementation of the industry tool SCADE.

Table 3. A list of feature specific formal specification languages

Year	Name of theory	Features	Key originators
1962	Petri nets [84]	True concurrency event based interaction	C. A. Petri
1977	Temporal logics [86]	Property oriented	A. Pnueli
1978	CSP [43]	Channel and event based synchronous communication algebraic reasoning denotational semantics	C. A. R. Hoare
1980	CCS [75]	Event based synchronous communication operational semantics algebraic reasoning	R. Milner
1987	I/O automata [69]	Asynchronous communication distributed system model	N. A. Lynch
1987	Statecharts [39]	State machine & diagram hierarchy communication	D. Harel
1991	Lustre [37]	Data flow synchronous language signal communication	N. Halbwachs, P. Gaspi, *et al.*
1992	Esterel [5]	Synchronous language signal communication	G. Berry
1995	Uppaal [55]	Real-time model checking	K. G. Larsen, W. Yi
2001	Interface automata [27]	Component-based model	L. de Alfaro, T. A. Henzinger

5.3 Formal Techniques in Software Development

We take the definition that a method is a set of techniques and tools for solving a class of problems which are developed based on a sound theoretical foundation. In software systems, a technique means *a way of carrying out a particular task in a software development process, especially the execution or performance of phases, such as requirements specification, design, or verification.* The term *formal software development* actually means a development that involve intensive use of mathematically-based techniques, such as formal specification and verification. We do not consider separate fully formalized development or *formal engineering methods.* However, we rather propose seamlessly to integrate formal methods, more precisely formal techniques and tools, into overall software development processes, that is to engineer formal methods in software development.

The Trinity of Formal Methods – Specification Languages, Models, and Software Correctness. The specifications languages in Table 2 and CSP, CCS, Lustre and Esterel in Table 3 have a formally (mathematically) defined syntax and semantics. They can specify both the static structural view and dynamic behaviours of software systems (we only consider functionality at the moment) and thus are *system specification languages.*

With a temporal logic-based formal method, abstract execution modes of software systems are defined, such as state machines and labelled state transition systems or automata. These provide the *semantics* (*interpretations*) of the underlying logic. The formal language of the logic is used for the specification of properties as logical formulas that the system being studied, or under construction, is required to satisfy. The formulas form the *formal specification* of *requirements.*

The *correctness* of the system with respect to the specification is defined as the satisfaction of the relevant formula by the model of the system.

This is denoted as $M \models Spec$ where M is the model of the software system and *Spec* is the specification of a formula. The theories of input-output automata are mainly about models for behavioural or execution of concurrent and communication systems. They are used with one or more temporal logic statements when dealing with software system verification. Therefore, the fundamental relation among specification languages, models, logic, and software correctness, is a direct inheritance of the trinity of logic, that is language, interpretation, and proofs.

Refinement and Code Generate. A system specification language usually is expressive enough to specify models of a software system at different levels of abstraction. These languages can specify *non-deterministic behaviour*. A specification $Spec_2$ is said to be a *refinement* of a specification $Spec_1$, denoted by $Spec_1 \sqsubseteq Spec_2$, if $Spec_2$ is not more non-deterministic than $Spec_1$. More formally, if the semantics of a specification *Spec* is defined as the set of non-deterministic behaviours $[\![Spec]\!]$, $Spec_2$ is a *refinement* of a specification $Spec_1$ means the behaviour set of $Spec_2$ is a subset of that of $Spec_1$, i.e., $[\![Spec_2]\!] \subseteq [\![Spec_1]\!]$.

If the semantics $[\![Spec]\!]$ of a specification *Spec* is defined as, or it itself is, a logical formula, such as a TLA (temporal logic of actions) specification, $Spec_2$ is a *refinement* of a specification $Spec_1$ if $Spec_2$ implies $Spec_1$, i.e., $Spec_2 \Rightarrow Spec_1$. This is why we say in general *specification and program refinement means behaviours inclusion and logical implication*. Therefore, the refinement relation between specification is a partial order, and two specifications are *equivalent* if they are a refinement of each other.

The behaviours of a specification are defined in different forms for different languages, and a language can have different but related semantics. For example, CSP has a *trace semantics*, a *failure semantics* and a *failure divergence semantic* [93]. CCS has mainly an operational semantics. In principle, however, each language can and should have both operational and denotational semantics and their correctness (or consistency) with each other should be proved. Denotational semantics supports the development of verification techniques for specification (or model) refinement and transformation.

Some theories, such as Z, CSP, Action Systems, the B-method, and rCOS have fully formal rules of refinement to support stepwise program derivation. Some other theories have a mathematical definition of the refinement relation, but steps of program derivation are carried out at the semantic level.

CCS and Larch (and CSP too, in addition to its refinement theories based on denotational semantics) provide algebraic reasoning about system model equivalence. However, equivalence is define between high-level models at a high level and those at low level through hiding information, internal interaction, or behaviour.

In purely logical theories such as TLA, UNITY, and dynamic logic (an extension of modal logic), program models are fully specified as logic formulas and specification refinement is directly defined as logic implementation. Then derivations for program development are performed using the deduction rules of the logic.

Lustre, Esterel, and SCADE, are mainly for signals in the control and monitoring of embedded systems. These systems are usually not data-intensive, but their control flow, synchronization, and real-time aspects, are crucial and can be safety-critical. The tools for these frameworks have strong support for correctness-preserving *code generation*. In theory, code generated from a model is a refinement of the model.

Refinement and code generation are, or should be, mostly used in a top-down development process. Decomposition of a large model into submodels and the composition of them are part of the refinement process. Figure 1 from a formal method review paper [104] illustrates the framework of formal refinement.

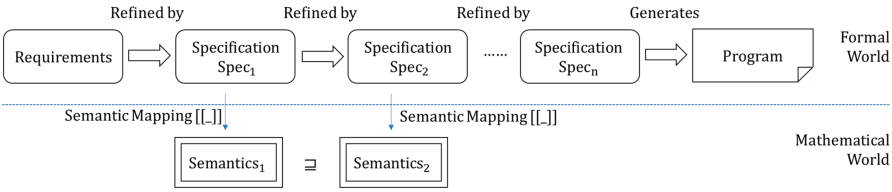

Fig. 1. Process of formal refinement

Verification – Deductive Theorem Proving and Model Checking. Given a model M of software systems and specification *Spec* of a property of the model, as a formula of the logic. The satisfaction of *Spec* by M, denoted as $M \models Spec$ can be proved from a set Γ of em known properties of M. These are made in the *deductive proof system* of the logic. This approach is known as *theorem proving*, rooted in classical *mathematical logic*.

Another method of verification of the satisfaction of a property by a model is using an algorithm that takes as inuts a model M and a specification (i.e., a property) *Spec* and outputs the answer as to whether M satisfies *Spec*. This approach is called *model checking* [22,87]. Model checking can be fully automatic and there are indeed quite a number of software tools for model checking. However, as many satisfaction problems are hard problems in terms of computational complexity, and some are NP-hard problems, the scalability of model checking is still a significant challenge. Therefore, model checking can only effective in some verification problems.

The approach of theorem proving complements model checking by enabling the proof of properties that are not feasible using model checking. Software tools for theorem proving are normally called *theorem provers*. These tools are interactive, rather than being completely automatic. The state of the art of verification tools are those with a combinations of model checking, automatic satisfaction solvers, and interactive deductive proving, although there is still some way to go to become generally effective.

Combination of Formal Techniques for Engineering Formal Methods. Theories of formal refinement help to provide insight into the understanding of the relation between development tasks and software correctness, as well as the validity of links between the techniques used in different development tasks. However, applying formal rule-based stepwise refinement to large system development is not feasible. Therefore, less formal steps of refinement are often used. In this case, verification of the correctness such refinement is needed and the verification can either be done by theorem proving or through model checking of conditions. Nowadays, interactive theorem provers employ model checking algorithms and SAT/SMT (*satisfiability modulo theories*) solvers. An SMT solver like Z3 [6] is, differently from model checking, based on decision algorithms for *constraint satisfaction problems*. Model-checking algorithms now also integrate SAT/SMT solving algorithms.

In addition to the techniques of refinement, theorem proving, model checking, and SAT/SMT solving, there are also techniques of *abstract interpretation* [25] and *program synthesis* [70]. If we consider refinement being best for top-down design, abstract interpretation can be used to build more abstract models from more concrete models, especially for execution models of programs. Program synthesis is the construction of a program from a specification. This is usually through constructing proof of the specification (which is a logic formula) to form a program. Alternatively it may be used to find a model such as an automaton from a formal logical specification. Abstract interpretation makes use of formal proof techniques and can have different degrees of automatization.

Through over 50 years research and development, there are now a large number of specification languages, models, and techniques for verification. We have covered some of the most significant advances in the development of verification (or proof) tools. Thanks to improving online resources, we refer readers to the Wikipedia pages for the techniques of theorem proving, model checking, and SMT solving techniques:

- https://en.wikipedia.org/wiki/Automated_theorem_proving
- https://en.wikipedia.org/wiki/Model_checking
- https://en.wikipedia.org/wiki/Satisfiability_modulo_theories

Each of the above pages contains a reasonably complete list of tools for the techniques.

The crucial importance of formal specification, design, and verification cannot be ignored now with the overwhelming increase of software-based safety-critical applications. For example, aircraft control software systems and car software systems may be required to be formally specified and verified (e.g., by standards).

However, with the availability of the large body of knowledge of theories, techniques, and tools, software engineering practitioners, those in industries often wonder which formal methods are the best. Indeed, we have been asked questions like "which of method X and Y is better?" Considering the complexity of software development and the large number design problems nowadays, our answer to this question is a restricted version of the earlier quotes from Brooks, and Sommerville [99] in Sect. 3, that is:

> There is no single formal method or technique that is alone enough for a non-trivial software system development. Rather, a diverse set of formal techniques and tools has to be combined and used with informal engineering methods.

Therefore, in the design and management of a software development process, there is a need to consider which development tasks require what formal techniques and tools, as well as the required expertise in the formal methods to be used. The software architecture should also serve as a knowledge map for the identification of expertise, techniques, and tools. This is the main intention of the notion of engineering formal methods, which we propose in this paper.

6 Domain Modelling

The important aspects of abstraction, development processes, and software architecture, as discussed in Sect. 3 and Sect. 4 have led to empirical modelling and design of software systems. The formal theories and techniques reviewed in Sect. 5 are motivated by and reflect those informal and conceptual ideas and principles.

As claimed in Sect. 5, that a development of a software system cannot be purely formal, it must involve interactions of formal and informal, objective and subjective, and technical and non-technical activities [105]. There is unfortunately an ignorance or lack of understanding of these interactions, in both the academic community and the software industries. This is a crucial factor for the slow adoption or integration of formal methods in industry software development and the lack of enough interest among students to study formal methods.

In traditional model-based system and software engineering, domain modelling and system modelling are in general separated to develop a framework for the combination of informal and formal modelling. We propose to study the relation between modelling the application domain and modelling the requirements of software systems for the domain, and to develop a unification of them into a framework for modelling *Human-Cyber-Physical Systems* (HCPS) in Sect. 8. To this end, we need to understand some core notions, which are important in scientific and engineering modelling, including *concepts*, *relations among concepts*, *functionality*, *processes*, *components*, *systems* and *architecture*. These ideas are key to the development of object-oriented, component-based, and service-based modelling frameworks in software engineering from both domain modelling and software systems modelling. We envision that they are important in developing a unification of these modelling frameworks for HCPS.

6.1 Modelling in General

Domain modelling is part of system requirements modelling and analysis, and in general is to clearly and precisely define the functions, operations, services, and processes, together of course with the related concepts and data too. It also relevant to the study of modelling more generally.

In general, a *model* of a *thing* or *object* is an abstraction or a representation of the *essential properties* of that thing or object. What is essential depends on the modeller's interest. In engineering and science, modelling is treated more seriously and systematically. As Lee states, "the act of modeling involves three distinct concepts: the thing being modeled, the model and the modeling paradigm" [57]. There, he gives two examples of modelling. The first is:

> Newtonian model of a mass and a spring (the thing being modeled) consists of an ordinary differential equation (ODE) (the model). The modeling paradigm is the mathematics of calculus and differential equations.
>
> Edward A. Lee 2015

The second is:

> . . . a computer program written in C (the model), which models the behavior of an electrical machine (a computer) that transforms binary data stored in electrical memory. Here, the modeling paradigm is the computer science theory of imperative programs.
>
> Edward A. Lee 2015

6.2 Domain Processes, Concepts, and Architecture

Software, and in fact any ICT system, is typically developed for use in an *application domain*, and evolves along with the evolution of the application domain. The first and most common informal activities are in the understanding of the application domain together with the capture and analysis of the requirements of the software system to be designed and maintained.

However, there are different definitions with respect to applications. One commonly used in the community of model-driven development is that the application domain of a software is the segment of reality for which a software system is developed, which can be an organization, a department within an organization, or a single workplace. In fact, a domain in this view is more precisely a concrete *domain scenario*. Another understanding of the domain of a piece of software concerns the *knowledge area* of the application, such as medical and healthcare, enterprise of trading, banking, aerospace, and automotive.

A domain is mainly described in terms of functions, operations, services, and processes in that domain. These are usually called *domain functions*, *domain operations*, *domain services*, and *domain processes*. These are understood at two levels. At the meta level (conceptual level), they define types (or classes) of *instances* of the functions, etc., that are actually executed in the domain. In the meta level, they are described in terms concepts and types of data. At the instance level, executions are defined in terms of instances of concepts (also called *objects*) and data of data types at the meta level. With the view of the domain as a *knowledge area*, the application can be described more generically, but it is then hard to define the stakeholders, and thus scope, boundary of the application, and

the concrete performance and properties of the services. Then processes cannot be easily specified. On the other hand, given a concrete application scenario, a domain is always defined with a boundary that demarcates its content. The boundary is often but not necessarily related to a boundary in the physical world, such as a department of a university, a floor of a smart building, or even, say, the heart of a person.

We take the view that a domain always has a boundary that demarcates its content. This *content* comprises the domain functions, domain operations, domain services, domain processes, together with the *concepts* and *types of data* (i.e., data at the conceptual level), which are used in the description of the functions, etc. It is important to understand that the concepts and types of data define the *objects* and *data* which are involved in the execution of the functions, operations, services and processes. Some of these objects are *actors* and some are *resources*. The data is usually used to represent properties of objects, functions, operations, and processes. For domain understanding in general, concerning domain understanding and modelling, analysis at both the meta level and instance level (or execution level) are needed.

Functions, operations, services and processes are all *functionality* or *features*. Their difference is mainly in their granularity and there are applications that are function oriented, operation oriented or interaction oriented, and service oriented or process oriented, depending on their domains. In general, a process-oriented domain often has a layered structure in which processes are formed at a higher layer though coordinating and orchestrating functions, operations, or services in the layer below (sometimes other layers). In this case, processes of a layer can also be abstracted and treated as interface operations or services. These processes can then be used to compose higher-level processes, in order to allow them to collaborate and share resources. For example, the processes related to sales, inventory management processes, and staff management processes in a supermarket are managed and coordinated at the supermarket management level. Furthermore, an enterprise of a supermarket chain comprises the processes that manage, control, and coordinate the processes of the supermarkets within the enterprise.

Thus, a well defined domain forms a *system* and it has a clear horizontally component-based and vertically layered *architecture*, regardless of the level of computerization (or digitization) in the domain. Even with a purely manual system, i.e., with no digital computers used at all, a domain also has an architecture of its organized services and processes in its sub-domains and different layers of abstraction. The architecture also has its executions, which are called the *dynamic behaviour of the domain (architecture)*. Communication for interaction and exchanges of data among the components in a horizontal level and in different layers are involved in a domain execution.

Taking the view of Lee on modelling in Sect. 6.1, the things to be modelled in domain modelling are the domain processing, together with the objects and data involved in the processes. The overall models of these processes form a model of the domain system architecture. There are a few well-known paradigms

for domain modelling, including Jackson's *problem frames* [47,48], Parnas' *four-variable* method [81], and the *use-case driven approach* [54,61]. The key idea of these methods is to identify the interface between the environment and (part of) the software system under design (SUD). At this stage, the SUD is actually a model of a domain process or a set of domain processes to be realized by the SUD. These domain processes are, in general, discrete sequences of state changes caused by interactions with their environments. Some domain processes are physical processes and modelled with continuous functions, which are often defined by differential equations. In more general and complex application domains, there are *hybrid processes* of discrete state changes and continuous evolutions between discrete state changes. Obviously, when producing models for processes, the objects and data involved must be modelled. A modelling paradigm provides a theory and techniques for modelling the processes and the architecture at different levels of *abstraction* and from different *view-points*. We give two examples to show these important ideas.

6.3 Discrete Interactive Processes and Physical continuous Processes

Consider a domain $\mathcal{D}$ with it boundary. We are usually (at a moment of time) concerned with a set of processes of the domain that forms a *component* $\mathcal{C}$ of $\mathcal{D}$. The processes in $\mathcal{C}$ can have interactions among themselves, called *internal interactions* or *internal communications*. They also have interactions with the rest of the domain, denoted by $\mathcal{E}$, called the *environment*. These interactions are the *external interactions* or *external communications*. The domain $\mathcal{D}$ can be represented as $\mathcal{D} = \mathcal{C} \parallel \mathcal{E}$, and in this model, $\mathcal{C}$ is modelled with "extensive" details, while the model of $\mathcal{E}$ only focuses on the interactions with $\mathcal{C}$ and abstract assumptions on its behavour. It can be seen that we can carry out incremental modelling of $\mathcal{C}$ by considering processes one by one.

A Modelling Example of a Discrete Interactive Process. We give an example in which the domain system mainly comprises data or information processes. The argument we make is that different views should modelled in different languages; different domains should be modelled in different languages; even when notations used are syntactically the same, their semantics may be different (at a certain level of abstraction); and the language used should be within the expertise of the modeller. These different modelling notations are known as *domain-specific languages* (DSL).

Example 1 (Point of Sale). We take the problem statement from Larman [54,61]:

> Consider the construction of a software system for Point of Sale to be used in a store to processes sales and manage inventory. The goal is for increased checkout automation to support faster, better and cheaper services and business processes, and [...] quick checkout for the customer fast and accurate sales analysis.

The modelling always starts with informal analysis and description of the domain. For such a data-intensive system (or information system), we propose that in the domain modelling, one needs to identify:

- *domain* (*business*) *processes* and represent them as *use cases*;
- *domain concepts* together with their *relations* and model them by a *conceptual class diagram*;
- *business rules* and *constraints* and describe them as *system invariant properties*.

It is important to understand that carrying out each action in a business process involves objects and data. The objects are the main instances or entities of domain concepts. Objects collaboratively involved in an action must be related in some way and this is defined as relations between the domain concepts that define these objects. We can see the understanding of use cases is intrinsically *object-oriented*, although the design and implementation of the software does not have to be object-oriented. When describing a use case, it is important to be clear about the "significant" concepts and objects, giving them definitons that are precise enough for our purposes. The same attention should be paid to relations between concepts.

Figure 2 is an sample description of the use case for checking out a customer, where only cash payments are handled, We give the use case the name "Buy Items with Cash". The description is informal, but tends to be structured and precise.

Note that here we focus on domain processes without even needing a computerized system. Therefore, when we have an interaction of a process with input or triggering actions, we ask what the action will actually have an effect in terms of storing data, checking conditions, doing computation, and making decisions. Specifying what to do is essential, but not what or who does it, or how it is done. In the *Buy Items with Cash* process, we can assume that the Cashier actually takes the computational responsibilities to carry out the actions with tools, even if these are just pen and paper.

We take the use case description as a global behavioural view. It is more concerned with the interactions, but it contains information about domain concepts and data. It also contains an abstract description of the data functionalities of each interaction. However, it is not easy to handle and communicate such an informally described use case. With further analysis, we can create the interaction view model as the sequence diagram in Fig. 3(b) and the conceptual structure view as the *conceptual class diagram* in Fig. 3(a), which we call the *conceptual model* of the use case. These models are more formal with symbolic representations of the process, its concepts, and associations among concepts. We treat a use case as a component of the whole domain system and document use case interaction actions in the component box of Fig. 3(c). These actions in the box are the interface operations of the component with its environment. To study the dynamic flow of control and for application dependency, property verification, such as reachability properties, a state machine model as in Fig. 3(d) is

Use Case: Buy Items with Cash

Typical Course of Events

Actor action	**What to do**
1. This use case begins when a **Customer** arrives a **Cash Desk** with **items** to purchase.	
2. The **Cashier** obtains and *records* the **identifier** from each **item**.	**3.** Determines the **item price** and *adds* the **item information** to the **running sale's transaction**. based on the **catalog** The **description** and **price** of the current **item** are presented.
If there is more than one same item, the Cashier can enter the **quantity** as well.	
4. On completion of the item entry, the Cashier indicates to the Cash Desk that item entry is completed.	**5.** Calculates and presents the *sale total*.
6. Cashier tells the Customer the total.	
7. The Customer gives a **cash payment**, possibly greater than the sale total.	
8. The Cashier *records* the **cash received amount**.	**9.** Shows the **balance** due back to the Customer. Generate a **receipt**.
Exceptional course	
1. Line 2: Identifier is not valid or found.	Exception handling.
2. Line 7: Customer does not have enough money.	Exception handling.

Fig. 2. A use case description

needed. The models in Fig. 3 become fully formalized only when the semantics of the diagrams there are also completely formal. Indeed, their formalization is given in the rCOS method for formal refinement of object-oriented (OO) and component-based systems [19,21,41,65].

Note that we are modelling a domain process and the name of the component *BuyItems-Controoler* represents the *role* or *agent* that carries out what responsibilities of the actions are required to do. In this example, it can be the cashier herself or another agent. Here, the entity *agent* is to provide the separation of the interface events from their effects. We will see in the next section that this separation helps us to identify what can be digitized in a system.

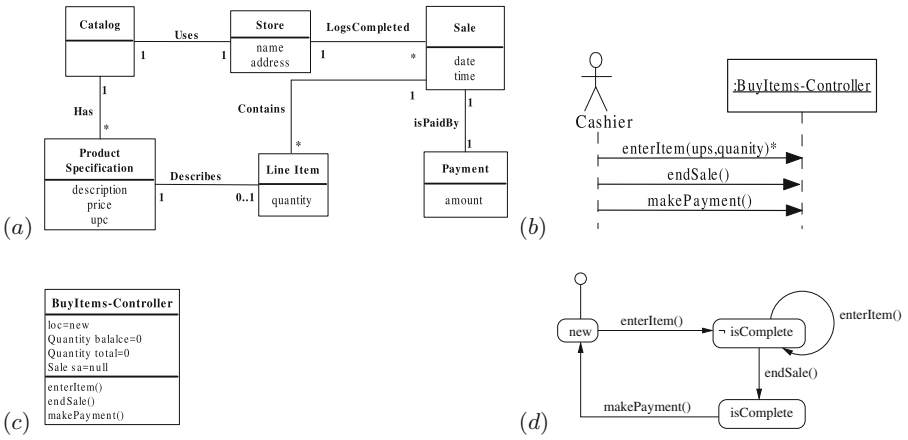

Fig. 3. (a) Class diagram of POST, (b) sequence diagram of use case *BuyItems*, (c) use case controller of *BuyItems*, and (d) state diagram of *BuyItems*

To represent what an action does clearly, we use the notion of a *contract*, which is a *precondition* and *postcondition*. For example, with respect to the action $enterItem(upc, quantity)$, we describe its contract as follows:

- Precondition: *upc* exists and is valid.
- Postcondition:
  - If a new sale, a *Sale* was created.
  - If a new sale, the new *Sale* was associated with the *CashDesk*.
  - A *SalesLineItem* was created.
  - The *SalesLineItem.quantity* was set to *quantity*.
  - The *SalesLineitem* was associated with the *Sale*.
  - The *SalesLineItem* was associated with a *ProductSpecification*, based on *UPC* match.

The precondition assumes what input parameters are allowed for execution and the postcondition specifies what property will be guaranteed after the action if the precondition holds before the action start. Again, the above description of the precondition and postcondition of an action, which form an *contract* of the operation, is informal. The formalization of contracts of use case operations is part of the rCOS method for formal refinement of OO and component-based systems [19,21,41,65].

In rCOS, we have extensively studied ways of creating these view models (see Fig. 3) and defined their formal semantics individually, together with their consistent integration. Now, let us use $\mathcal{D}$, representing the domain of a given supermarket, which has a large number of business processes and objects. Let BuyItems-Controller be the formal integration of the identified *Buy Items with Cash* use case. Then, $\mathcal{D} = \mathcal{E} \parallel$ BuyItems-Controller, where $\mathcal{E}$ represents the environment of BuyItems-Controller in $\mathcal{D}$, i.e., the interactions of actors with

BuyItems-Controller. Notice that here $\mathcal{E}$ is an abstraction of the rest of $\mathcal{D}$ apart from BuyItems-Controller, only stating the constraints on the external interactions with BuyItems-Controller. When another process (or use case) P is modelled by $\mathcal{E}_1 \parallel P$, then the domain model becomes $\mathcal{D} = \mathcal{E}_2 \parallel$ BuyItems-Controller $\parallel P$, where $\mathcal{E}_2 = (\mathcal{E} \cup \mathcal{E}_1) - (\text{BuyItems-Controller} \cap P)$. Note here that the set of behaviours of $\mathcal{E}_2$ is the union of those of $\mathcal{E}$ and $\mathcal{E}_1$, without the intersection of the behaviours of the use cases BuyItems-Controller and P.

The domain can be incrementally modelled, domain process by domain process. In our project on the CoCoME benchmark [92] problem, which is an extended version of the Point of Sale problem, we used model-driven design of a system with eight use cases [20,21]. The notations we used in this example are exactly from rCOS. In our previous work on rCOS, however, we were concerned with software system requirements modelling, considering the relation between the software system being modelled and the environment at the same time. In this section, we propose a framework for domain modelling independently, in order to allow us to identify where, what and why software systems are required in the domain.

A Modelling Example of a Continuous Process. This example shows the key ideas of modelling a continuous process. We wish to emphasize that the process of building a model of a thing is a cognitive procedure of analysis, simulation, and reasoning about a sequence of models, involving abstraction, refinement, decomposition, and composition.

Example 2 (Pacemaker). For example, consider the design of an artificial pacemaker, which is a device to maintain an adequate heart rate, for example if the heart's natural pacemaker is not fast enough. For the functioning of a heart, we quote the description from the paper [49]:

> "The Sinoatrial (SA) node, which is a collection of specialized tissue at the top of the right atrium, periodically spontaneously generates electrical pulses that can cause muscle contraction. The SA node is controlled by the nervous system and acts as the natural pacemaker of the heart. The electrical pulses first cause both atria to contract, forcing the blood into the ventricles. The electrical conduction is then delayed at the Atrioventricular (AV) node, allowing the ventricles to fill fully. Finally the fast-conducting His-Pukinje system spreads the electrical activation within both ventricles, causing simultaneous contraction of the ventricular muscles, and pumps the blood out of the heart."

End of Example

Model Refinement: The designer needs to understand a natural heart, both of its structure and behaviour. There can be many models about the rate of a heart, with different levels of accuracy. With each model, the healthiness of the heart can be decided. Therefore, we can have a number of models of a healthy heart.

For example, given a heart, we can use the number of beats per minute (b/m) as its model, and we denote this model as $H_{b/m}$. We define the heart to be healthy if its rate is within [50,90] b/m. Another model of a heart is produced by an electrocardiogram (ECG) test, denoted by H_{ecg}, and the heart is healthy under this model if its ECG test is "normal" (defined according to medical science). For use, there are sophisticated models produced by medical experts, as used for example in the paper [49].

We say that any heart H_{ecg} is a *refinement* of $H_{b/m}$, since there are more details represented in the former than in the latter. More importantly, whether the heart is healthy (or unhealthy) according to H_{ecg} must match this criterion according $H_{b/m}$.

Model Decomposition: Further study on the components of a heart by cardiac electrophysiologists has led to models of hearts as compositions of models of components. For example, a heart has three components, which work together for blood supply. They are the *sinoatrial* (SA) node, the left and right *atria*, and the left and right *ventricles*. The SA node, which is also called the natural pacemaker, regularly generates an electrical pulse; the pulse causes both atria to contract, causing the ventricles to collect and expel blood. We use SA to denote the model of the SA node, LA and RA to denote the models of the left and right atria respectively, and LV and RV the models of the left and right ventricles respectively. Then a component-based model H_c of a heart H is defined to be the composition $SA \| LA \| RA \| LV \| RV$.

The example shows that a model of an object is a representation of the *observation* of the modeller, and the level of details in observations can be incrementally refined top-down or conversely abstracted bottom-up. Refinement by decomposition is to open up an abstract "black box". Abstraction by composition of components is the process of hiding internal details of relations and interaction between components to make a black box.

Multi-views Modelling: When building a model of a complex object, modellers observe the object from a number of different viewpoints [19]. Each view is concerned with certain aspects of the objects. Different views can be orthogonal, but often interrelated, and thus they can be investigated at different times by a single modeller or concurrently by several modellers, with collaboration.

For example, we may start by observing the rate of a heart as its *global behaviour view* and then consider its organization to see its main components of sinoatrial node SA, the left and right atria, LA and RA, and left and right ventricles, LV and RV. For each of these components, we are concerned about the views of their observable functionality in separation, the structure view of their dependency for interaction, and their dynamic interaction view. We now discuss a number of views and show how we can build their models.

Heart Rate: We use a timed function $H_r : Time \mapsto \{0, 1\}$ to represent a heart rate as shown in Fig. 4, where $Time$ is the set of non-negative real numbers in this example. We omit the way the rate of a heart is measured here.

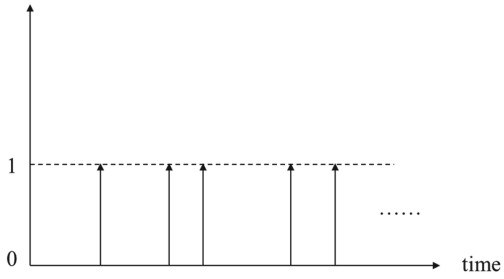

Fig. 4. Heart rate

Interface View of Components: We can use a declarative description for the static functionality view of the components. The following specifications are given using the syntax of rCOS component declarations, but without adhering to the rCOS semantics here however.

```
Component SA {
   provided interface {getSignal()};
   output interface {outPuls()};
  }

Component LA {
   provided interface {inSignal()};
   output interface {contract()};
  }

Component LV {
   provided interface {inSignal()};
   output interface {bloodIn(), bloodOut()};
  }
```

We can have the same models for RA and RV as for the left counterparts above, respectively, and they can have diagrammatic illustrations similar to UML component diagrams, such as those in Fig. 5.

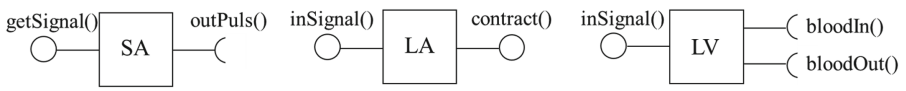

Fig. 5. Components of heart and their interfaces

System Structure View: We build the structural view of the heart by linking the operations (or signals) in the provided and required interfaces of the components. For example, we can use a renaming operation on a component

to change the names of operations in order to represent the linkages of the interface operations in different components for interactions. To this end, we rename `inSignal()` of LA and RA as `outPuls()`, to link the output signal of SA to the input signal of LA and RA for their interaction. We denote the versions of LA and RA after the renaming as $LA[outPuls()/inSignal()]$ and $RA[outPuls()/inSignal()]$ respectively.

In the same way, we can rename the `inSignal()` of LV and RV and have $LV[contract()/inSignal()]$ and $RV[contract()/inSignal()]$, to link the output of LA and RA with LV and RV for their interactions. Of course, we can rename interface operations with different renaming functions. For example, we can the output signal `outPuls()` of SA as `inSignal()` of LV, instead of renaming `inSignal()` of LV and RV. We can have the composite model:

```
Component Heart {
 LA || LA[outPuls()/inSignal()] ||
 RA[outPuls()/inSignal()]|| LV[contract()/inSignal()] ||
 RV[contract()/inSignal()]
}
```

In a formal modelling language, this can be written as an expression of the form $P_1 \parallel P_2 \parallel P_3 \parallel P_4 \parallel P_5$. Figure 6 give a diagrammatic illustration of the structure model. The boundary of the domain system is now clear, formed by the interfaces of SA, LV, and RV, to the outside of the heart.

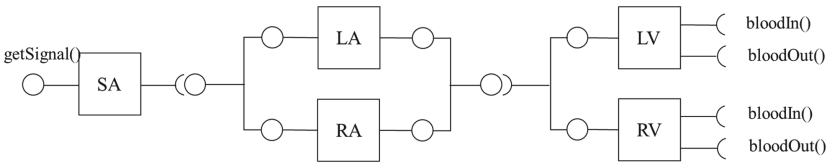

Fig. 6. Structure view of hear

Component Interaction View: The model of the local interface views and the structural view concern static aspects of a heart. To understand how a heart functions, we need to observe the dynamic behaviour of the components and that of the composite heart as a whole.

There are different views of the dynamic behaviour. An important view is how a component interacts with its environment, through its interface operations or signals. Furthermore, there can be different views of interest about interaction behaviours, such as an untimed temporal view and a timed view. We can model both together in one model, but dealing with them separately can be easier when these two aspects are complex. For the temporal order of interactions of a component with its environment, we can use component sequence diagrams [61], as shown in Fig. 7.

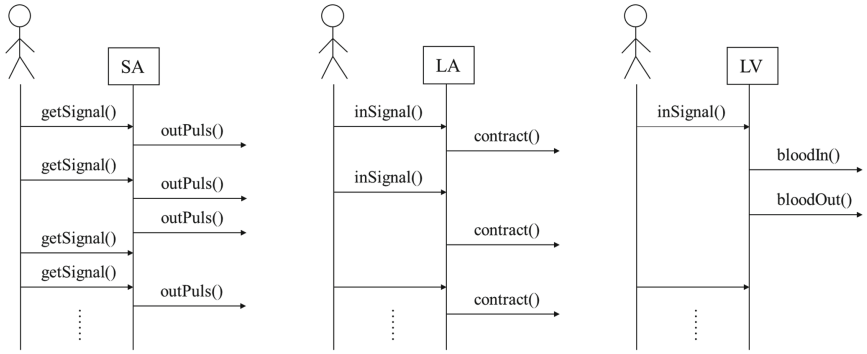

Fig. 7. Component interaction view

System Interaction View: As for the composed whole system, there are two interaction views, the black-box view and a "white box". With the black-box view, as shown in Fig. 8, one can only observe the interactions of the system with its environment without being able to see the internal interaction among the components of the system. Therefore, no other components can interact with the components inside the system through internal interfaces, i.e., those linked pairs of provided and required interface operations. On the other hand, in the white-box view, the interface operations remain visible to the environment and thus they still provide interactions to the outside environment.

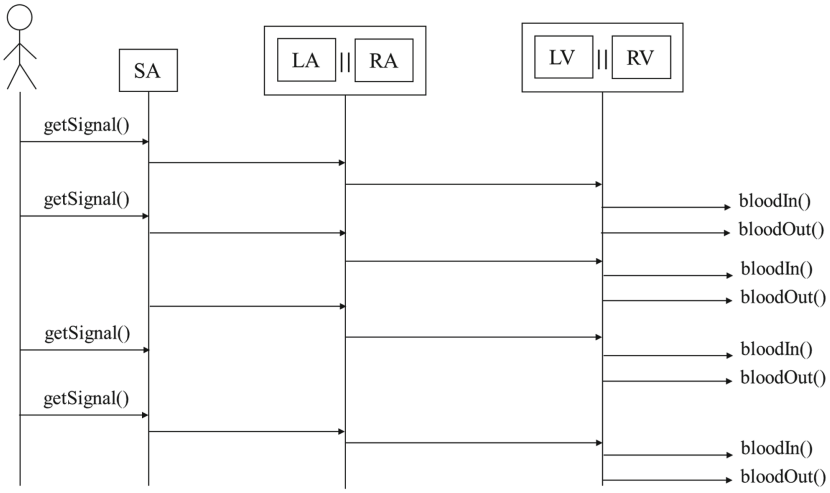

Fig. 8. System interaction view

We notice that it is difficult to use a sequence diagram to show the broadcast signal `outPuls` from SA to both LA and RA, and this is why we make their parallel composition $LA \parallel RA$ a single component.

Timed Interaction View: A model, such as a sequence diagram, for the untimed temporal order of interactions, expresses the causality relation between events, but it may not be expressive enough to define the exact time points of synchronization, depending on the notation used (true concurrency vs. interleaving). However, a model of the timed interactions will present the time information about when exactly an event can or will happen. We can simply use timed functions to represent timed interaction behaviours.

For a component, such as SA, a timed interaction behaviour is a function from time to a set of its interface operations:

$$Ttrace(SA) : Time \mapsto 2^{\{inSignal(), outPuls\}}$$

For any time point $t \in Time$, $Ttrace(SA)(t)$ is a subset of the $2^{\{inSignal(), outPuls\}}$ power set, representing the subset of events in $\{inSignal(), outPuls\}$ occuring at time point $t \in Time$. When $Ttrace(SA)(t)$ is empty, nothing is happening at time t; and when more than one event belongs to $Ttrace(SA)(t)$, these events occur at the same time (with no time delay, even when one causes another). When $Ttrace(SA)(t)$ is a singleton set, we use the element of the set to denote the set for notational convenience.

Obviously, not all functions from the time domain to the interface operations of the component are possible or allowed interaction behaviours. The set of allowed interactions can be constrained by using logic formulas, e.g., stipulating that when an `inSignal()` occurs in SA, an `outPuls()` must occur in a millisecond; and when an `outPuls()` occurs SA, `contract()` must start in half a millisecond.

Another important view can be trajectories of activation, movement (contraction) of the muscle, and the rate of inflow and outflow of blood. Models for these aspects can be defined using differential equations.

Note that in the example above, we only give conceptual discussions about modelling, but we do not claim that the example models in the discussion and figures are necessarily correct models of a heart. Correct models of a heart should be created by or in collaboration with domain experts, in this case cardiac electrophysiologists.

We choose the notations to use from a subset of UML diagrams, which are formally defined in rCOS. However, the semantics we use is intuitive and informal. It is different from the formal semantics, which is defined by rCOS, where interaction is through method invocations. The interactions for the models in Example 1 are signal-based and synchronous broadcasting communication is appropriate. This is more common for systems in which components (processes) are physical objects instead of discrete items of information and data.

7 Traditional Software Requirements Modelling

Different software systems to be developed play different roles in different applications, i.e., domains, or in different parts of domain system. Thus they are modelled, designed, and deployed differently. There are typically two kinds of software:

1. *Software systems to provide digital automation of domain processes.* In this case, a process of the software system is to automate, fully or in part, a domain process. This domain process can be for a computational process, data or information processing, or a control process, that was previously performed manually by *agents* or *roles* in the domain processes. We call this type of software system an *automation software system.* The software system to be developed for the *Point of Sale* system discussed in Example 1, indeed most information systems, are of this kind. Reactive control systems, such as traffic light control system and railway level crossing control systems, are also primarily automation software systems.
2. *Software systems to autonomously monitor and improving the performance of the domain system.* In this case, the processes of the software system are added to interact with processes of the domain systems. They can be seen as redundant components to complement, improve, or correct the behaviour of domain processes. Such software processes are more autonomous, and thus we call them an *autonomous monitoring software system.* The software for an artificial pacemaker is a software system of this kind. The software system for an autonomous room condition monitoring system is also such as system.

We do not claim that there is a clear boundary between these two kinds of software systems. It is usually the case components of both kinds co-exist in a software system. We make this classification because their requirements capture, analysis, and modelling are different Their design of interaction protocols is usually different too.

For a given domain, the requirements modelling of a software system is based on the model of the domain. We consider requirements modelling for automation software systems and autonomous monitoring software systems respectively in the following two subsections and then provide a uniformed notation for their models.

7.1 Requirements Modelling for Automation Software Systems

To design an automation software system, we identify the domain processes (or use cases or tasks) that are to be automated by the software system, together with purposes and added business values. Taking the domain model of each of these use cases, we produce a software model to replace the domain agent (generally called a *use-case controller*), which handle the use case in the domain.

For example, if we are to automate the use case *Buy Items with Cash*, we will build a model of a software component in the following way:

- First, we declare a component with a name, say `Component SalesHandler`, together with its provided and required interface operations (in this case none), where `Component` is a keyword to indicate that this is a software component:

```
Component SalesHandler {
   provided Interface {
      startSale();
      enterItem(upc:UPC, quantity: Quantity);
      ednSale();
      makePayment(amount: Currency)
   }
}
```

  This declaration is made based on the `BuyItems-Controller` component box and the sequence diagram in Fig. 2(b). We can also specify the state variables of the component, parameters of the interface operations, and define the types of the variables and parameters.
- Then we make a sequence diagram like the one in Fig. 2(b), but with the agent `BuyItems-Controller` replaced by `<<component>> SalesHandler`.
- Next we identify the concepts and the associations in the conceptual model within the domain model that are involved in the operations of the use case and create a UML class diagram, for example. For this, the contracts of the operations, both informal and formal, are very useful for making sure which objects need to be known (stored, operated on, and transferred through interactions) in the software system, i.e., the "need to know policy" in software modelling. This means the part of the conceptual model of the use in the domain is mirrored in the software model of the use case.
- Now we create the dynamic behaviour model, for example in a UML state machine of the software component.
- We refine the contracts of the operations in the domain use case, or create the contracts if no contract for an operation is given in the domain modelling. We specify the contracts formally when necessary.
- Finally, we define the mapping from the domain elements to the software model and record it as part of the knowledge of the modelling and the modelling elements, for later traceability and other purposes.

It is important to note that developing the models of the software model of the component for a use case is not necessarily a purely mechanical process. Rather, it is a further refinement to the domain use case. Also, in the domain model, the models of a use case do not have to include all the view models, most importantly, the mapping between the modelling elements in the domain use cases and those in models of the corresponding software components.

The modelling elements in the domain and the software components, including sequence class diagrams, sequence diagrams, state machines, contracts of operations, etc., can be defined with formal syntax and semantics for formal analysis, refinement, and verification. If we are only concerned with functionality, each use model consists of the following view model:

1. use case interaction view models – possible modelling notations includes UML sequence diagram, CSP, trace models: rCOS use case sequence diagrams, etc.;
2. conceptual class model (or data model) – possible modelling notations: UML class diagrams, Z notation, VDM, rCOS, etc.;
3. contracts of the case operations – possible modelling notations: rCOS or Hoare Logic;
4. state machine – possible modelling notations: labelled state transition systems (LTS), UML state diagrams, automata, Statecharts, etc.;
5. requirements specifications regarding value-added services, performance, safety, security, fault-tolerance, etc.

With the software components, the use case actors, the *Cashier* (or any other designated agent[2]), interacts with the software component by only passing information into the component and receiving outputs from it, without actually performing any of the functionality of the actions.

Replacing the original agent for actually doing the computational tasks in the domain with the software component requires the design of an interaction mechanism between the software components to realize the interaction between the computer and the actors of the use case. If the actor is human, human-machine interaction technology is used. If the actor is another digital system, appropriate protocols and mechanisms have to be designed. However, at the requirements modelling stage, concrete interaction technologies (or input/output technologies) and protocols are not the main concern.

The philosophical thinking behind this modelling approach is that if we put the model of the software component within the model of the real-world domain, in place of the original agent for the use case, we actually transform the former to a model of a new domain, with the realization and deployment of the software system in the real world transforming the old world to a new world. More precisely speaking, we transform a domain model $\mathcal{D} = \mathcal{E} \parallel \mathcal{C}$ to a model $\mathcal{D}' = \mathcal{E} \parallel \mathcal{S}$, where $\mathcal{S}$ is model of software components to realize the domain component $\mathcal{C}$. With this view, our mindset forms separated views of the real-world domain and the digital system, allowing a view of evolution within the software-defined world. Indeed, we are working in the real world with *software-defined domains*, or actually a software-defined world. This software-defined domain view also provide a model for prototyping and simulation.

Incremental Software Modelling and System Evolution. Sometimes it not possible (due to the state of the art of technology or for financial considerations) or not necessary to automate a whole process. We can model a component that interacts with the original domain agent of the use case. In this scenario, the software component will work in collaboration with the agent to fulfill the use case. For example, we can omit the actions of handling the payment from the automation. Then the software component only needs to inform the agent

[2] In old Chinese shops, the Cashier used to send the bill to an accountant sitting in a glass room doing the calculation.

(the *Cashier*) of the total after the `endSale()` to handle the payment. We can also take the domain system with software components that have already been modelled as new domain systems and identify further use cases for automation.

In other situations, we can also extend a use case for which a model of a software component is available. For example, we can extend the *Buy Items with Cash* to a more general *Buy Items* use case that, in addition to cash payments, provides services for credit card payments and cheque payments. In this case, we can extend the model of the software component for *Buy Items with Cash* to a general *Buy Items* use case [63]. This can even be done after the software component for the original use case *Buy Items with Cash* use case has become operational.

Note that new IT developments can enable new use cases too. Also, an automated use case may become outdated by advances in technology, just as a software component can replace a domain agent of a use case. For example, paying by cheque is increasingly rare and even paying by credit card is being replaced by apps on mobile phones.

Component-Based Software Architecture Model. With the view of evolutionary automation, we create evermore models of software components for domain processes and refine existing models of software components. We integrate the software components in ways that allow the interactions among the software components to represent the interaction among the agents who previously handled the use cases in the domain model. This gives us a component-based structure model of the software system. For example, with a number of iterations and refinements, we have created a model for the *Point of Sale* software system. Its component diagram is given in Fig. 9. The interfaces of this integrated software system are use for interactions with the domain system.

Considering the functionality only, we define a model of software architecture as an integrated set of models of software components in which all the five view models of each component as defined earlier in this section, together with their interface operations appropriately (re-)named for the integration. Hence, the architecture is defined syntactically and with a rich semantics.

Requirements Modelling for Autonomous Monitoring Software Systems. Now consider the requirements modelling for an autonomous monitoring software system, such as an artificial pacemaker. A system like this is required in a domain system, where some components may exhibit abnormal or faulty behaviour. The software system is to correct or adjust the behaviour of a faulty behaviour or even to work in its place if the faulty component stops working.

One important characteristic of a faulty component is that faults occur inside the component. The occurrence of a fault may not be observed through the interfaces of the component at the time (or within a certain period of time) of its happening. However, if an error caused by an occurrence of the fault is not corrected or minor abnormal behaviour is not corrected in time, a *failure* of the component can happen, being observed to violate the requirements of the

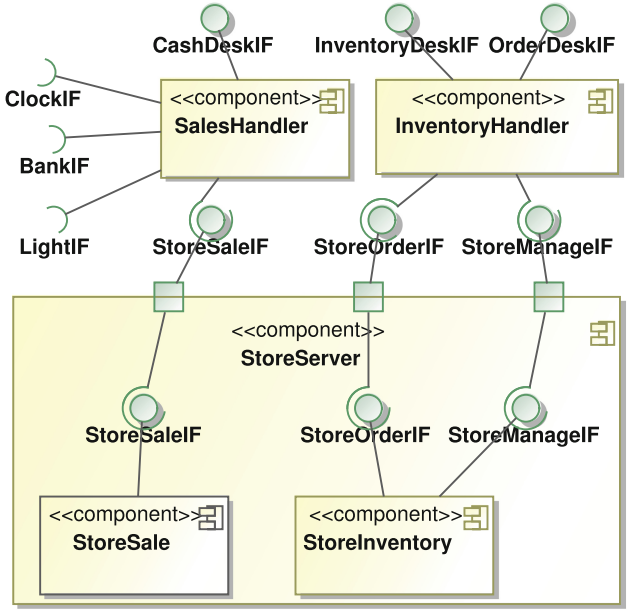

Fig. 9. A software component

component through its interface. Therefore, failures must be avoided to prevent them from spreading to other components and to the environment of the software system. A challenge in the design of a software system for such a domain is to how to detect occurrences of faults or abnormal behaviour where there are no existing interfaces for this.

Consider the design of a pacemaker in Example 2. Here, we have three models $H_{b/m}$, H_{ecg} of a heart H and a component-based model H_c, where we assume that $H_{b/m} \sqsubseteq H_{ecg} \sqsubseteq H_c$. Let us also make the assumption that there is a model for a healthy heart, which is the set of behaviours *HealthyH*.

Let AP be the model of the software system of an artificial pacemaker. We consider a pacemaker to work well if it is compatible with a heart modelled by M, where the model M can be any of $H_{b/m}$, H_{ecg}, and H_c. That is, when the pacemaker works together with the heart, the combined system behaves like a healthy heart. Formally, this means that for the composed model, $(AP \parallel M) \sqsubseteq$ *HealthyH*. By this, we mean that $(AP \| M)$ is a *fault-tolerant system* [66,67]. In general, we would like to have the property that if a heart is healthy under a more detailed model, it is also healthy under a coarser model, i.e.:

$$(AP \parallel H_{b/m}) \sqsubseteq (AP \parallel H_{ecg}) \sqsubseteq (AP \parallel H_c)$$

These algebraic properties do not alone provide enough of a specification for designing the software of a pacemaker. We need a model to describe what inputs it receives and the outputs it generates. The requirement for a pacemaker is that

it generates a beat (i.e., delivers electrical pulses) when the heart does not beat when it needs to do so. This is the identification of the outputs. In general, the outputs are to correct faults. To identify inputs, however, is to "get" information about occurrences of faults of abnormalities, i.e., to detect problems.

Consider further the case of a pacemaker. We can use a sensor to detect activation of the tissue in the left atrium. When the sensor receives no signal about activation for a given period of time, it generates an electrical pulse and causes contraction of the ventricles. Therefore, the sensor acts as the input interface to the model AP. The interactions of AP with LA and LV are shown in the *component sequence diagram* Fig. 10, where s_1 representing activation of LA is sensed by the sensor. Thus, AP does not generate any output to LV and s_0 representing no activation of LA is sensed.

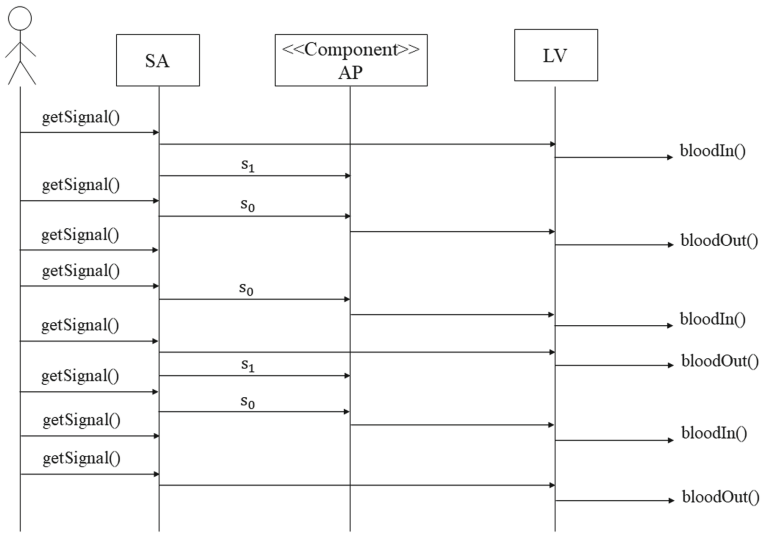

Fig. 10. Interactions between components PA, LA, and LV

Autonomous monitoring software components can also be designed to detect abnormalities of interactions between domain components. In such a case, we can take either of the following approaches:

- We model the protocol as a domain component first with a domain agent to handle the interactions. The monitoring software can then detect abnormalities, such as duplication, losses, corruptions, timing abnormality, etc.
- We take the components involved in the interactions as a composite whole. We then model the internal interfaces needed for detecting and correcting abnormalities as occurrences of internal faults.

Autonomous monitoring software components are usually embedded, physically or logically, within the faulty components in the domain. Therefore, including a

software model in the domain, does not change the domain structure: no domain agents are added or removed, but interaction interfaces among the domain agents are changed.

In general, we assume a domain $\mathcal{D}$, a model $\mathcal{C}$ of a part of $\mathcal{D}$ and its environment $\mathcal{E}$. We design a software system $\mathcal{S}$ for monitoring and controlling $\mathcal{C}$, to meet given requirements. Then the software modelling for $\mathcal{S}$ is to transform the domain model $\mathcal{D} = \mathcal{E} \parallel \mathcal{C}$ into a new model $\mathcal{D}' = \mathcal{E} \parallel \mathcal{C} \parallel \mathcal{S}$ that satisfies the requirements. For the artificial pacemaker example, AP is the model $\mathcal{S}$ in the new domain.

7.2 Human-Cyber-Physical Systems – A Unified Architecture View

From the discussion on domain modelling in Sect. 6 and that on software requirements modelling in this section, we can see both are cognitionive processes to obtain incremental and innovative understanding of the domain. For requirements capture and modelling of automation software systems, a use-case driven approach [21,54,63] is more effective and *use-case driven* requirements analysis is similar to, or developed from, Jackson's *problem frames* approach [47,48]. For requirements capture and modelling of autonomous monitoring software systems, it is more a combination a use-case driven approach and Parnas' *four-variable requirements analysis method* [81]. The architecture styles of both are *component-based* or *systems of systems*. Note that in an application domain, both kinds of software systems may be required and thus a combination of requirements analysis methods is needed.

For the development of a software system, either an automation software system or an autonomous monitoring software system, we incrementally transform the application domain by replacing domain processes with software processes or by adding software processes into the domain. These software processes operate with each other and with the domain processes in the transformed domain. It is easy to imagine that there are software processes in the original domain. In this case, software components can be replaced; software component can be monitored, controlled, and enhanced too. Therefore, in such a domain, human individuals, social organizations and systems, cyber-systems (i.e., software systems), and physical processes (such as a heart) interact and collaborate together to perform tasks in the system. Interactions and collaborations are realized through communication networks and interface devices, such as sensors and actuators. Furthermore, communications and interface devices can also be monitored, managed, and controlled by software processes. We thus call such a domain a *Human-Cyber-Physical System* (HCPS), as previously mentioned, and we can see that an HCPS is a continuously evolving system. Modelling an HCPS becomes a uniform modelling framework that combines traditional domain modelling and software requirements modelling.

The term HCPS has evolved from the concept of *Cyber-Physical Systems* (CPS), a type of system that has emerged over several decades [51,97]. We have not seen a reference architecture model for HCPS or any proposal for a systematic modelling framework HCPS architecture. What we propose is that

the model of the software systems should be considered in a unified approach to modelling an HCPS. Although there are challenging problems, both in theoretical foundation and in engineering methodology, we are mainly concerned about the interaction mechanisms among cyber-systems, physical processes, and human beings, as well as the impacts of these systems on each other.

8 Towards Architecture Modelling of Evolving HCPS

In general, a *Human-Cyber-Physical System* (HCPS) is a system in a particular domain that consists of human systems, cyber-systems and physical systems (processes). The cyber-systems interact and collaborate with the human systems to undertake living, social, business, and manufacturing tasks by using, controlling, and coordinating the physical systems. Control in an HCPS is dynamically shifted between humans and machines. An HCPS is normally a continuously evolving system where we are concerned with designing software systems for cyber-systems to maintain healthy evolution and to support beneficial evolution of an HCPS.

In this section, we discuss the nature of the component systems, especially the cyber-components required, in a general HCPS, We also argue why an engineering process and technology architecture for using formal methods is required and discuss what we mean by such an engineering process. We discuss why the unifying modelling of domain and software systems that we presented in Sect. 6 and Sect. 7 is desirable for HCPS evolution. Finally we present some research challenges in the design of software systems for HCPS.

8.1 Evolution from CPS to HCPS

The notion of HCPS has resulted due to an evolution from Cyber-Physical Systems (CPS), through integration with techniques of ubiquitous computing (also known as ambient environment), Internet of Things (IoT), big data, cloud computing, and artificial intelligence (AI), with an emphasis on human involvement in the system. CPS was formally proposed in 2006 at a workshop organized by the US National Science Foundation (NSF) [78]. Therefore, an HCPS represents the intersection of the technologies of traditional computation, communication and control, with the new technologies of IoT, big data, cloud computing, and AI. We now briefly discuss the three milestones (or three generations) in the development from CPS, through big data and cloud-based CPS, to HCPS.

Preliminary CPS. CPS was originally defined to refer networked cyber-systems and physical systems, in which:

- the *cyber-world* is formed by systems for computing, control and networking;
- the *physical world* includes mechanical, electrical, and chemical processes, etc.;
- the cyber-systems control the physical side using *sensors* and *actuators*;

- the *network* connects the sensor network, actuator network and the control units, and among the computing systems;
- a *database server* is needed to collect and process events generated in the system, those generated by the sensors, for computation of control decisions.

In such a system architecture, human users of the system are similar to users of traditional computer systems, but without being involved in control decision making. Important related technologies include communication networks, distributed computing and control, and network of sensors. Here, the concept of CPS mainly extends that of *embedded systems* in that the focus of embedded systems is on computational elements hosted in stand-alone devices, while CPS is designed as a network of interacting computational and physical devices. A CPS has many more features, including computations, control, and cooperation that need shared knowledge and information from physical systems to be responsiveness and self-adaptive.

We can see that a joint model of the physical world and the software systems in the cyber-world fits in our modelling framework discussed in Sect. 6 and Sect. 7. However, the challenge lies in the model of interactions and dynamic behaviour of the components in order for the whole system to have the required features. This including difficulties in modelling, such as:

- *communication:* how can the network realize the system-wide properties, such as timeliness,
- *control:* distributed and decentralized control applications are still challenging,
- *hybrid and mobile interactions*: modelling interactions between cyber and physical systems, some of them are mobile, and
- *heterogeneity:* integrated modelling of timing and concurrency of cyber-systems and physical processes, and interoperable composition and integration of models and software components, which are developed by different developers using multiple formalisms and tools.

The focus of a preliminary CPS is very much in collaborative control systems.

Big Data and Cloud-Based CPS. With the technological enabling power of CPS, sizeable and complex applications, such as power grids, industry manufacturing, and traffic control, have kept increasing in multiple dimensions. Different types of networks, including wired and wireless, internet and radio, local and global networks, etc., are used in such a CPS. Therefore, the networks in CPS can encompass the full scope of IoT. Also, a large amount of sensors and actuators, hardware and software components, and physical systems, which are developed and owned by different stakeholders, are managed, monitored, and controlled within the system. During the operation of such a CPS, many events are generated. Collecting and processing the data concerning these events are the basis for effective, reliable, and real-time management and control of the physical processes, hardware, software, and communication networks.

A *data server* is typically responsible for storing and processing the data in a CPS. The control system requests services from the data server according to feedback information or instructions from the users. It then generates control decisions. Data being generated in a complex CPS typically has the properties of extremely large *volume*, high *velocity*, and wide *variety* (i.e., different sources of data). With the huge amount and a large variety of data to be collected in a CPS, storing and processing such a massive amount data requires a technology that is "beyond the ability of typical database software tools to capture, store, manage and analyze" [71]. The solution lies in *big data technology*. However, big data analytics for CPS is not only offline processing of historic data, but more importantly it has to process real-time data and produce responses to events in real-time.

Data processing and analytic for CPS is significantly *valuable* (the fourth "v" of big data after volume, velocity, and variety), which is important in the following aspects:

- First of all, we can design software for data analysis and use AI, based on the large volume of sensors data and system execution data within a CPS, to realize effective, precise and real-time control, and collaboration of physical processes and software components.
- Secondly, the large volume of sensors data and system execution data in a CPS can be used to design and implement software components for monitoring, detecting, and handling abnormalities and faults in the system for fault-tolerance, resilience, and robustness.
- Thirdly, big data analysis is beneficial in designing software components for self-adaption, self-definition for managing the complex dynamic uncertainties in physical environments, and the communication network, and helping to ensure system predictivity, adaptivity, autonomy, reliability, and security.
- With the big data generated in a CPS, more value-added applications and services can be designed and implemented. Consider a lighting system of a city, for example. The data generated by this system can be used to develop services for the city management body, for the police authorities, and for the electricity company's business management.

However, to process and analyse CPS big data effectively, to synthesize information and generate knowledge for smart decision making, and for intelligent and flexible control, we require *cloud computing* and both rule based and machine learning based *artificial intelligence* (AI). The need for cloud computing is also due to the variety of the stakeholders and owners of the data and infrastructures within the CPS. A data server in the cloud becomes the obvious technology solution to providing an open and flexible platform for a variety of users to develop services on the CPS. With such a cloud platform, data analytic models and software can be provided to the control system of the CPS. With a cloud platform, all devices, hardware and software resources, including sensors, actuators, computation resources, data, application software, etc., can be managed, controlled, and shared as services.

A cloud platform is naturally service oriented. Therefore, a big data and cloud-based CPS can effectively leverage *service oriented architecture* (SoA) style to form a CPS of multiple layers from CPSs of different application domains. In the area of industry manufacturing, in particular, there is a clear trend in combinations of customized and personalized design and production, effective coordination of product market analysis and planning, product development, product after-sales services, manufacturing processes, and manufacturing systems management. For example, with marketing analysis systems and after-sales service systems, market information and information about the product quality can be used for product planning, design, quality control and assurance in the production process (including the use of resources, equipment, techniques, and skills of staff), and product testing.

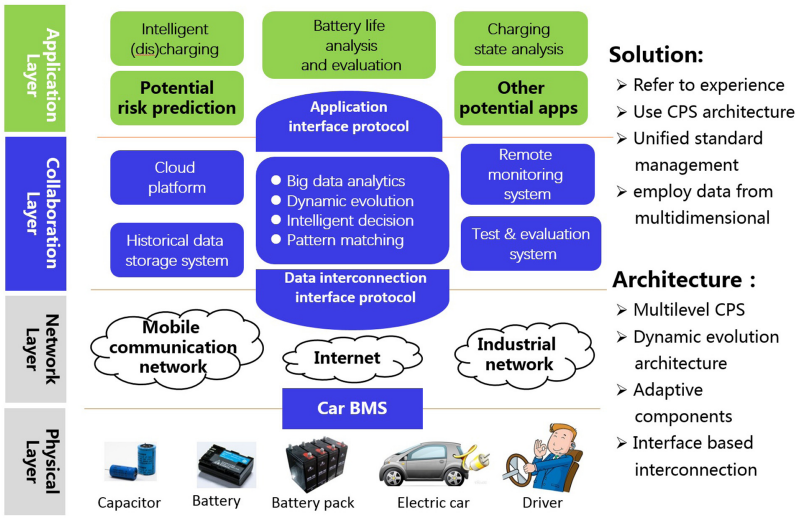

Fig. 11. A proposed architecture of cloud-based battery management CPS

Figure 11 shows a cloud-based CPS that we proposed in a project application for future car battery management. The architecture is organized into four layers. The bottom layer, called the physical layer, consists of car battery management systems and car systems. They are connected by the network layer to form an IoT. The network layer also transfers data collected in the IoT to the cloud computing layer for their storage management, analytics, information synthesis, and knowledge generation. These are to be used for computation, control decision making, and for the development of applications.

In addition to the challenges in modelling a preliminary CPS, there are more difficulties in modelling big data and a cloud-based CPS, among other problems. These includes the following issues:

- A big-data and cloud-based CPS requires software systems developed using different paradigms and architecture styles. These software systems are to interact via heterogenous networks. These pose huge problems of interoperability.
- A CPS is usually safety-critical, and thus its correctness, safety, and the real-time properties of its software components need to be formally provable, and verification of components is required to be compositional. However, there does not yet exists a theory for data-based AI software verification, refinement, and composition.
- Different processes and tasks share data, software services, physical infrastructures, and devices, as well traditional resources for programming execution. For efficient and real-time execution of these tasks concurrently, software abstractions are needed for the management and control of these shared heterogenous resources. Their capabilities and complexity are far beyond those of traditional operational systems. We do not have established models and techniques for these abstractions.

Furthermore, the issue of identifying desired features and quality of service of business, production, and social tasks, and deciding the relevant technologies for their realization. The state-of-the-art requirements modelling and design for CPS does provide full support for such a systematic definition of the technology architecture and process.

Human-Cyber-Physical Systems. The concept of HCPS emerged from two aspects of technology evolution. One is the evolution of integrating human-in-loop control within the control processes of a CPS. The other is the gradual integration of digital social media, social networks in big data technology, cloud computing for knowledge-based intelligent decision making, and the provision of smart services. The integration of these technologies has also been pushing forward with the overwhelming development of deep learning technology.

To consider human factors requires extending the cloud-based CPS architecture, such as the one in Fig. 11, by adding human agents or systems in each layer (except for the network layer). Such an extension then allows, and often requires, extension of the architecture horizontally for a number of HCPSs in different domains to be networked. For example, social systems may need to exchange knowledge about domains. An HCPS for farm machine manufacture, for example, is related to an HCPS for a farm. Extending the architecture horizontally typically leads to extension of the architecture upwards as well. Therefore, we envisage that an HCPS in general has an architecture horizontally and upwards with an open multiple three-layer *system of systems* (Open M3LSOS), as shown in Fig. 12. At the bottom layer are the *unit-level HCPSs* to be connected into the *system-level HCPS*, and the third level is the *system of systems-level HCPS*. Note that an HCPS at a particular level can be used abstractly as a unit to build an HCPS at higher level.

One of the most significant challenges, in addition to those of CPS and cloud-based CPS, is that we need mechanisms for modelling, detecting and possibly

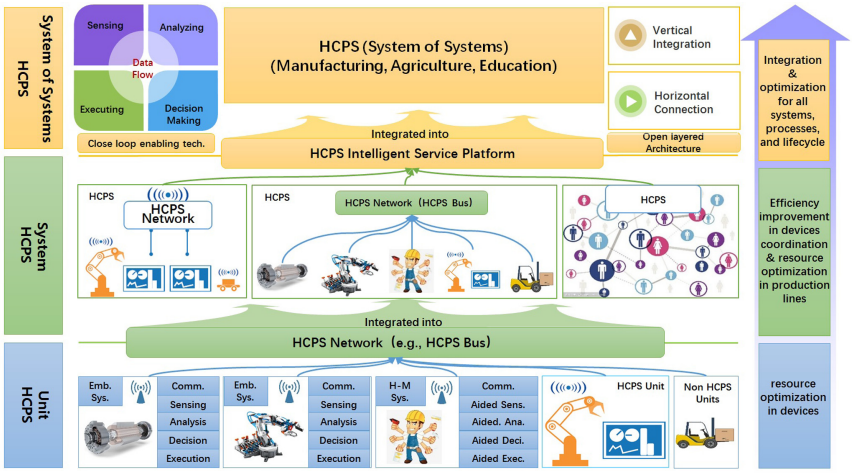

Fig. 12. A cloud-based CPS for battery management

predicting human behaviour, a model of the interactions between humans and cyber-systems, and the possible state changes that can be caused by human actions. We believe that advanced mathematical models and machine learning techniques are needed to help with these issues. Current state-of-the-art techniques are either very coarse and general or too application-specific, such as those of psychological computing, emotional computation, and brain modelling. The development of dynamic human behaviour models that are both accurate and general enough remains an enormous challenge.

8.2 Engineering Formal Methods for HCPS Development

Now we outline a framework for engineering formal methods in software development, software components for HCPSs in particular, in relation to the uniformed framework for domain modelling and software requirements modelling discussed in Sect. 6 and Sect. 7.

A General Model of HCPS. We first briefly summarize the following concepts, principles, and processes in domain modelling in Sect. 6 and software requirements modelling in Sect. 7, respectively:

– For a domain modelling:
  - identify processes and use cases with their actors, noting that some processes are autonomous and do not need an actor to execute;
  - identify the concepts and associations among the processes that involve in the execution of the processes, use cases, and tasks;
  - provide the models of the processes and use cases, their interfaces, together with the model of the domain concepts; and

  - integrate the models of the processes and use cases to form the domain model, denoted by $\mathcal{D}$.
- For software requirements modelling in an application domain:
  - identify the use cases that the software system is to automate and the processes that the software system is required to monitor;
  - produce software models for these processes and use cases, together with their conceptual class models (including data structures) and their interfaces with domains, the actors, or processes; and
  - integrate these software models together to form a software model with interfaces to the domain, denoted by $\mathcal{S}$.

We compose $\mathcal{D}$ and $\mathcal{S}$ in a way that replaces the use cases by the models of software components which automate the use cases. Then we add the models of the monitoring software components into the model $\mathcal{D}$. These software components will work together with the processes that are being monitored. We denote the composition as $\mathcal{D} \oplus \mathcal{S}$. Therefore, $\mathcal{D} \oplus \mathcal{S} = (\mathcal{D} - \mathcal{U}) \parallel \mathcal{S}$ if software component $\mathcal{S}$ automates the use cases in $\mathcal{U}$, and $\mathcal{D} \oplus \mathcal{S} = \mathcal{D} \parallel \mathcal{S}$ when software component $\mathcal{S}$ monitors and controls some processes in $\mathcal{D}$. We can see that in $\mathcal{D} \oplus \mathcal{S}$, the automation software component in $\mathcal{S}$ interacts with the actors of the use cases that it automates and the monitoring software in $\mathcal{S}$ interacts with the domain processes that it monitors. Since in general there are humans, cyber-systems, and physical processes interacting through communication networks, we can understand $\mathcal{D} \oplus \mathcal{S}$ as a model of an HCPS.

With model transformations, it is possible to organize $\mathcal{D} \oplus \mathcal{S}$ in a hierarchical component-based and layered architecture, as shown in Fig. 11 and Fig. 12. However, in practice this is only correct conceptually. We still have to solve the challenge of defining the models of humans and interactions between software components and human components before we are actually able to construct a model of an HCPS, as well as specifying and reasoning about its desired functionality and quality of service.

Engineering Formal Methods for HCPS Development. In Sect. 6 and Sect. 7, we discussed the issue that domain modelling and software requirements modelling are both cognitive processes involving informal and formal activities. In fact, a cognitive process cannot be totally formalized. Formal modelling is not necessarily creative or innovative. However, a formal model is beneficial in ensuring correctness and developing deeper insight about the object being modelled. It is the basis for further creative and innovative thinking. Thus, a formal approach is worthwhile in improved modelling for an HCPS.

One of the most significant purpose of formalization is, however, to stabilize the models as milestones in the modelling process. These stabilized formal models are used for formal treatment, such as refinement, integration, verification, simulation, and traceability checking. These formal activities are very important for further informal and creative modelling activities.

Next, we understand that, in general, both domain modelling and software requirements modelling require a variety of expertise and a number of modelling

languages. On the other hand, as we have shown in the formal methods review presented in Sect. 5, there is a large number of formal languages and methods. Typically there are several methods that are applicable for the same view or problem. For example, one can use either CSP or CCS for specifying concurrency and communication. Therefore, there is an issue of how to select the languages and methods to be used in the development.

In an HCPS, there is a large variety of software systems within different layers, including the following:

- Unit-level systems: embedded software, device drivers, and operating systems in sensors and devices;
- System level and system of systems (SOS) layer: control and monitoring/coordinating software, system software for resource management, big data processing and analytics, various kinds of AI software;
- Application layer: apps, web/cloud services, business and workflow management;
- Network layer: communication protocols, network infrastructure and resources management and scheduling, software defined networking (SDN).

Modelling and development of these software systems involves different software architecture styles and technologies, including object-oriented analysis (OOA), service-oriented architecture (SOA), model-driven architecture (MDA), artificial intelligence (AI), etc. These need different formal modelling languages and methods for requirements and design.

The modelling framework we propose is based on the conceptual architecture model in Fig. 12. It starts with software domain and software requirements as follows:

1. apply the domain modelling concepts and principles to build a domain model $\mathcal{D}$, i.e., an HCPS;
2. identify where in the domain model that software systems are required and what kind these will be, following the software requirements modelling concepts and principles;
3. define a process for the software requirements modelling according the nature of the software systems and the interactions with their environments and the expertise of modelling team; the modelling process includes when and what formal models should be produced, by who, and with what methods and tools;
4. build the software model $\mathcal{S}$ for the identified part of the domain in step 2, by the team following the process and using the methods and tools defined in step 3;
5. formalize and carry out verification, validation, and simulation for quality management for the safety-critical models produced in step 4;
6. take the model $\mathcal{D} \oplus \mathcal{S}$ as a new HCPS and repeat this process from step 2.

In each iteration of the above model development process, the HCPS can be extended and different HCPSs for different domains can be composed as a larger

HCPS. These extensions, together with above iterative process, decide the nature of the overall HCPS as it evolves.

With an architecture model of requirements, a process of design, implementation, and deployment, can be defined, either top-down or bottom-up. This demonstrates the importance of the architecture model in defining and managing the development process.

However, significant barriers to the realization of an HCPS development process are still numerous, including in particular modelling human behaviour, human-cyber interaction, composability, controllability and reusability of learning-based software systems.

8.3 Refinement and Evolution of HCPS Model

It is impossible to build a complex HCPS from scratch in one go. Instead, a practical HCPS is continuously evolving. For a given domain, we can think of a model HCPS for the domain at any given time as a result of evolution from the domain developing, incrementally adding an increasing number of software systems.

Given a model built in an iteration of the modelling process, as covered in the previous subsection, software components in this model, as for non-digital processes, can be treated as domain processes and can be:

- replaced by another software component that performs "better" than the original ones;
- monitored, controlled, and coordinated, to ensure that they perform better.

To ensure healthy evolution of an HCPS in this way, we need a notion of *refinement* of HCPS models. For an HCPS component $\mathcal{C}_1$ of an HCPS model $\mathcal{D}$, let P be a (desirable) *property* of $\mathcal{D}$. An HCPS component $\mathcal{C}_2$ is a *refinement* of $\mathcal{C}_1$ in $\mathcal{D}$ with respect to P, denoted by $\mathcal{C}_1 \sqsubseteq_P \mathcal{C}_2$, if $\mathcal{D}'$ satisfies P, where $\mathcal{D}'$ is obtained from $\mathcal{D}$ by replacing $\mathcal{C}_1$ with $\mathcal{C}_2$.

We do not have such a *partial order* $\sqsubseteq_P$ among HCPS components yet, but the notion of a *contract* [96] in an HCPS is believed to be important for developing such a refinement. With a philosophical intuition, given an HCPS component $\mathcal{C}$, a contract of $\mathcal{C}$ means " if the set of behaviours of its environment are assumed to be in the set A, then $\mathcal{C}$ guarantees to have it behaviours in G". This contract is written as $A \vdash G$. The meaning of this contract is defined to be $\neg A \cup G$, where $\neg A$ is the complement set of A.

For example, consider the contracts of two pacemakers p_1 and p_2 in the pacemaker example $C_1 = A_1 \vdash G_1$ and $C_2 = A_2 \vdash G_2$. Assume A_1 is the set of behaviours of LA such that the period of time between two contracts is no longer than 3 min; and A_2 is the set of behaviours of LA such that the period of time between two contracts is no longer than 5 min. So A_2 is a weaker assumption than A_1 as $A_1 \subseteq A_2$. If $G_2 \subseteq G_1$, that is G_2 is a stronger commitment than G_1, p_2 is a better pacemaker than p_1. This is because if p_1 works for a heart H, p_2 also works for heart. In general, a contract $C_2 = A_2 \vdash G_2$ is a *refinement* of

$C_1 = A_1 \vdash G_1$, denoted as $C_1 \sqsubseteq C_2$ if $A_1 \subseteq A_2$ and $G_2 \subseteq G_1$, intuitively meaning that latter commits a better service G_2 than G_1 with a weaker assumption (or requirements) A_2 than A_1 on the environment.

There are different definitions of contracts for different systems. For example, a triple $\{pre\}P\{post\}$ of a program P in Hoare Logic is a contract for P, and for a concurrent program P, Jones's *rely-guarantee triple* [24] $\{R\}P\{G\}$ is a contract of P. However, the theory of contracts of this form has yet to develop as a behaviour model of humans; heterogenous components and interactions do not exist.

Even though the formal theory of contracts for HCPSs does not exist, the way of thinking with a notion of a contract is useful in building models and in understanding the evolution of an HCPS. We can see that the following evolution can be applied effectively:

- develop and plug in new components into the HCPS;
- dynamically find and connect components in the HCPS;
- adding more interfaces and/or improving the performance of interfaces, allowing cyber-components to:
  - monitor more and better with respect to its environment;
  - be more autonomous (self-contained);
  - make more intelligent control decisions and provide smarter services;
  - control and coordinate more and better the associated physical components.

Finally, advances of HCPSs also rely on technology development in their application domains and related technologies such production of sensors and actuators. Therefore, rather than designing large HCPSs, it is better to keep evolving an HCPS to improve its optimization, smartness, connectivity, autonomy, and trustworthiness.

8.4 Conceptual Integrity and Domain System Architecture

In Sect. 6, Sect. 7, and this section, we have emphasized multi-view and incremental architecture modelling. Now we can draw the following conclusions:

1. In multi-view modelling, each view model represents certain aspects of the domain system. The *union* of the views gives the whole system.
2. Multi-view modelling is about separation of concern in modelling, dividing the problem of modelling or designing models into problems of modelling from different viewpoints.
3. Refinement of models is about incremental modelling. This can be applied to models of different views and to a group of models or even the models of the whole domain system.
4. Multi-view modelling requires the use of *domain-specific languages*.

We believe multi-view modelling with incremental refinement is helpful and a manageable approach to domain modelling. However, to define and ensure the

"integrated unity" of the system architecture from the view models and to maintain this "unity" through the incremental refinement modelling process is known to be a challenging problem in model-driven development [19]. We understand this unity is a major issue in the context of domain modelling[3], the intention of what is called "the conceptual integrity" by Brooks [11]:

> I will contend that conceptual integrity is the most important consideration in system design. It is better to have a system omit certain anomalous features and improvements, but to reflect one set of design ideas, than to have one that contains many good but independent and uncoordinated ideas... a critical need of large projects is that the conceptual integrity can be achieved throughout by chief architects, and implementation is achieved through well-managed effort the architects should develop an idea of what the system should do and make sure that this vision is understood by the rest of the team.

Brooks has not changed his mind. Indeed, he reiterated this in 1995, and again much later at the ICSE 2018 International Conference on Software Engineering [14].

In the context of multi-view and multi-level domain modelling framework, we develop an understanding of conceptual integrity with a domain system architecture view, including the following issues:

1. there is a need for a model of the conceptual view for each component of the system, called the conceptual model of the component;
2. there is a conceptual model of the whole system synthesized from those of the components, in which a concept is related to more than one component;
3. there is a system structure model that defines the partition of the system into subsystems (or components);
4. the subsystem boundaries must be at those places where interfaces between the subsystems are defined rigorously; and this is a defined in a model of each subsystem;
5. models must be defined for the interactions of the subsystems with their environments and their dynamic behaviour models, as well as the interactions of the system as whole and its dynamic behaviour (both black-box and white-box interactions, and behaviour if necessary);
6. the extensions, i.e., the instances (or objects) of the concepts and their relations in the conceptual models, must be clearly defined; and their should be consistency conditions among the concepts and relations across models;
7. the concepts and objects in the conceptual models and their consistency conditions must be consistent with the concepts and objects involved in the interactions and behaviour of the components, and among the components;

[3] The word "software" is not meant to be a restriction but in this paper we consider these ideas only within the context of software. Also, the lack of a formal definition for "conceptual integrity" has made it hard to apply this idea systematically and to understand it with better insight.

8. consistency conditions the dynamic models of the subsystems and those of whole system must be defined and ensured;
9. required constraints, both on statics structure models and on dynamic models, should be defined and ensured by the models.

Items 1–4 above model static aspects of the system; item 5 models dynamic aspects; items 6–9 are the required conditions for conceptual integrity, including the requirements in item 9. It is these conditions that are needed to make the models act in "unity".

Therefore, an architecture model of a domain system (e.g., an HCPS) at a given level of abstraction consists of the static structure model of the system described in item 3, the dynamic models described in item 5, and the conceptual models, with the consistency conditions and specification of required constraints.

Apart from the possible path of evolution for an HCPS discussed in Subsect. 8.3, a software architecture in general can be refined and extended in the following ways:

- adding a subsystem that:
  - extends the intention and extension (i.e., the set of instances) of the existing concepts and/or adds new concepts;
  - extends interactions with existing subsystems in the current model, preferably through the system interfaces of the current model; and
  - provides (value-added) interactions with the system environment, including functionalities and services; or improves functionalities and/or performance of the current model.
- refining an existing subsystem, through:
  - decomposing or adding more properties, behaviour, interactions, functionality, etc., according to new knowledge gained; and/or
  - decomposing some of its subcomponents (sub-subsystems).

For component-based design and analysis, we need to treat an architecture model hierarchically. Subsystems are organized in compositions. A subsystem has its own architecture that is the responsibility of its modeller (or architect). In each model refinement step, the conceptual integrity needs to be maintained. For this, traceability of changes and consistency checks are critical. There is a need to develop a framework of architectural modelling in which a set sound methods and tools can be applied in modelling processes. We discuss this in the next section together with software architecture modelling.

9 Concluding Remarks

This tutorial paper is assembled from a number lecture notes, with a mixture of:

- a historical summary about ideas of abstracts,
- a review of formal methods, and

– some thoughts about the relation between domain modelling and software requirements modelling.

The points which we wish to communicate include:

– using a seamless combination of informal and formal modelling, as well as maintaining the domain model as a whole, software system development is modifying the domain model for an HCPS, and then continuing its evolution;
– software system development, for an HCPS in particular, requires the use of multiple formal methods; and normally there is more than one method needed for a particular task;
– the importance of architecture modelling for keeping conceptual integrity as well as for development process definition and management;
– the shift of HCPS system design to system evolution, taking an existing HCPS as an infrastructure for the development new systems and services to extend the original system.

We also discussed the concepts of an HCPS as an integration of a number of recent technologies. The development of an HCPS involves all areas of ICT and its application domain. We have proposed a conceptual architecture model, which has been used to identify some remaining challenges. The architecture defines an HCPS as an open layered system of systems. Finally, we provide a quotation attributed to the pioneer computer scientist David Wheeler FRS (1927–2004), also used by Butler Lampson, on layers of indirection:

> All problems in computer science can be solved by another layer of indirection. But that usually will create another problem.
>
> David J. Wheeler [76]

Acknowledgements. Thank you for project support by the National Natural Science Foundation of China (61802318, 61732019, 61672435, 61811530327), the Capacity Development Grant of Southwest University (SWU116007), and the Natural Science Foundation of Chongqing (cstc2017jcyjAX0295). Thank you to Museophile Limited for financial support for Jonathan Bowen.

References

1. Abrial, J.R.: The B-Book: Assigning Programs to Meanings. Cambridge University Press, Cambridge (1996)
2. Ambler, S.: The Agile Unified Process (AUP). http://www.ambysoft.com/unifiedprocess/agileUP.html
3. Back, R.J.: On the correctness of refinement steps in program, development. Ph.D. thesis, University of Helsinki, Finland (1978)
4. Bergstra, J.A., Tucker, V.J.: Expressiveness and the completeness of Hoare's logic. J. Comput. Syst. Sci. **25**(3), 267–284 (1982). https://doi.org/10.1016/0022-0000(82)90013-7

5. Berry, G., Gonthier, G.: The Esterel synchronous programming language: design, semantics, implementation. Sci. Comput. Program. **19**(2), 87–152 (1992). https://doi.org/10.1016/0167-6423(92)90005-V
6. Bjørner, N.: The Z3 theorem prover. GitHub. https://github.com/Z3Prover/z3
7. Boehm, B.W.: A spiral model of software development and enhancement. IEEE Comput. **21**(5), 61–72 (1988). https://doi.org/10.1109/2.59
8. Booch, G.: Object-Oriented Analysis and Design with Applications. Addison-Wesley, Boston (1994)
9. Bowen, J.P.: The Z notation: whence the cause and whither the course? In: Liu, Zhang [68], pp. 103–151. https://doi.org/10.1007/978-3-319-29628-9_3
10. Bowen, J.P., Hinchey, M.G.: Formal methods. In: Gonzalez, T.F., Díaz-Herrera, J., Tucker, A.B. (eds.) Computing Handbook. Computer Science and Software Engineering, 3rd edn, pp. 1–25. Chapman and Hall/CRC Press, Boca Raton (2014). https://doi.org/10.1201/b16812. Section XI, Software Engineering, Part 8, Programming Languages
11. Brooks, F.P.: The Mythical Man-Month: Essays on Software Engineering. Addison-Wesley, Boston (1975)
12. Brooks, F.P.: No silver bullet: essence and accidents of software engineering. IEEE Comput. **20**(4), 10–19 (1987). https://doi.org/10.1109/MC.1987.1663532
13. Brooks, F.P.: The mythical man-month: after 20 years. IEEE Softw. **12**(5), 57–60 (1995). https://doi.org/10.5555/624609.625509
14. Brooks, F.P.: Learn the hard way - a history 1845–1980 of software engineering. In: Keynote at 40th International Conference on Software Engineering (ICSE), Gothenburg, Sweden (2018). https://www.icse2018.org/info/keynotes
15. Broy, M., Stefanescu, G.: The algebra of stream processing functions. Theoret. Comput. Sci. **258**(1–2), 99–129 (2001). https://doi.org/10.1016/S0304-3975(99)00322-9
16. Broy, M., Wirsing, M.: On the algebraic extensions of abstract data types. In: Díaz, J., Ramos, I. (eds.) ICFPC 1981. LNCS, vol. 107, pp. 244–251. Springer, Heidelberg (1981). https://doi.org/10.1007/3-540-10699-5_101
17. Chandy, K.M., Misra, J.: Parallel Program Design: A Foundation. Addison-Wesley, Reading (1988)
18. Chen, L., Babar, M.A., Nuseibeh, B.: Characterizing architecturally significant requirements. IEEE Softw. **30**(2), 38–45 (2013). https://doi.org/10.1109/MS.2012.174
19. Chen, X., Liu, Z., Mencl, V.: Separation of concerns and consistent integration in requirements modelling. In: van Leeuwen, J., Italiano, G.F., van der Hoek, W., Meinel, C., Sack, H., Plášil, F. (eds.) SOFSEM 2007. LNCS, vol. 4362, pp. 819–831. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-69507-3_71
20. Chen, Z., et al.: Modelling with relational calculus of object and component systems - rCOS. In: Rausch, A., Reussner, R., Mirandola, R., Plášil, F. (eds.) The Common Component Modeling Example. LNCS, vol. 5153, pp. 116–145. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-85289-6_6
21. 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). https://doi.org/10.1016/j.scico.2008.08.003
22. Clarke, E.M., Emerson, E.A.: Design and synthesis of synchronization skeletons using branching time temporal logic. In: Kozen, D. (ed.) Logic of Programs 1981. LNCS, vol. 131, pp. 52–71. Springer, Heidelberg (1982). https://doi.org/10.1007/BFb0025774

23. Codd, E.F., Lowry, E.S., McDonough, E., Scalzi, C.A.: Multiprogramming STRECH: feasibility consideration. Commun. ACM **2**(11) (1959). https://doi.org/10.1145/368481.368502
24. Collette, P., Jones, C.B.: Enhancing the tractability of rely/guarantee specifications in the development of interfering operations. In: Plotkin, G.D., Stirling, C.P., Tofte, M. (eds.) Proof, Language, and Interaction: Essays in Honour of Robin Milner, pp. 277–308. The MIT Press, Cambridge (2000)
25. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Proceedings of Fourth ACM Symposium on Principles of Programming Languages, pp. 238–252. ACM Press, Los Angeles (1977)
26. Pigott, D.: Online historical encyclopaedia of programming languages (2020). http://hopl.info
27. de Alfaro, L., Henzinger, T.A.: Interface automata. ACM SIGSOFT Softw. Eng. Notes **26**(5) (2001). https://doi.org/10.1145/503271.503226
28. Dijkstra, E.: Guarded commands, non-determinacy and formal derivation of programs. Commun. ACM **18**(8), 453–457 (1975). https://doi.org/10.1145/360933.360975
29. Dijkstra, E.W.: A Discipline of Programming. Prentice Hall, Upper saddle River (1976)
30. Dijkstra, E.W.: The humble programmer. Commun. ACM **15**(10), 859–866 (1972). https://doi.org/10.1145/355604.361591. An ACM Turing Award lecture
31. Floyd, R.W.: Assigning meanings to programs. In: Schwartz, J.T. (ed.) Mathematical Aspects of Computer Science. Proceedings of Symposium on Applied Mathematics, vol. 19, pp. 19–32. American Mathematical Society (1967). https://doi.org/10.1007/978-94-011-1793-7_4. Republished in Program Verification (1993)
32. Forsberg, K., Mooz, H.: The relationship of system engineering to the project cycle. In: Proceedings of the First Annual Symposium of National Council on System Engineering, pp. 57–65, October 1991
33. Giloi, W.K.: Konrad Zuse's Plankalkül: the first high-level, "non von Neumann" programming language. IEEE Ann. Hist. Comput. **19**, 17–24 (1997). https://doi.org/10.1109/85.586068
34. Goguen, J.A.: Higher-order functions considered unnecessary for higher-order programming. In: Research Topics in Functional Programming. Programming Research Group, Oxford University (1987)
35. Grattarola, F.: Margaret Hamilton - coding to the moon. In: A Computer of One's Own, December 2018. https://medium.com/a-computer-of-ones-own/margaret-hamilton-coding-to-the-moon-6ba70b7e6b43
36. Guttag, J.V., Horning, J.J.: Larch: Languages and Tools for Formal Specification. Springer, New York (1993). https://doi.org/10.1007/978-1-4612-2704-5
37. Halbwachs, N., Caspi, P., Raymond, P., Pilanud, D.: The synchronous data flow programming language LUSTRE. Proc. IEEE **79**(9), 1305–1320 (1991). https://doi.org/10.1109/5.97300
38. Hamilton, M.H.: The language as a software engineer. In: keynote at 40th International Conference on Software Engineering (ICSE), Gothenburg, Sweden (2018). https://www.icse2018.org/info/keynotes
39. Harel, D.: Statecharts: a visual formalism for complex systems. Sci. Comput. Program. **8**(3), 231–274 (1987). https://doi.org/10.1016/0167-6423(87)90035-9

40. He, J., Li, X., Liu, Z.: Component-based software engineering. In: Hung, D.V., Wirsing, M. (eds.) Theoretical Aspects of Computing. LNCS, vol. 3722, pp. 70–95. Springer, Hanoi (2005). https://doi.org/10.1007/11560647_5. UNU-IIST TR 330
41. He, J., Liu, Z., Li, X.: rCOS: a refinement calculus of object systems. Theoret. Comput. Sci. **365**(1–2), 109–142 (2006). https://doi.org/10.1016/j.tcs.2006.07.034
42. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM **12**(10), 576–580 (1969). https://doi.org/10.1145/363235.363259
43. Hoare, C.A.R.: Communicating sequential processes. Commun. ACM **21**(8), 666–677 (1978). https://doi.org/10.1145/359576.359585
44. Hoare, C.A.R., He, J.: Unifying Theories of Programming. International Series in Computer Science. Prentice Hall, Upper Saddle River (1998)
45. IEEE: SWEBOK V3.0: software engineering body of knowledge. IEEE Computer Society (2014). http://www.swebok.org
46. Jackson, M.A.: Principles of Program Design. Academic, Cambridge (1975)
47. Jackson, M.: Software Requirements & Specifications: A Lexicon of Practice, Principles and Prejudices. ACM Press/Addison-Wesley Publishing, Boston (1995)
48. Jackson, M.: Problem Frames: Analysing and Structuring Software Development Problems. Addison-Wesley, Boston (2001)
49. Jiang, Z., Pajic, M., Moarref, S., Alur, R., Mangharam, R.: Modeling and verification of a dual chamber implantable pacemaker. In: Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 188–203. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-28756-5_14
50. Jones, C.B.: Systematic Software Development Using VDM. International Series in Computer Science. Prentice Hall, Upper Saddle River (1990)
51. Khaitan, S.K., McCalley, J.D.: Design techniques and applications of cyberphysical systems: a survey. IEEE Syst. J. **9**(2), 350–360 (2014)
52. Kroll, P., Kruchten, P.: The Rational Unified Process Made Easy: A Practitioner's Guide to the RUP. Addison-Wesley, Boston (2003)
53. Lamport, L.: The temporal logic of actions. ACM Trans. Program. Lang. Syst. **16**(3), 872–923 (1994). https://doi.org/10.1145/177492.177726
54. 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)
55. Laxsen, K.G., Pettersson, P., Yi, W.: Diagnostic model-checking for real-time systems. In: Alur, R., Henzinger, T.A., Sontag, E.D. (eds.) HS 1995. LNCS, vol. 1066, pp. 575–586. Springer, Heidelberg (1996). https://doi.org/10.1007/BFb0020977
56. Leavens, G.T., Baker, A.L.: Enhancing the pre- and postcondition technique for more expressive specifications. In: Wing, J.M., Woodcock, J., Davies, J. (eds.) FM 1999. LNCS, vol. 1709, pp. 1087–1106. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-48118-4_8
57. Lee, E.A.: The past, present and future of cyber-physical systems: a focus on models. Sensors **1**(3), 4837–4869 (2015). https://doi.org/10.3390/s150304837
58. Leondes, C.T.: Intelligent Systems: Technology and Applications. CRC Press, Boca Raton (2002)
59. Lindsey, C.H., Boom, H.J.: A modules and separate compilation facility for ALGOL 68. ALGOL Bull. **43** (1978). https://doi.org/10.5555/1061719.1061724
60. Liskov, B., Zilles, S.: Programming with abstract data types. SIGPLAN Not. **9**, 50–59 (1974). https://doi.org/10.1145/942572.807045. In: Proceedings of the ACM SIGPLAN Symposium on Very High Level Languages

61. Liu, Z.: Software development with UML. Technical report 259, UNU-IIST: International Institute for Software Technology, United Nations University, Macau (2002)
62. Liu, Z.: Fault-tolerant programming by transformations. Ph.D. thesis, University of Warwick, UK (1991)
63. Liu, Z., Chen, X.: Model-driven design of object and component systems. In: Liu, Zhang [68], pp. 152–255. https://doi.org/10.1007/978-3-319-29628-9_4
64. Liu, Z., Jifeng, H., Li, X.: rCOS: refinement of component and object systems. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2004. LNCS, vol. 3657, pp. 183–221. Springer, Heidelberg (2005). https://doi.org/10.1007/11561163_9
65. Liu, Z., Jifeng, H., Li, X., Chen, Y.: A relational model for formal object-oriented requirement analysis in UML. In: Dong, J.S., Woodcock, J. (eds.) ICFEM 2003. LNCS, vol. 2885, pp. 641–664. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-39893-6_36
66. Liu, Z., Joseph, M.: Transformation of programs for fault-tolerance. Formal Aspects Comput. **4**(5), 442–469 (1992). https://doi.org/10.1007/BF01211393
67. Liu, Z., Joseph, M.: Specification and verification of fault-tolerance, timing, and scheduling. ACM Trans. Program. Lang. Syst. **21**(1), 46–89 (1999). https://doi.org/10.1145/314602.314605
68. Liu, Z., Zhang, Z. (eds.): Engineering Trustworthy Software Systems. LNCS, vol. 9506. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-29628-9
69. Lynch, N.A., Tuttle, M.R.: Hierarchical correctness proofs for distributed algorithms. In: Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing (PODC 1987), pp. 137–151, August 1987. https://doi.org/10.1145/41840.41852
70. Manna, Z., Waldinger, R.: A deductive approach to program synthesis. ACM Trans. Program. Lang. Syst. **2**, 90–121 (1980). https://doi.org/10.1145/357084.357090
71. Manyika, J.: Big data: the next frontier for innovation, competition, and productivity (2011). http://www.mckinsey.com/insights/business_technology/big_data_the_next_frontier_for_innovation
72. Mauchly, J.W.: Preparation of problems for EDVAC-type machines (1947). In: Randell, B. (ed.) The Origins of Digital Computers. MCS. Springer, Heidelberg (1982). https://doi.org/10.1007/978-3-642-61812-3_31
73. McCarthy, J.: Towards a mathematical science of computation. In: IFIP Congress, pp. 21–28. IFIP (1962)
74. Mills, H.: Top-down programming in large systems. In: Ruskin, R. (ed.) Debugging Techniques in Large Systems. Prentice Hall, Eaglewood Cliffs (1971)
75. Milner, R.: A Calculus of Communicating Systems. Springer, Heidelberg (1980). https://doi.org/10.1007/3-540-10235-3
76. Murray, D., Fraser, K.: Xen and the beauty of virtualization. In: Spinellis, D., Gousios, G. (eds.) Beautiful Architecture: Leading Thinkers Reveal the Hidden Beauty in Software Design, p. 172. O'Reilly Media, Newton (2009)
77. Naur, P., Randell, B. (eds.): Software Engineering: Report of a Conference Sponsored by the NATO Science Committee, Garmisch, Germany, 7–11 October 1968, Brussels, Scientific Affairs Division, NATO. NATO, January 1969
78. NSF: Workshop on cyber-physical systems, Austin, Texas, 16–17 October 2006. https://cps-vo.org/node/179
79. Nygaard, K., Dahl, O.J.: The development of the SIMULA languages. ACM SIGPLAN Not. **13**(8), 439–480 (1978). https://doi.org/10.1145/960118.808391

80. Parnas, D.L.: On the criteria to be used in decomposing systems into modules. Commun. ACM **15**(12), 1053–1058 (1972). https://doi.org/10.1145/361598.361623
81. Parnas, D.L., Madey, J.: Functional decomposition for computer systems. Sci. Comput. Program. **25**(1), 41–61 (1995). https://doi.org/10.1016/0167-6423(95)96871-J
82. Paul, C., et al.: Documenting Software Architectures: Views and Beyond, 2nd edn. Addison-Wesley, Boston (2010)
83. Perry, D.E., Wolf, A.L.: Foundations for the study of software architecture. ACM SIGSOFT Softw. Eng. Notes **17**(4), 40–52 (1992). https://doi.org/10.1145/141874.141884
84. Petri, C.A., Reisig, W.: Petri net. Scholarpedia **3**(4), 6477 (2008). https://doi.org/10.4249/scholarpedia.6477
85. Plotkin, G.D.: The origins of structural operational semantics. J. Logic Algebraic Program. **60–61**, 3–15 (2004). https://doi.org/10.1016/j.jlap.2004.03.009
86. Pnueli, A.: The temporal logic of programs. In: Proceedings of the 18th Annual Symposium on Foundations of Computer Science SFCS 1977, pp. 46–57. IEEE, September 1977. https://doi.org/10.1109/SFCS.1977.32
87. Queille, J.P., Sifakis, J.: Specification and verification of concurrent systems in CESAR. In: Dezani-Ciancaglini, M., Montanari, U. (eds.) Programming 1982. LNCS, vol. 137, pp. 337–351. Springer, Heidelberg (1982). https://doi.org/10.1007/3-540-11494-7_22
88. Randell, B.: System structure for software fault tolerance. IEEE Trans. Softw. Eng. **22**, 220–232 (1975). https://doi.org/10.1109/TSE.1975.6312842
89. Randell, B.: Position statement: how far have we come? In: Proceedings of the 32nd Annual IEEE International Computer Software and Applications Conference, COMPSAC 2008, 28 July–1 August 2008, Turku, Finland, p. 8. IEEE, IEEE Computer Society (2008). https://doi.org/10.1109/COMPSAC.2008.233
90. Randell, B.: Fifty years of software engineering or the view from Garmisch. In: Keynote at 40th International Conference on Software Engineering (ICSE), Gothenburg, Sweden (2018). https://www.icse2018.org/info/keynotes
91. Randell, B., Buxton, J.N. (eds.): Software engineering: report of a conference sponsored by the NATO science committee, Rome, Italy, 27–31 October 1969, Brussels, Scientific Affairs Division, NATO. NATO (1969)
92. Rausch, A., Reussner, R., Mirandola, R., Plášil, F. (eds.): The Common Component Modeling Example. LNCS, vol. 5153. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-85289-6
93. Roscoe, A.W.: Theory and Practice of Concurrency. International Series in Computer Science. Prentice Hall, Upper Saddle River (1997)
94. Royce, W.W.: Managing the development of large software systems. In: Proceedings of IEEE WESCON, pp. 1–9. IEEE (1970). https://doi.org/10.5555/41765.41801. Reprinted in ICSE (1987)
95. Ryckman, G.F.: 17. The IBM 701 computer at the general motors research laboratories. Ann. History Comput. **5**(12), 210–212 (1983). https://doi.org/10.1109/MAHC.1983.10026
96. Sangiovanni-Vincentelli, A., Damm, W., Passerone, R.: Taming Dr. Frankenstein: contract-based design for cyber-physical systems. Eur. J. Control **18**(3), 217–238 (2012). https://doi.org/10.3166/ejc.18.217-238
97. Schlingloff, B.H.: Cyber-physical systems engineering. In: Liu, Zhang [68], pp. 256–289. https://doi.org/10.1007/978-3-319-29628-9_5

98. Scott, D., Strachey, C.: Toward a mathematical semantics for computer languages. In: Technical Monograph PRG-6, Programming Research Group, Oxford University (1971)
99. Sommerville, I.: Software Engineering, 10th edn. Pearson, Upper Saddle River (2016)
100. Spivey, J.M.: The Z Notation: A Reference Manual. International Series in Computer Science, 2nd edn. Prentice Hall, Upper Saddle River (1992)
101. Stoy, J.E.: Denotational Semantics: The Scott-Strachey Approach to Programming Language Semantics. The MIT Press, Cambridge (1977)
102. Turing, A.M.: Checking a large routine. In: Report of a Conference on High Speed Automatic Calculating Machines, pp. 67–69. Cambridge University Mathematical Laboratory (1949). https://doi.org/10.5555/94938.94952. Reprinted in The Early British Computer Conferences (1989)
103. von Neumann, J.: Introduction to "the first draft report on the edvac". Archive.org. (1945). https://web.archive.org/web/20130314123032/http://qss.stanford.edu/~godfrey/vonNeumann/vnedvac.pdf
104. Wang, J., Zhan, N., Feng, X., Liu, Z.: Overview of formal methods. Ruan Jian Xue Bao/J. Softw. **30**(1), 33–61 (2019). (in Chinese)
105. West, D.: Hermeneutic computer science. Commun. ACM **40**(4) (1997). https://doi.org/10.1145/248448.248467
106. Wheeler, D.J.: The use of sub-routines in programmes. In: Proceedings of the 1952 ACM National Meeting, p. 235. ACM, Pittsburgh, USA (1952). https://doi.org/10.1145/609784.609816
107. Wilkes, M.V., Wheeler, D.J., Gill, S.: Preparation of Programs for an Electronic Digital Computer. Addison-Wesley, Boston (1951)
108. Wirth, N.: Program development by stepwise refinement. Commun. ACM **14**(4), 221–227 (1971). https://doi.org/10.1145/362575.362577
109. Woodcock, J., Larsen, P.G., Bicarregui, J., Fitzgerald, J.: Formal methods: practive and experience. ACM Comput. Surv. **41**(4), 19:1–19:36 (2009). https://doi.org/10.1145/1592434.1592436