# Chapter 1
# Multi-Paradigm Modelling for Cyber-Physical Systems: Foundations

Paulo Carreira, Vasco Amaral, and Hans Vangheluwe

**Abstract** Modeling and analysis of Cyber-Physical Systems (CPS) is an inherently multi-disciplinary endeavour. Anyone starting in this field will unavoidably face the need for a literature reference that delivers solid foundations. Although, in specific disciplines, many techniques are used already as a matter of standard practice, their fundamentals and application are typically far from practitioners of another area. Overall, practitioners tend to use the technique that they are most familiar with, disregarding others that would be adequate for the problem at hand. The inherent cross-disciplinary nature of CPS requires distinct modelling techniques to be employed, thus prompting for a common background formalism that enables communication between all specialities. However, to this date, no such single super-formalism exists to support the multiple dimensions of the design of a CPS. Indeed, to effectively design a CPS, engineers (in the role of modellers) either need to be versed in multiple formalisms, or a fundamentally new modelling approach has to emerge. Herein, we motivate Multi-Paradigm Modelling of CPS (MPM4CPS), introducing fundamental definitions and terminology regarding CPS modelling and Multi-Paradigm, and finally, laying the ground for the rest of the book.

## 1.1 Introduction

Cyber-Physical Systems (CPS) refer to systems that consist of *cyber* (as computerised implementations) and *physical* components [130]. The general idea is that the cyber and physical components influence each other in such way that the cyber is able to cause the physical component to change state, and that the change, in turn, will feed-back, resulting in a change of state on the cyber component.

Having emerged from earlier concepts, among other, in the fields of mechatronics, embedded systems, and cybernetics, literature gives the coining of the term 'Cyber-Physical System' (CPS) to Hellen Guille in 2006 [130]. CPS are often regarded as networks of multi-physical (mechanical, electrical, biochemical, etc) and computational (control, signal processing, logical inference, planning, etc) processes, often interacting with a highly uncertain and adverse environment, including human actors and other CPS.

Example application domains of CPS include energy conservation, environmental control, avionics, critical infrastructure control (electric power, water resources, and communications systems), high confidence medical devices and systems, traffic control and safety, advanced automotive systems, process control, distributed robotics (telepresence, telemedicine), manufacturing, and smart city engineering. The design of CPS is currently a driver

Paulo Carreira
Instituto Superior Técnico, Universidade de Lisbon, Portugal
e-mail: paulo.carreira@tecnico.ulisboa.pt

Vasco Amaral
FCT, Universidade NOVA de Lisboa, Portugal
e-mail: vma@fct.unl.pt

Hans Vangheluwe
McGill University, Canada
e-mail: hans.vangheluwe@uantwerp.be

P. Carreira et al. (eds.), *Foundations of Multi-Paradigm Modelling for Cyber-Physical Systems*,
https://doi.org/10.1007/978-3-030-43946-0_1

for innovation across various industries, creating entirely new markets. More efficient and cheaper CPS will have a positive economic impact on any one of these applications areas.

CPS are notoriously complex to design and implement mostly because of their cross-discipline borders, leading to inter-domain interactions, in applications that are often safety-critical. Indeed, due to the nature of their application, failure or underperformance of CPS can have direct, measurable economic costs, can harm the environment or even directly affect humans. Expectably, engineering practice has, over the years, sought to address these concerns by improving the languages, frameworks, and tools used in the design and analysis of CPS. This effort led to the emergence and strong adoption of model-based design, in which systems are designed at a higher level of abstraction, and an implementation is then produced by automatic generation.

A striking aspect of CPS design is that it is inherently multi-disciplinary. One source of this multi-disciplinarity arises from the domain of the application itself, such as e.g., medical, biological, or aeronautical industries. Another source is the heterogeneous nature of CPS, which consists of computerised, electronic, and mechanic parts. To design a CPS, engineers from various disciplines need to explore system designs collaboratively, to agree, to allocate responsibilities to software and physical elements, and to analyse trade-offs between them.

Originating from the Modelling and Simulation Community, the term *Multi-Paradigm-Modelling* (MPM) finds its origin in 1996, when the EU ESPRIT Basic Research Working Group 8467 formulated a series of simulation policy guidelines [286] identifying the need for *"a multi-paradigm methodology to express model knowledge using a blend of different abstract representations rather than inventing some new super-paradigm"*, and later on proposing a methodology focusing on combining multiple formalisms [294]. Since 2004, during the yearly Computer Automated Multi-Paradigm Modelling (CAMPaM) Workshop series at McGill University's Bellairs Research Institute, many ideas surrounding MPM were developed. Since then, MPM became a well-recognised research field with a large body of research produced and published, in particular in the MPM Workshops co-located with MoDELS.

The recent COST Action IC1404 MPM4CPS (http://mpm4cps.eu) aimed at exploring how MPM can be employed to alleviate the engineering complexity surrounding the conception of CPS. Among other efforts, the scientific community gathered around this action surveyed existing languages, techniques, and tools commonly used for modelling CPS and organized them into an ontology [167].

This work identified the need come up with a theoretical foundation for MPM and identified useful language features. Among other, the most relevant can be summarised as follows: Closeness to the essential concepts that engineers use to reason about the behaviour of physical systems, in a computationally a-causal fashion (i.e., without the need to specify early on what are inputs and outputs); the ability to precisely describe computation, at different levels of detail of the time dimension; the ability to elegantly express concurrency, synchronisation and non-determinism and to reason about properties over all possible behaviours of a system; the ability to express modal, timed, reactive and autonomous behaviour and to synthesise code; suitability to model competition for shared resources, which leads to queueing, as a basis for quantitative performance analysis; suitability for easy and correct architectural composition; the ability to express workflows at a high level of abstraction and finally, the high-level feature of modularity, supporting re-use, compositional verification of properties and the integration of black-box components such as co-simulation units. The breath of the techniques is broad.

The results of the efforts held by the MPM community, mentioned above, culminated in a book that compiles in a coherent manner well-established knowledge around fundamentals and formalisms for modelling of CPSs with a particular focus on tools and techniques that are multi-paradigm.

## 1.2 Understanding Cyber-Physical Systems

We now turn to discussing which classes of systems can be considered a CPS, what are their properties, and the sources of engineering complexity worth solving using the so-called MPM approach. Before embarking in the discussion some preliminary concepts need to be made clear.

### 1.2.1 Systems and their models

The notion of *system* is a fundamental concept used in multiple disciplines. The term conceptualises a physical existence such as an ecosystem, an organism, a machine, or a purely *abstract existence*. The latter case refers to processes, rules (such as a socio-economic system), or mathematical models. Regardless, any conception of a system is understood *(i)* as a set of *components (ii)* identifiable as a *whole*, that *(iii) cooperate* (also said to *interact*) to *(iv)* perform a particular *function*. There is a distinction between the actual systems (physical or abstract) and the human understanding of them. The formalisation of this understanding, usually limited, is called a *model*. [308]

According to Systems Theory, a system can be understood in two ways. One way is the *black-box* approach that seeks to model the *external behaviour* of the system in terms of how it interacts with the environment. Here, the behaviour of the system is seen as the relationship between the evolution of the history of *manifestation*s (also said *output*s) and the history of *stimuli* (also said *input*s). Another way to understand the system is the *white-box* approach, that seeks to understand its *internal structure* in terms of *components* and *connectors* through which interaction occurs.

Both the black-box and the white-box approach formalise knowledge about the behaviour of the system with *models*[1], which are simplified representations of the system.The utility of working representations is that they are cognitively effective means to reason about the actual system (or systems) they represent. By reasoning, we mean analysing properties of the system or predicting the future behaviour of the system. A system can thus have multiple models that cater to distinct requirements in terms of reasoning. Control Theory has historically taken the black-box approach where outputs are modelled in terms of inputs using differential equations; Computer Science has traditionally taken the white-box approach by modelling the internal structure of systems using object diagrams.

The driving idea behind understanding the internal structure is that components, bound through a certain arrangement of connections, display specific external behaviour. Moreover, the overall behaviour can be derived from *(i)* the known behaviour of the components and *(ii)* the characteristics of the connectors. Connections are abstract flows of information, or energy, that bind components through pre-designated points of interaction known as *ports*. Some component ports are *directional*, we can thus talk about *input ports (inputs)*, and *output ports (outputs)*. Other ports are *adirectional* in that they model an exchange and not necessarily a flow (consider a thermal coupler, for example).

A component can itself be seen as a system itself with its own components and connectors. The ability to continue composing systems from previously constructed ones is known as *hierarchical construction*. It is common also to model components with a *internal state* that reflects (partially and sometimes inaccurately) what the system knows about the surrounding environment and a *state-transition* mechanism typically referred to as *transition function* that is capable of producing new states from previous states upon receiving certain inputs and creating certain outputs.

### 1.2.2 Types of systems

Inputs, outputs and the state of the components are modelled in terms of *variables* that can take values from their corresponding support sets. These variables are said to be *continuous* or *discrete* if their support sets are, in a mathematical sense, dense or discrete, respectively. The behaviour of the system, as modelled in terms of internal state and outputs, evolves from a previous state according to some notion of time. Time can also be understood as continuous or discrete. In *continuous time*, it is possible to derive the new state and the new outputs for the system for an arbitrarily small time delta; whereas in *discrete time*, the new state and outputs can be derived only at predefined intervals, or upon the occurrence of a certain event.

Systems can be classified into distinct types depending on the nature of inputs, outputs, state variables, state transition function, and the notion of time. A well-accepted classification is as follows [308]:

- **Static vs Dynamic Systems.** A system is said to *static* if its output depends only on the present input. If the output of the system depends on the history of past inputs, then the system is said to be *dynamic*.

[1] The precise meaning of the term 'model' is discussed later in Section 1.3.1

- **Causal vs Acausal (Non-causal).** Whenever the output value of the system is independent of future values of input, the system is said to be a *causal system*; whenever the output values of the system depend on input values at any instant of time, the system is said to be a *acausal system*.
- **Linear vs Non-linear.** A system is said to be *linear* (respectively, *non-linear*) if changes on the output are proportional (respectively, not proportional) to the changes of the input.
- **Discrete State vs. Continuous State.** Those systems in which the state variable(s) change only at a discrete set of points in time are said to be *discrete state* systems; systems in which the state variable(s) change continuously over time (e.g., a water tank filling in), are said to be *continuous state* systems.
- **Discrete Time vs Continuous Time.** A system is said to be *discrete*, in contrast with *continuous*, if it has a countable number of states.
- **Time-Driven vs. Event-Driven.** A *time-driven* system changes state in response to a uniform physical time. While, a *event-driven* system changes state in reaction to the occurrence of asynchronous discrete events (not changing state between event occurrences).
- **Time-Variant vs. Time-Invariant.** A system in which certain state variables change with time causing the system to respond differently to the same input at different times is called *time-variant*; a system that yields the same output for a given input at distinct points in time is said to be *time-invariant*.

A dynamic system that exhibits both continuous and discrete time behaviour is said to be an *hybrid system*. The study of hybrid systems is very important as they arise often in the composition of discrete with continuous components typical of cyber-physical systems.

### 1.2.3 What are Cyber-Physical Systems?

It is assumed that the cyber component *controls* the physical component in the sense that the cyber component has some 'intelligence' or, at least, some strategy to drive the physical to reach a predefined observable goal. The converse is not true, i.e., the physical does not aim at driving the cyber to reach a certain predefined goal. This formulation puts CPS in the realm of Control Systems theory. Despite de comprehensive nature of Control Systems theory, CPS are not a particular case of Control Systems.

In a *control system* one component, known as the *controller* realises a control model that acts upon the physical environment component known as *plant* by means of a *control action*. The controller knows the desired value of a variable and the current value of that variable in the physical component. The controller then creates a sequence of control actions to correct (i.e., to minimise the distance over time) of a variable (measured from the physical system) to the desired value (the goal). This arrangement is known as *control system*. It is, therefore, reasonable to ask: *"what distinguishes a control system from a CPS?"* Besides the very idea of a cyber control component—a discrete computer algorithm controlling continuous physical phenomena—it seems that there is no single characteristic that of itself defines a CPS. However, it is well-accepted that cyber-physical systems consist of a large number of interacting components and display a number of recurring characteristics that distinguish them from classic control systems. In particular:

- **Extensive 'cyber' components** that encode complex control and supervisory control logic. Typically, they have multiple cyber sub-components that support complex action coordination and require the processing of very large amounts of historical data.
- **Very large scale of operation** outreaching several millions of elements (sometimes heterogeneous) involving an inherent complexity of hierarchies and interactions. Examples of those systems are Smart grids, Smart cities, Particle Physics Detectors, among others.
- **Hybrid discrete-continuous nature -** where a very large number of discrete components (especially 'cyber' components) are connected to physical components (continuous in nature) thus creating a hybrid system. Also, many of these components are quite heterogeneous with respect to their types (Section 1.2.2).
- **Integration with multiple external systems** by processing data and events in distinct formats from multiple systems with varying bandwidth and message delivery guarantees.
- **Highly networked and hierarchical** connecting many components typically through digital networks with distinct communication buses and protocols.
- **Adaptable** where the system must adjust their behaviour to patterns that could not be accounted for at design time.

- **Human in the loop** offering specific provision for Human actors to consume outputs and give inputs. Human actors are often modelled as components with specific behaviour requirements. and assumptions.

The pervasiveness of IoT with a large number of connected devices has created an upsurge of interest in CPS. However, the term is somewhat overused, and it has now become an umbrella for any system that interacts with the physical environment. For instance, at the current state of practice, a developer of a temperature logger might as well claim that his device is a CPS. This raises the question of *what is the minimum requirements for a system to be considered a CPS?* One key observation is that having identifiable cyber and physical components are not enough for a system to be considered cyber-physical; the cyber and physical components must influence each other[2]. Another reasonable question is whether this relationship must be of mutual influence, or whether, instead, the cyber component may not be influenced by the physical component. It is clear that since the cyber controls the physical, *the cyber must have the means to influence (act upon) the physical.* It follows, that a system where the cyber component does not influence the physical component is not a CPS.

CPS also builds on *embedded systems*, which are self-contained systems that incorporate elements of control logic and real-world interaction. An embedded system is typically a single device, while CPS include many constituent systems. Further, embedded systems are specifically designed to achieve a limited number of tasks, often with limited resources. A CPS, in contrast, operates at a much larger scale, potentially including many embedded systems or other CPS elements including human and socio-technical systems.

It becomes clear from the above that having a cyber and a physical component is not a suficient condition for system to be considered cyber-physical systems. The example of the temperature logger system, which only reads from the physical system, despite having a cyber and a physical component, is not a CPS.

### 1.2.4 Sources of Engineering Complexity

Cyber-physical systems are complex to build due to the inherent complexity of the problems they solve that consist of coordinating action to optimise multiple (possibly contradicting) goals in systems with vast numbers of sensors and actuators. The other source of complexity is, more of accidental nature, and has to do with the heterogeneity of the components.

One starting source of complexity is that cyber components are discrete in nature but must act in a time-driven fashion and, for that matter, they must reason about time. Concepts of *duration*, *deadlines*, and *simultaneity*, must be dealt with and modelled explicitly. Yet, programming languages abstract away the notion of time and provide little or no support for time[3]. Timing behaviour is achieved through dedicated timing hardware, interrupt control routines, and timer libraries—in the words of Eduard Lee, *"programmers have to step out outside of the programming abstraction to specify timing behaviour."* Instead of being explicitly modelled, timing behaviour is obfuscated and buried under the complexity of the orchestration of these mechanisms making the *correctness* of the composition between the cyber and physical components very hard to achieve. The problem is exacerbated as more components are added onto the system.

Not only discrete components often display incorrect timing behaviour but interfacing discrete and continuous components poses another unexpected engineering challenge. The composition of a deterministic model of the cyber with a deterministic model of the physical results in a non-deterministic model that is very difficult to analyse. To understand why to consider that it is impossible, in practice, to guarantee that the components that implement the two models will be perfectly aligned, concerning time. This applies especially to the mechanism that they use to communicate and synchronise because this mechanism operates according to certain assumptions (constraints) of time.

The actual implementation of CPS is largely component-based. These components are multi-vendor and multi-technology and, as a result, they often have different communication protocols, response timings, distinct tolerances and operating conditions. One side of the problem is that some of these constraints are not known or accounted for upfront. Another side of the problem is that while constructing the system, these constraints are not modelled and handled explicitly. A lot of accidental complexity arises when trying to assemble components with such variability, especially because typical CPS consists of a large number of components.

[2] In terms of modelling, this means that they must be bound through at least one connector

[3] A rigorous semantics of time is absent from standard computer languages

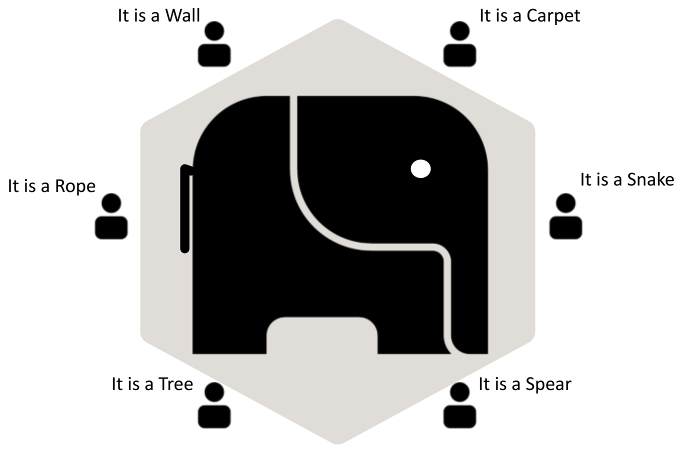

Fig. 1.1: Illustration of the tale of the 6 blind men. Each blind man, from his own perspective, understands the elephant as a diferent object.

CPS also have stringent requirements that cannot be relaxed. Besides the usual guarantees of functional correctness that apply to any engineered object, CPS are especially known for their large number of extra-functional (also known and 'non-functional') requirements that cover issues of reliability, performance, safety, security, among other. What is relevant to note is that extra-functional requirements are known to pose complex constraints to timing component design and to timing behaviour. Moreover, since these systems are critical, they often have to undergo strict qualification/testing processes. When changes to the system are needed, they are often discouraged due to the costs involved, making CPS difficult to adapt to changing requirements. There should be a reliable means to guarantee that certain aspects of a system remain untouched and therefore, do not need to be tested again.

Actual realisations of components sometimes interact in ways that were not designed upfront. Often physical components interfere due to electromagnetic or thermal interference. Emerging behaviour as of complex behaviour that arises from the interaction between components, which was not planned upfront. While developing CPS, it is important to have the means to explore alternative designs.

## 1.3 Modeling of a Cyber-Physical System

The engineering process of CPS requires distinct disciplines to be employed. Each discipline typically creates a design (also said to be a *view* or *understanding*) of the system for its own purposes in the form of a model. Models are created using abstractions, heuristics of decomposition, and tools of analysis typical of each discipline. Each discipline also brings along a body of knowledge that enables humans in charge of modelling the reality to critically assess the correctness and soundness of the model being produced. In this sense, the piecemeal approach of having distinct models organised by discipline is effective. Another motivation for having distinct models is that, even within the same discipline, they are required to answer distinct questions. Models also have to be produced with distinct levels of detail, and therefore, the modeller has to find the right balance and capture the right things creating a model adequate to the question being studied.

Modelling of a CPS system is inherently represented in multiple views of the system (most of the times following the principle of separation of concerns). As in the ancient tale of the six blind men (see Fig. 1.1), no single view (nor corresponding modelling formalism) can model all aspects of a system. Similarly, in CPS engineering, the results are models reflecting distinct views of the problem, expressed in multiple notations. One problem for the CPS engineer, as it is also for blind men, is *"how to integrate knowledge to form a more approximate model of the reality?"* Naturally, modelling of CPS systems calls for a trans-disciplinary approach that merges the different models into a unified abstraction of reality. An essential challenge of CPS is thus how

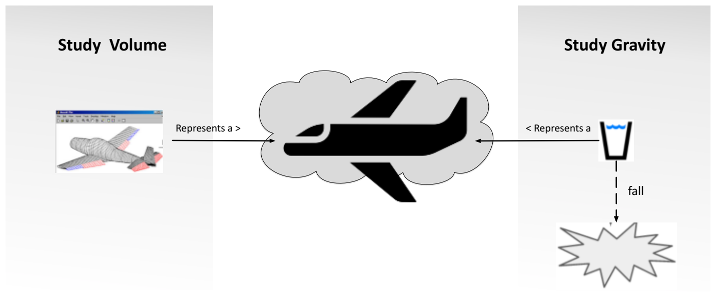

Fig. 1.2: Models of an airplane: The CAD model on the left to study volume; on the right side the model for studying gravity.

to conjoin abstractions of the various engineering disciplines and the models for physical processes including differential equations, stochastic processes, among others.

Currently, there is no standard design and modelling approach to integrate models produced by distinct disciplines of CPS. Indeed, the knowledge captured in models only comes together when assembling physical (prototype) implementations to evaluate some relevant properties of a given CPS. A more sophisticated approach is to avoid creating prototypical implementation altogether and (co-)simulate the models. This is a more generic approach as it enables verifying properties independently from the particularities of the implemented prototype.

Previously, we have mentioned the term *model* assuming some implicit notion of modelling based on abstraction, which is the process of removing details in the study of objects or systems in order to focus attention on details of higher importance. However, not all conceivable abstractions or representations of a system can be considered a model. Indeed, a more precise understanding of these concepts is required. Let us look at these in further detail.

### 1.3.1 What is a Model?

A model of a system is an abstraction (a representation) to make predictions or inferences [176] about a reality. More specifically, this reality is a system under study (SUS), whose governing rules and properties we want to understand, within the context of a given *experimental frame*. An experimental frame denotes the limited set of circumstances under which a system is to be observed or subject to experimentation.

According Stachowiak [256], three main properties should hold in a model. The first property is the *mapping feature*, which means that any model, to be called as such, should be faithful to or based on an original, that exists or simply be the formalisation of an idea to be realised at some point in the future. The second property is *reduction feature*. In this case, there is no model if it does not remove unnecessary detail, and select original properties useful to the purpose of the model in hands. The third is the *pragmatic feature*. In this last property, a model needs to be usable in place of an original with respect to some purpose. Other important, yet not fundamental characteristics of a model are *purposeful* [243], *understandable* and *cost-effective*.

To make the idea of models more concrete, let us illustrate these properties with a simple example. Consider the case of an aircraft where we are concentrating on the study of the single properties of mass and gravity. For this purpose, we can claim that a glass of water is a rough approximation of an aircraft (and it is chosen as a physical representation of the SUS). When it falls, it takes some time to reach the floor, and then it breaks. Can the glass be considered a model for the aircraft? For the purpose of evaluating the effects of gravity (our scope) we have *(i)* a mapping feature, as the object (glass) represents the original (the airplane); *(ii)* a reduction feature, as all the unnecessary details like shape, aerodynamics, architecture, among other, are removed, and *(iii)* it is pragmatic in terms that we can substitute the glass for the real airplane. Finally, it is *(iv)* purposeful, as it is meant to study gravity, *(v)* understandable, as it is a straightforward representation that everyone understands; and *(iv)* cost-effective, to study and substitute the real one during the fall it is several orders of magnitude cheaper and ethically acceptable. Another possible example of a model, for the same case study, can be the CAD drawing of the aeroplane for the goal of studying the volume of the machine, for purposes like studying how it fits in a hangar or what is the internal volume for the purpose of choosing the proper ventilation system.

### 1.3.2 Multiple Formalisms in CPS

As it became clear above, the engineering process of CPS results in a collection of models. Models are abstractions of a system (a reality) that has properties worth studying. It is well known as well that models, especially of distinct disciplines, are are expressed using correspondingly different modelling formalisms. A *modelling formalism* is a language that has formal syntax and semantics. The usual meaning of 'formal' is precisely and unambiguously defined, mathematically, in the form of Differential Equations, Finite State Automata, State Charts, Petri Nets, among others. Distinct formalisms exist because they are more concise and enable answering efficiently to distinct classes of questions. Indeed, no single formalism can be used to model all aspects of a system, as the formalism to be used depends on the nature of the problem to be solved.

Formalisms also enable the manipulation and re-writing of models, or parts of models, into other that are equivalent to the originals (for a given semantics), and whose realisations have more desirable properties. Such as, being less redundant, more compact, faster, or consuming less energy, for example.

In order to overcome the complexity of the problem, a common modelling practice, is to describe models of the same reality at different levels of abstraction (sometimes using correspondingly distinct formalisms.) Models expressed at distinct levels of abstraction are linked to one another through structure-preserving maps. Indeed, an overarching issue with distinct formalisms is merging models of the same system through these maps. There needs to be a notion of *consistency* among them.

These multiple formalisms are used to model a system interacting with its environment, its architecture and components, at different levels of detail, approximation and abstraction, and from different viewpoints, as well as the platforms the software components of the system will be deployed on. The integration of models produced according to distinct formalisms is achieved by mapping (compiling) the model into lower level *super formalisms* that integrate different domains such as Bond Graphs, or other formalisms to integrate discrete and continuous modelling constructs such as DEVS.

To support the design of CPS, not one single super-formalism, but rather of a multitude of modelling formalisms, chosen for their particular reasoning and analysis features need to be employed. These features make each of them most appropriate for a particular CPS design (sub-)goal. Pragmaticaly, the engineer (in the role of modeller) needs also to know the strategies (formalised as *processes*) to describe reality according to the formalism. The formalism together with these said processes and constraints form what is otherwise known and as a *modeling paradigm*—the object of study of Multi-Paradigm Modelling [286].

## 1.4 Multi-Paradigm Modelling of CPS

Multi-Paradigm Modelling (MPM) has been recognised as a powerful approach (a paradigm in its own right) that may be helpful in designing, as well as communicating and reasoning about CPS, which are notoriously complex because of their cross-discipline borders and inter-domain interactions.

To develop a CPS, project managers and engineers need to select the most appropriate development languages, software lifecycles and "interfaces" to specify the different views, components and their interactions of the system with as little "accidental complexity" [52] as possible. For example, when it is known that system/software requirements are likely to change frequently during the project's course, selecting an Agile development process may help to cope with evolution and change. If the system's behaviour requires that operations are triggered when data becomes available, similar to reactive systems, Data Flow languages may help to specify the most critical parts of the software behaviour in a precise way, making it amenable for timing analysis.

### 1.4.1 What is a Paradigm?

In Computer Science, general-purpose programming languages (GPLs) can be classified according to the paradigm(s) they support. For example, Eiffel is object-oriented and supports the contract-based-design paradigm, Prolog is declarative, and Lisp is functional. The paradigm characterises the underlying syntactic and semantic structures and principles that govern these GPLs. In particular, object orientation is imperative in nature and imposes viewing the world in terms of classes and communicating objects, whereas the declarative style relies on term substitution and rewriting. As a consequence, a statement in Eiffel has very little in common with a Prolog sentence due to the very different view supported by each language. A programming paradigm directly translates into different concepts encoded in the GPL syntax definition (known as a metamodel in the Model-Driven Engineering world). Very naturally, the idea of combining several paradigms at the level of GPLs led to more expressive, powerful programming languages such as Java (which is imperative, object-oriented, concurrent, and real-time and, recently, functional) and Maude (which is declarative, object-oriented and also concurrent and real-time).

What is a *paradigm* then? The science philosopher Kuhn [175], while investigating how science evolves through paradigm shifts, defines it as an open-ended contribution that frames the thinking of an object of study with concepts, results and procedures that structure future achievements. Though seemingly far from the concerns in the discipline of Computer Science, this definition does highlight the emergence of a *structure* (a formalism) that captures the object of discourse, and the notion of *procedures* (the processes) that guide achievements.

### 1.4.2 The dimensions of Multi-Paradigm Modelling

The application of MPM requires *modeling everything explicitly*, using the *most appropriate formalism(s)*, at the *most appropriate level(s) of abstraction* [266]. This suggests that a *paradigm* can be understood as an arrangement of the properties in each of the dimensions described above: the *formalisms* and the *levels of abstraction* in the modelling activities.

Oftentimes, formalisms are general-purpose, and hard to be used by modellers (domain users, or domain experts) who need to start by picking the most adequate formalism based on its well-known semantics, e.g., Petri Nets for workflows and concurrency, or Statecharts for describing event-based systems, among others. However, having to master mathematical notation poses a steep learning curve. To alleviate this problem, specialised languages, called *Domain-Specific Modelling Languages*, are created to simplify the act of expressing the modeller's specification intent. The constructs in these languages are designed to be closer to the way domain experts are used to conceptualize problems. The systematic approach of building new modelling languages, is called *Modelling Language Engineering* (MLE) and must, itself, follow an engineering process [**?**].

To tackle complexity during the course of system development, three basic abstraction approaches are commonly combined: *Abstraction/Refinement*, *Architectural decomposition*, and *View decomposition*.

- **Model abstraction (and its dual, refinement)** is used when focusing on a particular set of *properties* of interest. While abstraction implies removing unnecessary detail, in opposition to refinement, the same set of chosen properties should hold both on the abstract and detailed models. Verifying the property on the abstract model is, expectably, cheaper (or simpler) than in the detailed model. Yet, note that the more detailed model does have some advantages as it will allow the correct assessment of a larger set of properties which can not be covered otherwise.
- **Architectural decomposition (and its dual, component composition)** is used when the problem can be broken into parts, each with an appropriate *interface*. Such an encapsulation reduces a problem to *(i)* a number of sub-problems, each requiring the satisfaction of its own properties, and each leading to the design of a component and *(ii)* the design of an appropriate architecture connecting the components in a way that the composition satisfies the original required properties. This a breakdown often comes naturally at some levels of abstraction, using appropriate formalisms (which support hierarchy), for example, thanks to locality or continuity in the problem/solution domain. Note that the above describes a top-down workflow where decomposition of the requirements leads to the design of components followed by the architectural composition of these components. A bottom-up workflow is also possible, where existing components are combined to satisfy full-system requirements.

- **View decomposition (and its dual, view merge)** is used to enable the collaboration between multiple stakeholders, each with different concerns. Each viewpoint allows the evaluation of a stakeholder-specific set of properties. When concrete views are merged, the conjunction of all the views' properties must hold. In the software realm, IEEE Standard 1471 defines the relationships between viewpoints and their realisations, views. Note that the views may be described in different formalisms.

One particular combination of the former approaches leads to Contract-Based System Design [84]. Indeed, modelling activities are combined into *processes* (or workflows) that relate the various MPM activities. Processes may be *descriptive*, charting the sequence of activities carried out as well as the artefacts involved, *proscriptive* by declaratively specifying constraints on the allowed activities and their combinations, and *prescriptive* allowing enactment. Processes are often supported by toolchains whereby different tools support different activities. It can be said that a MPM framework aims to support (meta-)tool builders who assist practitioners to reason about CPS and figure out which formalisms, abstractions, workflows and supporting methods, techniques and tools are *most appropriate* to carry out their task(s).

Ultimately, by selecting, organising and managing the three dimensions above (formalisms, abstractions, and processes), MPM facilitates the communication between experts to help them better grasp the essence of how their CPS are built. Moreover, it also facilitates a rigorous comparison of distinct approaches to MPM based on their core MPM components. The implications and challenges that MPM brings to formalisms and to abstraction mechanisms need to be discussed further.

## 1.5 A foundation for MPM4CPS

This book introduces a representative set of modelling formalisms, each with a characteristic collection of features. The set is by no means complete but rather intended to showcase the wide variety of features available in well-established formalisms, often supported by scale-able tools. These may be used to choose a most appropriate formalism for a particular task at hand, as a starting point for looking into more formalisms, with other desirable features (such as the inclusion of spatial distribution as found in Cellular Automata or Partial Differential Equations), or as a basis for the design of Domain-Specific Modelling Languages (DSMLs) to maximally constrain a modeller to a specific application domain.

The formalisms introduced in this book may also be combined, leading to "hybrid" languages, when a particular combination of features is required that is not available in a single formalism. Note, however, that some of the formalisms introduced in this book are already hybrid in the above sense. Bond Graphs, for example, unify modelling of systems in various physical domains by focusing on power flow, and Modelica combines features of Object-Orientation with those of computationally a-causal (equation-based) modelling. The Architecture Analysis and Design Language (AADL), which focuses on embedded systems, with architecture at its core, brings together different viewpoints, making it suitable for documentation, analysis and code synthesis.

The material is presented in a bottom-up fashion. Starts by presenting the formalisms to model physical components. Then mechanisms encapsulate and re-use description of CPS components are presented as a means to tame the complexity of large descriptions. We then present formalisms analyse the behaviour of CPS. Finally, these formalisms are put together through the use of architectural descriptions and processes.

### 1.5.1 Modelling physical components

When modelling a physical system, the first decision to make is whether the properties of interest of the system depend on the spatial dimension. The heating of a metal object due to an electrical current flowing through it, for example, is determined by the interaction between the electrical and thermal physical domains. It depends on the geometry of that object as well as on the object's material properties such as density, electrical conductivity, relative permittivity, heat capacity, and thermal conductivity, and their distribution across the entire object. The object's dynamics can then be described using the mathematical expression of the relationships between the physical quantities of interest. Due to the dependence on spatial coordinates, this requires the use of "distributed parameter" models. These are typically expressed using the Partial Differential Equation (PDE) formalism.

When the parameters of an object are sufficiently homogeneous over its geometry, the properties of interest may not depend on the spatial dimension. In that case, the parameters may be aggregated over an entire object, and it may be reduced to its dimensionless essence. A rigid body in the mechanical domain with a constant density over its geometry may, for example, be reduced to a simple “point mass”. Its dynamics can be described using Newton’s Laws or a Hamiltonian or Lagrangian formulation. Such “lumped parameter” models are typically expressed using the Ordinary Differential Equation (ODE) or Differential Algebraic Equation (DAE) formalisms.

Often, the physical components of a Cyber-Physical System span distinct physical domains (electrical, mechanical, thermodynamic, hydraulic, etc.). The Bond Graphs formalism described in Chapter 2 unifies the different domains at a “lumped parameter” level of detail. It recognises the analogy between physical processes in different physical domains, such as energy storage and dissipation. A system is modelled as a Bond Graph connecting nodes representing physical elements. These nodes encode how physical quantities such as voltage and current are related in, for example, a resistor. The Bond Graph’s edges—called Power Bonds—denote the power flow between the nodes. Special nodes—junctions— encode conservation laws, generalisations of Kirchoff’s current and voltage laws in the electrical domain. The chapter introduces a systematic procedure for modelling multi-domain physical systems. It starts from Idealised Physical Models and converts these into Bond Graph models. These Bond Graph models are computationally a-causal and can be translated to a set of Differential-Algebraic Equations (DAEs). Computational a-causal models consist of equations relating signals (variables, functions of continuous-time), without specifying which variables are known (inputs) and which are unknown (outputs), nor how these equations need to be solved (i.e., how the unknowns are computed from the knowns). Such DAEs can be represented in (mathematical) Equation-Based modelling languages such as Modelica. Modelica is described in Chapter 3 The Bond Graph chapter then shows how computational causality can be assigned, effectively converting to a Continuous-Time Causal Block Diagram (CT-CBD). Causal Block Diagrams are described in Chapter 4. Causality assignment on a Bond Graph model may give insight based on physics, into flaws in the model. This aid in “model debugging” is thanks to the (physical) domain-specificity of the Bond Graph formalism.

### 1.5.2 Joining the ‘Physical’ with the ‘Cyber’

Cyber-Physical Systems are composed of networked physical and computational components. To allow for a modular and hierarchical design of such systems, maximising model re-use and enabling the construction of model libraries, the Modelica language, described in Chapter 3 combines features of Object-Orientation such as encapsulation and inheritance with those of computationally a-causal (equation-based) modelling. Computationally a-causal models allow the modeller to express the fundamental laws of physics using mathematical equations. The semantics of Modelica is given by expanding object-oriented constructs such as inheritance, by instantiating classes, and by flattening the hierarchy. This results in a set of (hybrid) Differential-Algebraic Equations. For each particular simulation experiment context, computational causality can be assigned by a Modelica compiler. This effectively generates a model in the Continuous-Time Causal Block Diagram formalism. Most Modelica compilers will further (time-)discretise these models, either symbolically through "inline integration" or by calling upon external numerical solvers. This ultimately leads to an executable simulation code. Note that it is possible to create a Modelica library with Bond Graph components. In this case, the Bond Graph causality assignment procedure will not be used. Rather, Modelica’s causality assignment will be applied to the entire model, including non-Bond Graph parts. Through the code-based specification of functions, Modelica also allows traditional object-oriented code to be represented. It is this combination of code, equations, and hybrid constructs such as "when" (which allows the introduction of discrete events, so-called "state events", based on conditions over continuous behaviour such a crossing a threshold value) that makes Modelica suited to build models of Cyber-Physical Systems, spanning their physical, network and computational parts.

As mentioned earlier, declarative, computationally a-causal models need to ultimately be transformed into a causal form, which allows for their computational solution. In Chapter 4, a family of Causal Block Diagram (CBD) formalisms is introduced. Causal Block Diagrams consist of a network of computational blocks. Each block specifies the computationally causal relationship between its input and output signals. The block diagram network specifies how outputs of one block are connected to inputs of other blocks. Such a connection denotes that the values at connected output and input ports must at all times be equal. The three CBD variants are built up gradually. The Algebraic Causal Block Diagram (ALG-CBD) formalism has no notion of time: values are propagated through a CBD according to a computation “schedule” (i.e., the order in which block computations are invoked) derived from the dependency structure encoded in the block diagram network. Special care needs to be taken to detect and properly solve dependency cycles known as “algebraic loops”. A discrete (Natural Number) notion of time is then added, as well as a delay/memory block, resulting in Discrete-Time Causal Block Diagrams (DT-CBDs). These are equivalent to Synchronous Data Flow (SDF) models. Finally, the Real Numbers are introduced as a time base to give Continuous-Time Causal Block Diagrams (CT-CBDs). These have the same expressiveness as mathematical equations and need to be discretised to allow for their computational solution. Numerical discretisation techniques are used to turn a CT-CBD into a DT CBD.

Very often, it is reasonable to abstract away many of the details of the behaviour of a system and to only focus on pertinent “events”. Such Discrete-Event abstractions see a system as changing its internal state, either reacting to input events of autonomously changing its state after a certain time (due to an internal “time event”) and possibly producing output events at certain times. As only the events are what changes the state of the system, and in between event instances, nothing pertinent is assumed to happen, the evolution of the state over time is piecewise constant. Unlike in Discrete-Time (DT) formalisms, in Discrete-Event (DE) formalisms, time advances in leaps and bounds, from pertinent event to pertinent event. One advantage of DE abstraction is performance: a simulator will directly step to the next time at which an event occurs whereas a DT simulator would have to step through time in fixed increments even if nothing noteworthy happens (i.e., the state remains unchanged). The DE abstraction is commonly used to study the competition of different processes for shared resources. If resources are constrained, this inevitably leads to queueing. The abstraction is hence useful for simulation-based performance analysis. Utilization of resources, time spent to complete an activity, the distribution of queue length and queueing time are all examples of the typical performance measures that are obtained from discrete-event simulations. Note that DE formalisms are often deterministic. Through the inclusion of distributions for parameter values such as Inter Arrival Time rather than unique values, and using Monte-Carlo simulation, distributions of performance measures are obtained. Thus, repeatable (as pseudo-random number generators – which are deterministic– are used to sample from distributions) stochastic simulations are obtained. Many DE formalisms were developed over the years. The Discrete EVent Specification (DEVS) formalism described in Chapter 5 is a DE formalism that is primitive and expressive enough to act as a DE "assembly language": models in all DE simulation formalisms can be mapped onto a DEVS equivalent. As such, it can be used to architecturally connect models in different formalisms by first mapping all components onto DEVS. The resulting architecture only contains DEVS components. As DEVS is modular and supports hierarchical architectural composition, the resulting model has a precise meaning. DEVS’s support for hierarchy makes it suitable to build model libraries and to subsequently build up highly complex models.

A different way of modularly combining state automata is found in the Statecharts formalism described in Chapter 6. The Statecharts formalism consists of hierarchies of state automata, of parallel composition of these automata, of a notion of time, and an event broadcast mechanism. The popularity of Statecharts is partly due to its intuitive visual notation. The main purpose of Statecharts is to not only simulate models, but also to synthesise from them, autonomous, timed and reactive software and/or hardware.

When a parallel composition is made of state automata, many interleavings are possible. One option is to choose a unique interleaving, leading to a unique, deterministic behaviour trace. This is what is done in the DEVS and Statecharts formalisms. To model true concurrency, this artificial sequentialisation is not always appropriate. Rather, a non-deterministic choice should be allowed. This leads, not to a single behaviour trace, but to a collection of possible behaviour traces. This collection of traces may be summarised in a compact representation in the form of a state reachability graph. The satisfaction of interesting properties may then be checked over the collection of traces. An example is the reachability of a certain undesirable state. Such properties are also expressed in an appropriate property language. Non-deterministic languages, as described above, usually have a weak notion of time: not the Natural or Real numbers are used as a time basis, but only a (partial) ordering of event instants. The focus is on concurrency and synchronisation. Rather than simulation or synthesis, such formalisms are mostly used for the analysis of properties, across all possible behaviours of a system. This makes them suited for, for instance, safety analysis. One such formalism is Petri Nets, as described in Chapter 7. Petri Nets encode state as a "marking", an $n$-dimensional vector of Natural numbers. Each element of the vector corresponds to the number of "tokens" in a Petri Net Place. The evolution of the state is encoded in a Petri Net graph which, apart from Places, contains transitions and Arcs. Thanks to the use of Natural numbers, the number of possible states can be infinite (but countable). Petri Nets are a simple formalism that, like DEVS, is often used as a common semantic domain onto which to map diverse other, often domain-specific, formalisms. The chapter also demonstrates how Petri Nets can be combined with other formalisms. In particular, the co-simulation of Petri Nets with Functional Mockup Units (encoding discretised continuous models) is introduced. This effectively leads to non-deterministic hybrid models.

### 1.5.3 Tooling support for MPM4CPS

Complex engineered systems consist of heterogenous components arranged in an architecture. Furthermore, multiple viewpoints on the same system may be of interest and ultimately (part of) a system model needs to be "deployed" on an often embedded software/hardware architecture. The Architecture Analysis and Design Language (AADL) described in Chapter 8.1 is foremost an Architecture Description Language. It allows one to provide a description of the overall system and the environment into which it will operate. From such a description, other models in other formalisms such as those described in this book can be generated. These can be further augmented to study various aspects of the system, which is essential for its optimisation, verification and validation. After a brief introduction to ADLs and their role in MPM4CPS, the AADL is presented and its use illustrated through the modelling, analysis and code generation for a simple Lego Mindstorm robot for carrying objects in a warehouse. A simple top-down architecture-centric design process is followed, starting from the capture of stakeholder goals and system requirements, followed by system design, design analysis and verification and finally automated code synthesis.

It becomes apparent from the above that complex systems modelling involves not only different abstractions, architectures and views, modelled using varying formalisms but also complex development workflows. Chapter 9 looks into the topic of process (workflow) modelling. Often complex, concurrent development processes, modelled in the form of a Process Model (PM) are built up of primitive activities which take as input, modelling artefacts and produce modified or new modelling artefacts as output. The activities may require human or computer resources, possibly leading to delays as described in the chapter on DEVS. As the artefacts manipulated by activities are models in various formalisms, it makes sense to “type” them with the appropriate formalisms. To chart the many formalisms used, and to show how they are related, a Formalism Transformation Graph (FTG) is introduced. The FTG+PM, combining FTG with PM, allows one to characterise the essence of Multi-Paradigm Modelling solution patterns to CPS development problems. One advantage of the explicit representation of the FTG+PM is that is can be used as a basis for the synthesis of MPM tools.

## 1.6 Summary

The field of CPS is affected by the complexity of different approaches, processes, and modelling languages. Indeed, this field is well-known to have an inherently multi-displinary nature and, since there is no single well accepted modelling approach, multi-paradigm modelling has been advanced as a solution. Yet, to date, literature still lacks a solid reference that introduces distinct CPS modelling and analysis techniques towards a multi-paradigm approach.

This first chapter motivates Multi-Paradigm Modelling for CPS and the need for its clear foundations. Staring from an introduction to the concept of System, it introduces distinct classes of systems, discusses their characteristics, and then derives a definition of CPS. As CPS are complex to build, we dedicate part of the chapter to discussing and drilling down on their common sources of complexity. Overall, designing a system means that one has to make use of several kinds of abstractions to describe different properties and system’s concerns, using languages (with existing, modelling formalisms and paradigms) and processes to be able to avoid unnecessary complexity. The chapter then defines what a model is, and what does it mean to use multiple formalisms when engaging in multi-paradigm modelling. Finally, a structure of the book is described, further explaining how the formalisms and techniques presented fit together.