Cyber-Physical Systems Engineering

Bernd-Holger Schlingloff

Fraunhofer FOKUS & Humboldt-Universität zu Berlin, Germany
`hs@informatik.hu-berlin.de`

Abstract. Building complex embedded- and cyber-physical systems requires a holistic view on both product and process. The constructed system must interact with its physical environment and its human users in a smooth way. The development processes must provide a seamless transition between stages and views. Different modeling techniques and methods have been proposed to achieve this goal. In this chapter we present the fundamentals of cyber-physical systems engineering: identification and quantification of system goals; requirements elicitation and management; modeling and simulation in different views; and validation to ensure that the system meets its original design goals. A special focus is on the model-based design process. All techniques are demonstrated with appropriate examples and engineering tools.

Keywords: Cyber-physical systems, embedded systems, model-based design, systems analysis, requirements analysis, systems modeling, block diagrams, state-transition systems, code generation

1 Introduction

The systematic, model-based design of cyber-physical systems is a fascinating subject both for academic researchers and for practitioners from industry. Embedded systems, which control, activate and supervise technical systems, have become an integral part of our daily lives. Already at present (year 2015) there are more embedded systems than people on earth. Moreover, in the last decade the number of such systems has been exponentially increasing; it is estimated that by 2020 on average each human will possess around one hundred different embedded systems. Mostly we do not even notice that we are using such systems: a modern car contains between 50 and 100 electronic control units, from driver assistance systems to motor- and battery controllers. Also their functionality and usability is steadily being increased. More and more functions are realized by software, and the software running on each device becomes more and more complex. In some cases even human life depends on the correct functioning of the software. For example, think of an artificial pacemaker for the heart, or of a signalling device for a high-speed train.

Another order of magnitude in complexity is added by the fact that more and more embedded systems are being equipped with communication links, so that they can collaborate to deliver some combined service. Such *cyber-physical*

systems will be the next big revolution in information technology. Current keywords describing this fact are "the internet of things", "ambient intelligence", "smart environment", and others. However, this technological advance will only be possible if engineers can manage to master the ever increasing design complexity. The combined software in a "smart car" presently consists of more than 100,000,000 lines of code, written jointly by more than 1.000 software developers — imagine! For the development of these devices, conventional and ad-hoc engineering techniques are approaching their limits.

Thus, advanced design methods for these systems are absolutely necessary. In this chapter, we will describe the state of the art and some research directions for systematic engineering of embedded and cyber-physical systems. The material is based on a lecture series with the same title, where the overall curriculum is described in [Sch14]. We start in Sect. 2 by defining cyber-physical system and listing some of their characteristic attributes. In Sect. 3 we give a short introduction into systems analysis, and show how to define requirements with the example of a pacemaker. Section 4 comprises the main part of this chapter. It deals with modeling in various views: systems modeling in SysML, continuous modeling with block diagrams in Simulink/Scicos, and discrete-state modeling with UML state machines. In Sect. 5 we show how to transform these models into executable code. Finally, Sect. 6 concludes the chapter.

2   Embedded- and Cyber-Physical Systems

We begin with some definitions of relevant terms. The word *system* may be the most over-used word in computer science. From its Greek origins ($\sigma\upsilon\sigma\tau\eta\mu\alpha$) we can infer that a system is "something which is composed". Since probably everything in this world is composed from something else, the term does not define a class of objects; it does not separate those things which are systems from those which are not. However, it serves to describe an aspect of objects, namely being a combination of several other objects called *components*. Components are things "to be put together" to constitute a system. They can be elementary, meaning that we do not decompose them further, or subsystems, which are again composed. In a system, the components interact in some way, or else we would not consider them to be parts of the same system. The international standard ISO/IEC 15288:2008 [ISO08] defines a system to be "a combination of interacting elements organized to achieve one or more stated purposes". A similar definition is given in the "Systems Engineering Handbook" of the INCOSE (International Council of Systems Engineering) [INC00]:

> [A system is] an integrated set of elements, subsystems, or assemblies that accomplish a defined objective. These elements include products (hardware, software, firmware), processes, people, information, techniques, facilities, services, and other support elements.

Both of these definitions refer to an objective or purpose of the composition. That is, only those systems are included which are composed by humans. Natural

systems such as ecosystems, biological systems, or social systems are not included in the considerations.

Human-made systems serve some purpose, the provide a *function*. In a *technical system* this function is to process matter or energy. By *processing* we refer to the *transformation* or *transport*, that is, the change of form or location of something.

Typical examples of technical systems are

- a thermal power plant (transforming one form of energy to another),
- an injection-moulding machine (transforming the shape of matter), and
- a forklift truck (transporting matter).

The function of a *computational system*, in contrast to the function of a technical system, is to process *information*. Similar to the notions of "matter" and "energy", the term "information" describes a basic concept which we will not try to define here. Typical computational systems are

- a pocket calculator,
- a word processor (transformation of information), and
- a mobile phone (transport of information).

The last of these examples exhibits a general problem of delimitation when dealing with information processing: Since the representation of information in a material world is always bound to physical objects, all information processing contains the processing of these physical objects. In order to decide whether a system is a technical or computational system, it is important which processing aspect is predominant. In case of a pocket calculator, the category is pretty clear: The function is best described as "a device performing arithmetic operations on the input and displaying the result". It would be strange to describe it as "a device transforming battery power into light signals and excess heat". In case of a phone the situation is not so clear: With the first "tele-phones" in the 1870's the (technical) aspect of transforming sound waves into electrical signals and back predominated. No information processing took place: the voltage level on the microphone or speaker directly reflected the amplitude of the corresponding sound waves. With modern smartphones, however, the information processing clearly is predominant: the wave forms are digitally recorded, split into packets, wrapped, encoded and decoded according to the chosen transmission protocol, etc.

With the above definitions of technical and computational system we can define a central term of this chapter.

Definition 1. *An* embedded system *is a computational system, which is a fixed component of a technical system.*

In other words, an embedded system is a computer which is an integral part of some machine. Without the embedded system, the machine would not work properly. In an embedded system, the information processing is designed, built

and operated with a particular purpose in a technical process. A schematic diagram of this definition is given in Fig. 1: The embedded system is the computational system inside the technical system, which again is part of a physical environment. The embedded system communicates with the technical systems via sensors and actuators, whereas the technical system processes material and energy in the "real world".
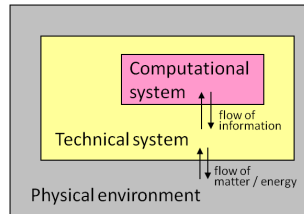
Fig. 1. Schematics of an embedded system

Typical examples of embedded systems are

- the attitude control of an Ariane space rocket,
- the TCAS (traffic collision avoidance system) of a commercial airplane,
- the ETCS OBU (on-board unit) of a high speed train,
- automotive electronic control units: ABS, ESC, cruise control, etc.,
- the temperature control of a nuclear power plant,
- the controler of an industrial punching machine,
- a bicycle computer, and
- an artificial pacemaker for the human heart.

The definition implies some characteristic properties of embedded systems. The following properties are fundamental:

- **Fixed part of a technical system:** An embedded system is usually physically attached to the technical system it belongs to. Its dimensions and capacities are fitted for the particular system, and it cannot be easily exchanged or replaced.
- **Dedication to a particular purpose:** Within the technical system, the embedded systems fulfills a predetermined function. In contrast to a universal Turing machine (or any general-purpose computer) often it can do certain special computations only, and cannot be arbitrarily programmed.
- **Interaction with a physical environment:** The technical system containing the embedded system performs a physical process, transforming matter or energy in the real world. To interact with this physical environment, the embedded system uses sensors and actuators.
- **Reactivity to external stimuli:** Pure computational systems usually terminate after finishing the calculation of their result. Since the function of an

embedded system is determined by a physical process, it must be constantly able to react to inputs from this process and cannot terminate. Often there are certain time limits for the computation of the output, thus the system must react in real time.

Besides these fundamental properties, there are some secondary attributes which embedded systems often, but not always, possess:

– **Supervising and controlling:** In most embedded systems, the function of the computational part is to supervise and control the technical system in which it resides. As an example, consider the controller of a washing machine which regulates motor, water valves and detergent flow, heating, etc. However, there are also embedded systems which are not control systems, e.g., devices for data acquisition.
– **Mass-produced:** Many embedded systems are integrated in end-consumer goods and thus have to be manufactured at extremely low cost. For example, for an automotive device which is to be incorporated in a million cars, saving 1ct in production saves 10,000 dollars in total. Nevertheless, there are also embedded devices which are unique, for example, a spacecraft on-board unit.
– **Difficult to maintain and extend:** Since the embedded system is distributed together with the technical system, software updates are often hard to realize or commercially unattractive. Sinve embedded systems are made for a particular purpose, it is mostly not possible to extend the functionality to a "version 2.0". For example, updating the software of an automotive device costs up to 100 dollars per car which can be very expensive if a large number of cars is concerned. However, a trend is to connect embedded devices to the internet in order to make upgrades possible.
– **Highly available, trustworthy and safety-critical:** Since embedded systems are becoming ubiquitous, we rely more and more on their availability and correct behaviour. For example, without electronic engine control it would be impossible to build a car or plane satisfying modern environmental standards. Embedded systems are increasingly also realizing safety-critical tasks, e.g., in an antilock braking system, where a failure might have fatal consequences.

An important trend in embedded systems is that they are being equipped with communication facilities (WLAN, Bluetooth, GSM/UMTS/LTE, Zigbee, etc.) so that they can exchange information with other computational systems. By the interconnection of a significant amount of embedded systems, new functionalities can be realized. This leads to the notion of *cyber-physical systems*.

Definition 2. *A cyber-physical system is a system of embedded systems which are interconnected and/or connected with other computational systems via communication networks.*

Of course, there is no strict separation between the notions of "embedded system" and "cyber-physical system": On the one hand, each embedded system is

a cyper-physical system with just one component; on the other hand some embedded control devices consist of several interacting processors and thus can be viewed as a cyber-physical system. Thus, in this article, there will be no strict differentiation between these two notions.

By definition, the composition of embedded systems in a cyber-physical system is such that it accomplishes a defined objective. The objective is such that it cannot be achieved by any single one of the constituent technical systems. Typical examples of cyber-physical systems are

- the set of electronic control units in a car: In present-day cars there are typically between 50 and 80 electronic control units (ECUs) which are interconnected via different on-board networks (CAN, LIN, MOST, FlexRay, etc.). The interconnection serves to exploit or avoid certain interferences; e.g., if the electric tailgate is closed, ventilation is decreased to avoid a pressure increase in the car.
- the device controllers of an assembly line: In automated factories the different production machines are interconnected in order to enable a "just-in-time" production of indivualized products.
- a sensor network for earthquake early warning: Whereas a single sensor node cannot make a solid statement about the epicenter and strength of an earthquake, a network of such nodes can predict the arrival of destructive waves in advance.
- a team of autonomous soccer robots: It is the declared target of the international RoboCup federation to have by 2050 a team of humanoid robots winning against a human team according to the usual FIFA rules. Already now there are annual competitions in this direction, with simplified rules.

3 Systems and Requirements Analysis

When constructing an embedded or cyber-physical system, the most important early phases are *systems analysis* and *requirements analysis*. Errors or omissions during these phases critically affect the complete project. Systems analysis is concerned with the design and construction processes of complex systems, and in requirements analysis processes are defined for the elicitation, management and linking of desired system properties.

Systems Analysis

The main problem in the development of "large" technical systems is the increasing complexity. Systems engineering tries to master this complexity by defining design and construction processes which take the whole development cycle into respect. All aspects of the system under construction are considered, both technical as well as non-technical ones such as user behaviour, commercial factors, operations, maintenance, and disposal. The subject of systems engineering is not limited to embedded or computational systems; e.g., also in building a new airport, systems engineering should be applied.

Systems analysis is the process of understanding, designing and developing a system as a whole (in contrast to the view of a system as a set of components). Esssential to this is the holistic view of the system requirements, in particular with respect to the integration and operation of the system in its socio-technical context. The main methods of systems analysis are to focus on system goals, to continuously explore several design variants, and to maintain a holistic view on processes and activities during the design and implementation.

> Systems engineering is an interdisciplinary approach and means to enable the realization of successful systems. It focuses on defining customer needs and required functionality early in the development cycle, documenting requirements, and then proceeding with design synthesis and system validation while considering the complete problem: operations, cost and schedule, performance, training and support, test, manufacturing, and disposal. Systems engineering considers both the business and the technical needs of all customers with the goal of providing a quality product that meets the user needs. [INC00]

A cyber-physical system admits several levels of abstraction or consideration. Sommerville [Som10] identifies five different levels (see Fig. 2): Hardware-, platform-, application-, process-, and organizational level. Each computational system is based on some *hardware* on which is running. This basic layer includes PCs, microcontrollers, processor boards, printed circuits, FPGAs, sensors, actuators etc. The next abstraction levels considers the *platform(s)* for the system, i.e., the basic software such as firmware, operating system, software libraries, middleware, protocol stacks, etc. Building on the platform level is the layer of *applications* or the *application software*, which realizes the user functionality and user interfaces. Above that, the *process* level is concerned with the technical and organizational conditions in which the system is operated. On top of that, all processes are performed by organizations, e.g., companies or public authorities; the *organizational* level considers activities within and interactions between organizations.

Computer engineering integrates several fields of electrical engineering and computer science to develop artefacts on the hardware and platform levels. *Software engineering* is about developing artefacts on the platform and application level. *Systems engineering* is concerned with the process and organizational level; the artefacts usually are not directly executable, but consist of models and formalizations of goals and circumstances. An engineer working on a particular level must have knowledge of the adjacent levels, in order to know what to rely on from the level below, and what to guarantee to the level above. A holistic view, as it is advocated in systems engineering, must consider relevant aspects from all levels.

As an example for this layering, consider the development of an artificial cardiac pacemaker for the human heart [Bos07]. The hardware mainly consists of a pulse generator with electrodes. Here, aspects like casing materials, form factor, battery life time, mechanical faults etc. have to be considered. When designing
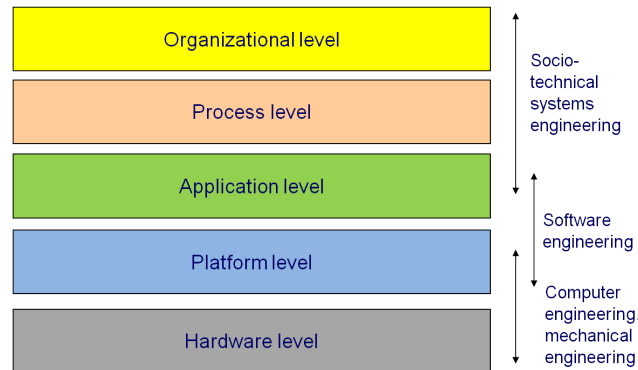
Fig. 2. Engineering levels for cyber-physical systems

the platform, operating system and software architecture must be decided. On the application layer the software for driving the pulse generation and communicating with the attending physician must be implemented. To this end, processes like implantation into the body, follow-up care, and long-term diagnosis are to be considered. These processes are executed in organisations like hospitals or doctors' offices. Without a certain understanding of the complete hierarchy, a project to develop a new pacemaker can not be successfully accomplished. As an example, it is necessary to understand the hospital's patient registration system in order to design a decent export function of the data-logging component in the pacemaker.

Requirements Analysis

Typical processes to be performed in systems engineering are the *stakeholder requirements definition* and the *requirements analysis* process: see [INC00]. In the stakeholder requirements definition, the needs and wishes of all people involved in the operation of the system under development are gathered. A stakeholder is "any entity (individual or organization) with a legitimate interest in the system" ([INC00]). This is a more general term than the common term "user". In our pacemaker example, stakeholders are the patient, doctor, surgeon, nurse, hospital, developing and producing company, supplier, maintenance technician, ambulance, administrative authorities, and others.

The *stakeholder requirements document*, also called *user requirements specification* or similar, describes the services to be provided to the stakeholders, and the operational conditions necessary for delivering these services. This document is the main reference for all subsequent developments. It should be written in a language which is easily understandable by all stakeholders, and describes the desired functionality from the viewpoint of the user / stakeholder. That is, implementation aspects are unimportant for the specification. However, for embedded systems, also the physical operational environment must be specified. On the

one hand, this means that environmental parameters like maximal admissible temperature or acceleration, sensor ranges etc. must be precisely described. On the other hand, the physical properties of the system under development must be part of the specification: maximal dimensions, plugs and connectors, etc. are also part of this document. In contrast to a software specification, the specification of an embedded system has to consider all aspects, including software, hardware, potential sensors, admissible tolerances and deviations, cost or power limitations, etc. However, no commitment to specific technical solutions is to be made; this is to be delayed to later development stages.

An example of an industrial stakeholder requirements document is the PACE-MAKER System Specification [Bos07]. This document has been made public to serve as a practical case study for scientific investigations. An in-depth modeling of certain aspects has been made in [KHCD13]. The excerpt in Fig. 3 describes part of the pulse generation functionality for rate-adaptive pacing. Subsequently, we will show how these requirements are reflected in the system's software design.

5.7 Rate-Adaptive Pacing: The device shall have the ability to adjust the cardiac cycle in response to metabolic need as measured from body motion using an accelerometer.

5.7.1 Maximum Sensor Rate: The Maximum Sensor Rate (MSR) is the maximum pacing rate allowed as a result of sensor control. The Maximum Sensor Rate shall be (1) required for rate adaptive modes, and (2) independently programmable from the URL (Upper Rate Limit)

5.7.2 Activity Threshold: The activity threshold is the value the accelerometer sensor output shall exceed before the pacemakers rate is affected by activity data.

5.7.3 Response Factor: The accelerometer shall determine the pacing rate that occurs at various levels of steady state patient activity. Based on equivalent patient activity: (1) The highest response factor setting (=16) shall allow the greatest incremental change in rate. (2) The lowest response factor setting (=1) shall allow a smaller change in rate.

5.7.4 Reaction Time: The accelerometer shall determine the rate of increase of the pacing rate. The reaction time is the time required for an activity to drive the rate from LRL (Lower Rate Limit) to MSR.

5.7.5 Recovery Time: The accelerometer shall determine the rate of decrease of the pacing rate. The recovery time shall be the time required for the rate to fall from MSR to LRL when activity falls below the activity threshold.

Fig. 3. Part of a pacemaker system specification

Many software projects fail due to an inadequate requirements analysis. There are some important properties a requirements specification must have. Such a document should be

- **understandable:** All stakeholders (not only the engineers involved) must be able to read and understand the document. Thus, e.g., it should contain a list of used terms and definitions, relevant standards, etc.
- **unambiguous:** When read by different stakeholders, the requirements must not leave any freedom of interpretation. With respect to the functionality, they must be precise, i.e., describing uniquely what the system shall or shall not do.
- **verifiable:** For each requirement, there must be a clear criterion whether it is correctly implemented in the final system or not. That is, it must be possible to design test cases which pass if and only if the requirement is satisfied. The test must be realistic, and the test result must be clearly defined.
- **complete:** The intended behaviour of the system under development must be fully determined by the specification. This includes the possibility of "leaving parts unspecified", if they are not necessary for the goals of the system.
- **consistent:** All requirements must be realizable, that is, must not be self-contradictory or impose insurmountable obstacles to an implementation. There must not be requirements which are in direct- or indirect conflict.
- **traceable:** For each requirement, it should be stated why it exists, who is the source of this requirement, where it is relevant, and how it contributes to the system's goals.
  Furthermore, it should be made clear what the significance of the requirement in the specification is, and which other requirements are connected to it.
- **abstract:** The requirements should focus on the stakeholder's perspective, not the developer's perspective onto the system. They should not unnecessarily constrain the implementation: If the same functionality can be realized in more than one way, this can be fixed at a later stage only.
- **adaptable:** The requirements document should be written in a way so that it can be modified and extended later on. All requirements should be largely independent, such that they can be deleted or replaced by alternatives, if this should turn out to be necessary.

There have been various methods proposed to achive a stakeholder requirements document which has the above properties. The first tasks during requirements engineering is the *requirements elicitation*, where the engineers communicate with all stakeholders to determine what their needs are. This can be supported by techniques such as guided interviews, brain-storming sessions, check lists, competitor analysis, etc.

The next step is the *requirements recording*, where the elicitation results are grouped and documented in a designated form, such as natural-language sentences, use cases, user stories, diagrams, or process specifications. Here, requirements management tools can be used which organize the items in a database of assets. Requirements are recorded as entries which can be searched for, selected, versioned, and linked to other entries. Many requirements management tools offer additional possibilities such as interaction to word processing and document generation software.

After the recording, the actual *requirements analysis* takes place. Here, the requirements are validated with respect to the above properties. Requirements which are unclear, ambiguous, not verifiable, incomplete or inconsistent must be re-written. Requirements where the origin or significance is unclear, which are written from a developer's perspective, or overlapping with other requirements should be modified. It is helpful to design some abstract models during this process, in order to apply academic tools such as model checkers, consistency checks, refinement and transformation.

The last part of the requirements engineering phase is the *requirements tracing*. This is an ongoing activity which spreads over the whole development cycle. Here requirements are classified, prioritized, and linked with other artefacts. It is important to use software tools for tracing the requirements in order to keep track of their evolution. In particular, since most embedded systems are produced in several variants, the possibility of reusing certain requirements must be investigated. Variant management in software product lines is an active ongoing research field [PBvdL05].

Requirements can be classified into three groups: system goals, scenarios and strategies [Poh10]. A *system goal* is the intentional description of a characteristic feature of the system under development. For example, one goal of the pacemaker system is the following.

> **PSG 2.1.1** *The pacemaker system supports the needs of patients that require bradycardia pacing support. ... It supports the recovery process of a bradicardiac heart (i.e., a heart beating too slow) by providing dual chamber, rate adaptive pacing support.*

System goals are a refinement and elaboration of the overall conception of the system, and act as a guiding star to other artefacts. Each development step and each developed artefact should be justifiable by a system goal. In this way, system goals can be used to identify irrelevant activities and to evaluate and choose different design alternatives. Usually, system goals are organized as AND-OR-trees, where topmost goals have as children the direct subgoals (AND-node), and each goal may have different alternative realizations (OR-node).

A *scenario* is an operational description of the way that the system achieves its goals, by means of a concrete example run. It consists of a sequence of steps both of the system and its environment or user. This way, a scenario is a concretization of (some of) the system goals, giving a step-by-step description of the actions and reactions which ends in the satisfaction or dissatisfaction of the goal. A common way to write down a scenarios is in a so-called *user story* or *use case description* [Coc01]. Fig. 4 gives as an example a use case description which isoperationalizing the requirements from 3.

A *strategy* is a high level plan to achieve some goals. In the context of cyber-physical systems engineering, a system strategy is the description of a plan to achieve a goal or to realize a scenario. Whereas a system goal determines *why* something should happen, and a scenario describes *what* should happen, a strategy answers the question *how* it should happen.

1. The pacemaker is in operating mode "permanent" with an operational pacing rate of f, where $LRL \leq f \leq URL$.
2. The patient moves with an activity rate below the threshold.
3. The patient increases the activity above threshold.
4. The pacemaker increases the pacing rate by the appropriate reaction factor.
5. The patient further increases the activity.
6. The pacemaker increases the pacing rate only up to the maximum sensor rate MSR.
7. The patient stops the activity.
8. Within the set recovery time the pacemaker reduces the pacing rate from MSR to LRL.

Fig. 4. Pacemaker scenario: use case "rate-adaptive pacing"

While formulating system strategies, it makes sense to distinguish between static (spatial) and dynamic (temporal) properties. Strategies can be seen under three different perspectives: the structural, functional, and behavioural perspective. The structural perspective forms a static viewpoint, whereas the functional and behavioural perspective focus on dynamic aspects of the system under development.

In the *structural perspective*, the composition of the system from parts, the relation between the individual parts, the data to be transmitted and processed, and the data attributes are described; as a catch-phrase: "this part sends data in such a format to that part". The *functional perspective* focusses on the transformation of data and information by the system; "this input signal is combined with that internal signal to yield those output signals". In the *behavioural perspective*, the reaction of the system to stimuli from the environment is described; "if the user does this, then the system does that".

For the formulation of strategies, various kinds of diagrams can be used. Typical diagrams for the structural perspective are block diagrams, as well as object and class diagrams. For the functional perspective, data and object flow diagrams are used. The behavioural perspective can be denoted with state-transition diagrams and activity diagrams. We will see examples for all three perspectives in the next section.

4 Modeling

Before actually building a technical system, it is good engineering practice to first construct a model. A model is a formal or semiformal representation of the system under development. It allows certain experiments to be performed even before the system is realized, thus providing early feedback and error-correction possibilities.

For the modeling of software, many different formalisms have been developed and are being used both in academia and industry. The software for cyber-physical systems differs from other software systems in that it has to interact

with a physical environment. Whereas for a computational system it is usually adequate to model it with discrete states, modeling formalisms for physical objects often include continuous dimensions. For a technical system containing both computational and mechanical components, models thus include discrete and continuous parts.

Subsequently, we will describe how to transform requirements for cyber-physical systems into system models using SysML, the systems modeling language. Then, we will show how to refine system models into continuous and discrete models, representing, respectively, physical and computational aspects of the cyber-physical system under development.

4.1 Systems Modeling

Modeling has been performed ever since people began constructing complex systems. The word stems from the latin "modulus", which is the unit or gauge according to which scale the pillars of a temple are made. That is, the model of a temple describes the relative dimensions of the different parts it is made of, in small size. This is an essential feature of a model: it shows only some aspect of an object under consideration, reducing its size or complexity. In general, a *model* is a reduced representation of some object.

This object either already exists (e.g., a children's toy model of a race car). In this case, the model is an *image* of the original. Or, it can be an prototype of something which is to be built (e.g., a small-size design model of a new car). Here, the model is a *pre-image* of the real thing. In general, there exists a *reduction mapping* between an object and its model, which preserves only some aspects.

The reduction in size is made since it is much easier to produce a model car than an actual car. The main functionality of a car (to transport people) is obviously lost in the mapping. However, other important properties like aerodynamic efficiency can be demonstrated on a model as well as on an actual car. In general, each model is a *purposeful* reduction: it is made for specific reasons and serves some defined purposes. For example, the purpose of aerodynamic model is to optimise air resistance in a wind channel, whereas the purpose of a toy model might be to win an RC car race. Thus, an aerodynamic model of a car will be very different from a model used in RC races.

When designing a model, great care has to be taken that the aspects which are important for the intended purpose are preserved under the reduction mapping, i.e., that the model faithfully represents the actual system. "Modeling errors", i.e., deviations between the modeled and actual behaviour within the represented aspect, usually make the whole model useless.

In models of complex technical systems, often the reduction in comparison to the actual system is not with respect to the physical size, but with respect to the logical complexity. An *abstraction* is a special reduction mapping which reduces the information content of a concept or an observable phenomenon, selecting only those aspects which are relevant for a particular purpose. A technical model is an abstraction of a system, which is used to demonstrate some function or behaviour of the system, to help in the construction of the system, or to enable or simplify

an analysis or investigation. Modeling a technical system is done by "leaving out unnecessary details", i.e., omitting structural or behavioural aspects which are not relevant.

Depending on the purpose, there might be several technical models of a system. E.g., for a building there might be statical models helping to calculate the stability, graphical 3D-models showing the architecture and facade, and floor plans giving detailed instructions where to build the walls. Consider, for example, an architectural floor plan. The abstraction function is then a simple mathematical scaling; and the property preserved under this scaling is whether objects overlap or not. Thats why the floor plan can be used for figuring out where your furniture can go in your new home, before you build it. Using computer models for the statics and appropriate architecture software, it is even possible to compare and modify different "virtual" buildings before actual construction work begins. Such analyses would be very hard or even impossible to do without models.

Subsequently, we use the following definitions:

Definition 3. *A* model *is a purposeful reduction of some existing or planned system. A* technical model *is an abstraction of a technical system, made for the purpose of demonstrating, constructing or analysing certain aspects of the system.*

Other definitions, which can be found in the literature, focus on special kinds or usages of models. For example, the aerospace standard DO-331 concentrates on the use of models in software construction and analysis: "A model is an abstract representation of a set of software aspects of a system that is used to support the software development process or the software verification process". The automotive standard ISO 26262 emphasizes the importance of models for requirements analysis and demonstration. It defines the process of building a model: "modeling is used for the conceptual capture of the functionality to be realised (open/closed loop control, monitoring) as well as for the simulation of real physical system behaviours (vehicle environment)".

For cyber-physical systems, which combine both physical and computational components, models fall into two classes:

- *Physical models* represent the operational environment and physical behaviour of the system under development. This includes models of mass and energy flow, the reaction of sensors to changes in pressure, heat, humidity, etc., the behaviour of actuators with respect to the supplied voltage, and so on. A model concentrates only on certain of these parameters (e.g., pressure and temperature), abstracting from all others physical actualities. Since "reality" is often considered to be continuous, physical models mostly use continuous elements and variables. An examples for a physical model is a system of differential equations over real numbers describing the temperature of a gas in a combustion chamber in relation to the applied pressure.
- *Logical models* are computer diagrams representing the computational parts of the system. They are used to model the *structure*, the *function* and the

behaviour of the system under development. This includes the decomposition into modules, the data structures used, the message exchange protocols, etc. Logical models abstract from implementation details of the software, such as the used programming language, the internals of certain library functions, or the contents of certain variables. Since computation is mostly considered to take place in discrete steps, logical models normally use states variables over countable domains. An example for a logical model is a finite automaton translating input sequences of a's and b's into sequences of 0's and 1's.

Modeling of physical systems by differential equations has been practised by engineers for hundreds of years. Compared to that, software modeling is a relatively new discipline. Since the beginning of software engineering, the need has been recognized to deal with the ever increasing complexity of computer software. Thus, many different formalisms have been proposed to model software artefacts. Amongst these are classical formalisms for modeling the behaviour, such as flowchart diagrams and Nassi–Shneiderman diagrams; finite automata, labelled transition systems and state machines; Petri nets and activity diagrams; StateCharts, message sequence charts, etc. For modeling the structure of software, different sorts of architecture and component diagrams have been suggested.

Models denoted in one or several of such formalism have traditionally been used to document and visualize large software systems. A relatively new idea, however, is to use models also as "first-class citizens" in an embedded systems design process. For software systems, *model-based development and analysis* is a means to reduce the design complexity by using a model of the system. Traditional software design methods are usually ordered in stages such as stakeholder requirements definition and analysis, architectural design, module design, implementation, debugging and testing, system integration, installation and operation. Fig. 5 gives a graphical representation of such a classical V-shaped cyber-physical engineering process. The horizontal arrows indicate that the artefacts on the left (constructive) half of the "V" correspond to those on the right (analytic) half.
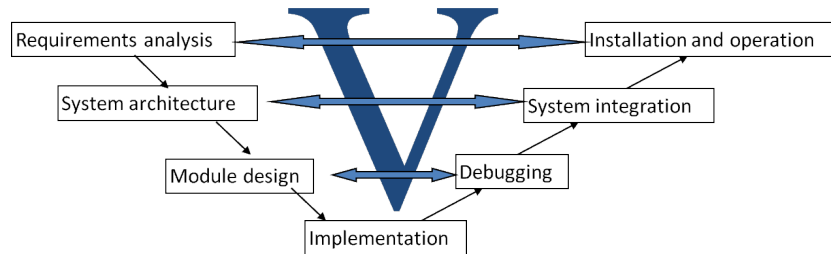
Fig. 5. A classical V-shaped engineering process

If different types of artefacts are being used as workflow results between the different design and analysis stages, gaps in the overall process appear. Stakeholder analysis produces a requirements document, which is used as the input for function and control system design. The resulting system specification document is used by programmers to implement software modules, which in turn are integrated into software systems. These systems are tested in various test environments and deployed onto the target hardware.

In a traditional development process, all these stages use different notations and formalisms for their work results. For example, a control system may be described via circuit diagrams, whereas its software may be implemented in the C programming language. These different notations lead to misunderstandings and to the introduction of errors between the phases. Moreover, different tools have to be used for each stage; thus, there are frequent incompatibilities between the artefacts, and no continuous work flow in the development is possible. Since the software system is available at a late development stage only, also testing can start only late, which leads to high costs for error correction.

Model-based development (also called *model-based design* or *model-based engineering*) is a paradigm which mitigates these deficits. In model-based development, there is one *system model* which is the central artefact of the whole design process. The development steps consist in transformation and enrichment of this system model. ISO 26262 defines *model evolution* to be the "evolution of the functional model from an early specification model via a design model to an implementation model and finally its automatic transformation into code".

Model-based development can be described by the following process steps. The system model is built at the earliest possible time, from the stakeholder requirements specification. Then, it is transformed and augmented with additional information, such that parts of the model can be executed. A *virtual prototype* can be derived from this initial system model, which is used to simulate and validate the behaviour of the target system, even before it physically exists. The main development process consists of a stepwise refinement of the system model to an *implementation model*. From the implementation model, executable code is generated automatically. If necessary, the target code is augmented with additional code (e.g., special library routines), and deployed onto host- and target platform.

Between the refinement steps, tests and simulations are applied frequently, to assure that each step preserves the desired behaviour. To this end, a *test model* can be developed from the requirements which is independent from the system model, and from which test suites are obtained by *model-based test generation*. The test cases are executed with respect to the system model (*model-in-the-loop testing*, MiL), with respect to the generated code (*software-in-the-loop*, SiL), and with respect to the target hardware (*hardware-in-the-loop*, HiL). Additionally, in highly safety-critical systems, the requirements can be formalized to a *logical specification* and the system model can be verified with respect to these formulas. (This process has been called *model checking*). Fig. 6 displays the various artefacts and activities which can occur in model-based design.
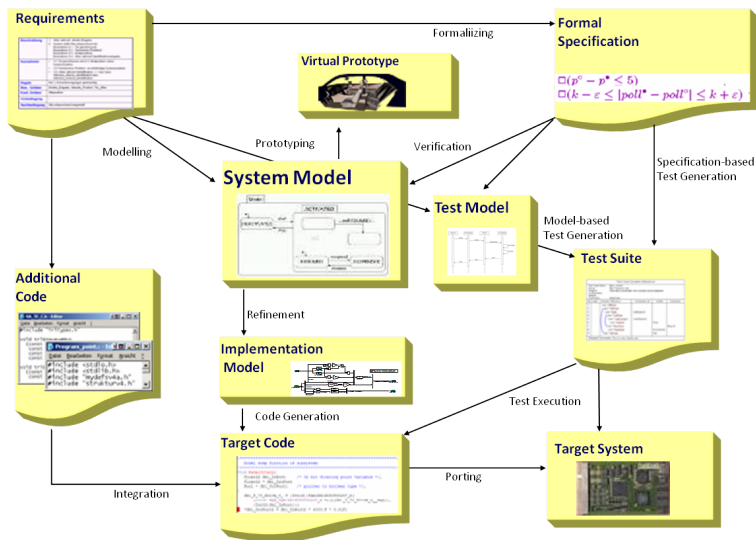
Fig. 6. Artefacts and activities in model-based development

Model-based development has shown to yield a significant increase in productivity and quality, reducing both the development time and the number of errors in the design [PHAB12]. The possibility for early demonstration and simulation of the systems' functions helps to detect specification errors. Virtual prototyping yields a better understanding of the functionality to be developed. Automatic code generation speeds up the time-consuming coding process, and continuous testing on all development stages avoids design errors. The system model serves as a basis for the technical documentation which is required for homologation of safety-critical systems. Therefore, it is expected that in the future this technique will be the major paradigm for cyber-physical systems engineering.

In order to effectively apply the model-based design technique, a modeling language is needed which supports as many phases of the process as possible. UML, the unified modeling language ([OMG15b]), has been standardized by the Object Management Group (OMG) in order to harmonize and combine different modeling notations. For systems engineering, it has been augmented by SysML, the systems modelling language ([OMG15a]). Furthermore, several variants and extensions of UML called *profiles* have been defined such as MARTE for the Modeling and Analysis of Real-Time and Embedded systems.

The goal of these standards is to provide a uniform, unique description language for (almost) all artefacts in the model-based design process. UML and SysML are targeting a common language basis for all stakeholders in cyber-physical systems engineering, in order to make continuous tool chains possible. Such an integrated tool chain would support all of the above mentioned model-

based development activities, providing a seamless integration of artefacts and work result.

In particular, this involves engineers from mechanical and electrical design as well as software engineers. That is, an integrated development environment (IDE) for cyber-physical systems engineering would have to support concepts such as differential equations, flow diagrams, and state-transition systems. Furthermore, it would have to help maintaining the consistency of all artefacts throughout the whole process, as well as facilities to migrate and evolve models. Finally, it would enable quality assurance of both the model and the automatically generated code. Unfortunately, up to now this goal has only been reached partly. Although tool providers and researchers are working towards this ultimate goal, today there are still different languages and tools for different phases of the process.

SysML strives to integrate requirements engineering into the modeling activities. In principle, stakeholder and system requirements can be given in a variety of formats: as textual contract specification, as use-case descriptions, as algebraic or logical formulas, as component descriptions with pre- and postconditions, as state diagrams or automata, or even code and pseudo-code. This is a spectrum of possibilities, where in practice even combinations and profiles of the above mentioned formats are being used today. In order to deal with this situation, SysML provides the concept of a *requirement diagram*. These are particular model elements which provide a connection between informal and formal notation. Requirement diagrams allow to integrate textual requirements into a formal model. They are used to model the content and structure of a stakeholder or system requirements document.

Each UML/SysML model consists of *elements*, which are connected by *relations*, where relations themselves are model elements. Relations may be directed and contain multiplicities; typical directed relations are

- **generalisation**, characterizing the relation between a specialized element and its general classifier, and
- **composition**, characterizing the relation between a whole and its parts.

UML contains several other relations between elements such as associations, dependencies, inclusions, realizations, etc. In SysML, a requirements diagram contains one or more *requirement* elements. The <<requirement>> stereotype characterizes a named textbox which may include an identifier, the text of the requirement, and additional properties (such as the requirement category or its verification method). For requirements, the following relations can be used in addition to the ones above:

- **include**: This relation is drawn between requirements R_1 and R_2 to indicate that R_2 is a sub-requirement of R_1. For example, the requirement **5.7** on rate-adaptive pacing in Fig. 3 above includes requirement **5.7.1** about the maximum sensor rate.
- **derive**: This relation indicates that R_2 is a logical or physical consequence of R_1 . For example, the requirement that the pacemaker device includes

an acceleration sensor is derived from the requirement that it shall provide rate-adaptive pacing.

- **refine**: R_2 is a refinement of R_1, if it describes the same content, but with more detail than R_1, possibly eliminating some choices in the design space. For example, the requirement **5.7.1.2** that the MSR shall be independently programmable from the URL might be refined by a requirement **R** saying that there must be a variable `msr` in the non-volatile memory which can be set by the physician via the device-controller monitor.
- **verify**: A requirement may be verified by a test case. For example, the refined requirement **R** from before could be verified by a test case in which the variable `msr` is set to a certain value.
- **satisfy**: A requirement may be satisfied by a particular model element describing its implementation. For example, the requirement **R** could be satisfied by an object diagram describing the non-volatile variables of the pacemaker.

In Fig. 7, a breakdown of the requirements from Fig. 3 in the open-source tool Papyrus ([Ecl15]) is presented. Papyrus builds on the Eclipse IDE, and is freely available.
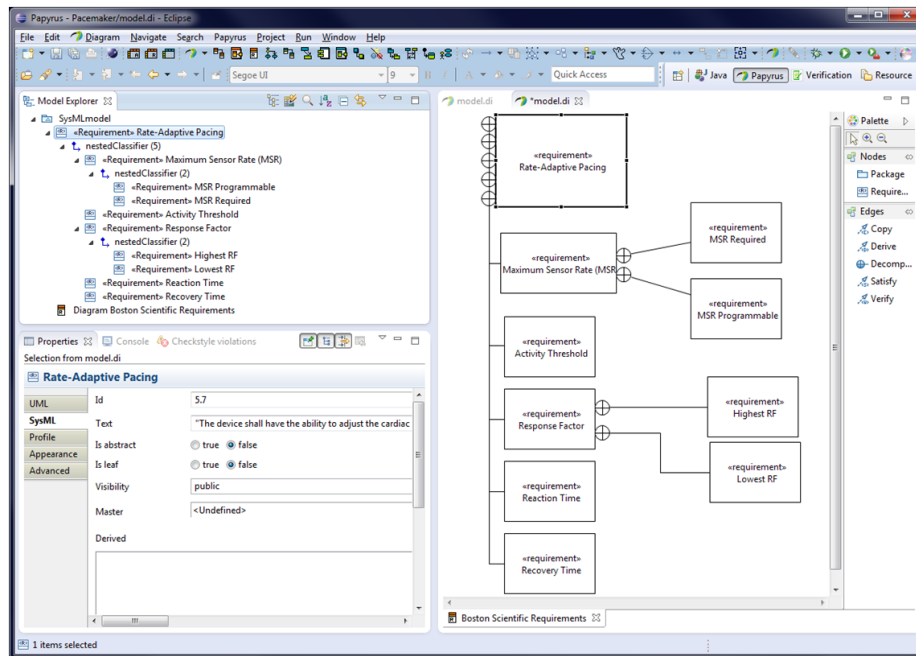
Fig. 7. Requirements diagram for the pacemaker in the Papyrus tool

4.2 Modeling of Continuous Components

Cyber-physical systems differ from pure computational systems in that they interact with a physical environment. For model-based development of the (software for the) embedded systems which constitute a cyber-physical system, we need a model of the physical environment and/or the technical system in which it is working. In some cases, this model can be rather trivial. For example, assume that the system has sensors to observe its environment. Then the simplest environment model is one where all sensors may give arbitrary values (within their respective ranges) at all times. As another example, assume we want to model a user operating the system with knobs and buttons. Then a very simple model is the one which expresses that the user may press any button or choose any knob setting at any time.

However, such a model might not be very useful. It ignores physical and logical dependencies within the environment. Firstly, the value of a physical observable in the environment usually can not change arbitrarily — often it is a *continuous* quantity. For example, the outside temperature just doesn't change instantaneously from minus 20° to plus 40° Celsius. Even with discrete parameters, physical side conditions may impose restrictions on the user behaviour. It is simply not possible to turn a knob from setting 1 to setting 5 without going through the intermediate settings 2, 3, and 4. Secondly, one physical parameter may rely on another one. For example, in a closed container, the temperature of a gas is proportional to the pressure applied to it. Or, a touch sensor may give a signal only if a certain height has been reached. Thirdly, and most importantly, the outputs of the embedded system affect the behaviour of the technical system in the environment, and thus also the inputs which the system gets from its environment. For example, turning on a motor might increase the pressure onto a gas and thus its temperature which is read by a thermo sensor. This type of feedback is called a *control loop* – the embedded system is controlling the technical environment, which in turn influences the behaviour of the embedded system.
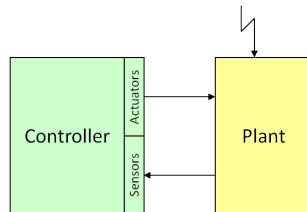
**Fig. 8.** Feedback loop between controller and plant

Control theory is a discipline traditionally concerned with such control loops between continuous input and output magnitudes. In control theory, the embed-

ded system is called a *controller*, and the enclosing technical system is called the *plant*. The plant is a *dynamical system* which is described by some continuous parameters. These parameters are influenced by the outside world. Mathematically, a parameter in a dynamical system is just a continuous function from time points (real numbers) into values (real numbers). Some of the parameters can be observed by the controller via sensors, and some can be controlled via actuators. The environment imposes a disturbance on the controlled parameters, and the goal of the controller is to bring them back to some admissible values. Fig. 8 above displays this basic control theory paradigm; note that it is just another visualization of the same actual situation as presented by Fig. 1.

The above discussion hopefully made clear that in order to design cyber-physical systems, there is a need for modeling formalisms with which the environment of an embedded system can be described. This "material world" to a large extent can be seen as a dynamical system, where measurable quantities such as lenght, volume, temperature, etc., evolve over time. To model such dynamical systems, concepts are needed such as the continuous change of some variable, the continuous sum of one or more flows, and the continuous dependency of one value upon others. In mathematics, real-valued functions, integrals, and differential equations provide such concepts. That is, mathematically the physical world can be described by a set of real-valued functions over time. In engineering, several other formalisms have been invented which build on mathematical modeling via differential equations, and which are generally accepted. Amongst these are several forms of diagrams such as electrical-circuit diagrams, fluid-mechanics diagrams, process-flow diagrams, and functional-block diagrams.

For computer science, no similar formalism is generally accepted. The main questions are how to come up with a convenient modeling notation for the relevant physical aspects of a cyber-physical system, and how to integrate it with computational modeling notations. Although SysML contains various mechanisms for dealing with continuous flows, there is not yet an adequate tool support to use them in an academic environment. In the industrial context, tools like Simulink (from MathWorks), Simplorer (from Ansys), LabVIEW from (National Instruments), Ascet (from ETAS), and Dymola (from Dassault) are being used for modeling dynamical systems. In these notes, we use Scicos, which is a free graphical modeling and simulation tool. Scicos allows *block diagrams* to be drawn which describe continuous flows. It can evaluate the diagrams with a numerical solver, and plot the resulting functions. Furthermore, it contains a code generator to compile these models into executable code.

As an example, in this section we will use a simple water tank with gain and drain valves, as depicted in Fig. 9. This example is representative for a large class of controlled systems such as heaters with thermostats, batteries with chargers, lights with dimmers, etc. It is also more intuitive than the pacemaker example since it requires only general knowledge of physical contexts and differential calculus. Subsequently we will model several variants of this system using block diagrams.
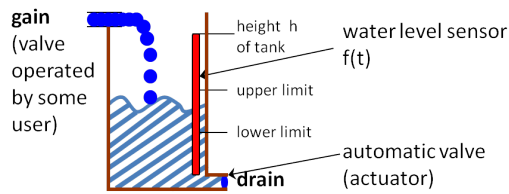
Fig. 9. Schematics of a water tank

A block diagram consists of a set of *blocks*, where each block has a dedicated number of *input* and *output ports*. Blocks without input are called *sources*, blocks without output are called *sinks*. In the diagram, each output of a block must be connected to at least one input of another port, and each input must be connected to some output. These connections are the signals or data values that flow between the blocks. Sources are, amongst others, a constant generator (a block with one output generating a constant value on this output), a ramp (generating a linear function), or a sinusoid generator (generating an output in the form of a sinus wave). Typical sinks are the various "Scope" blocks for displaying the input signal(s) on screen. Blocks which have both input and output ports are, for example, adder, multiplier, integrator and differentiator. In commercial tools, there is usually a huge library of blocks. Often, these are specialized for a particular domain, e.g., the modeling of temperature distibution in combustion engines, or the simulation of water flow in hydraulic machines.

As an example, consider the block diagram in Fig. 10. This diagram contains two sources (*gain* and *drain*) and one sink (the scope). The sources are arbitrary piecewise linear functions, which are multiplied by constant factors. The lower signal is subtracted from the upper, and the difference is integrated over time. The result $f(t)$, as displayed by the scope, is shown below the diagram. The diagram can be seen as a rough model of the water tank, with input pipe and output pipe. The different diameters of the pipes are modeled by the constant multiplication factors $c_1 = 3$ and $c_2 = 2$; opening and closing of the valves is modeled by the randomly chosen functions *gain* and *drain*. Function $f(t)$ then represents the resulting filling level of the tank; for the moment we abstract from the fact that each real tank has a limited capacity and can not contain a negative amount of water.

If we evaluate ("simulate") the diagram, the scope shows the function depicted in the lower part of Fig. 10. This evaluation is equivalent to the numerical solving of a system of ordinary differential equations (ODEs) Each block diagram using only standard mathematical blocks can be transformed into an ODE as follows. We give each signal/connection a unique name. Then we equate the output of a block to the function it represents, applied to the inputs of the block. In the example of Fig. 10, this yields $v_0 = gain$, $v_1 = c_1 * v_0$, $v_2 = drain$, $v_3 = c_2 * v_2$, $v_4 = v_1 - v_3$, $v_5 = \int v_4$ Letting $v_5 = f(t)$ and elimination all variables
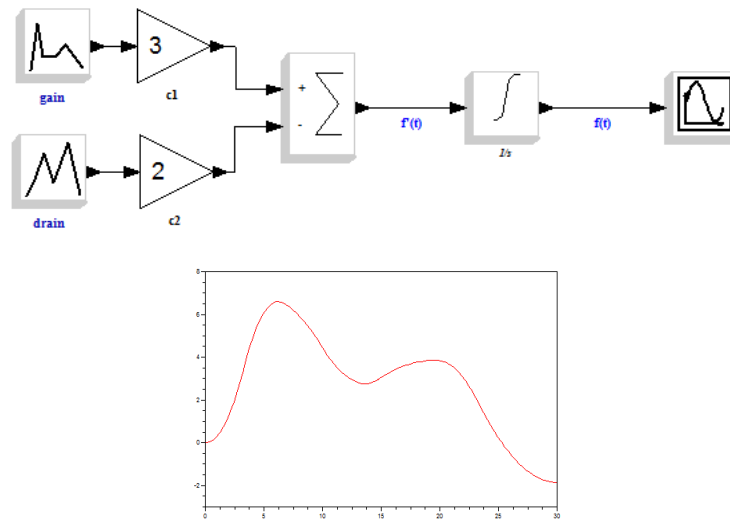
Fig. 10. A simple block diagram and its evaluation result

but v_4, we get

$$(*) \quad f'(t) = (c_1 * gain(t) - c_2 * drain(c)).$$

The translation works also in the other direction: each ODE can be represented by a block diagram. Instead of defining the general procedure, we just give an example. In our water tank, let $g(t) = (c_1 * gain(t) - c_2 * drain(t))$ denote the net flow in or out of the tank, and let h be the height of the tank. We refine equation $(*)$ as follows.

$$(**) \quad f'(t) = \begin{cases} \max(0, g(t)), & \text{if } f(t) \leq 0, \\ \min(0, g(t), & \text{if } f(t) \geq h, \text{ and} \\ g(t), & \text{else, i.e., if } 0 < f(t) < h \end{cases}$$

The first case defines what happens if the water tank is empty ($f(t) \leq 0$): In this case, water can only flow in; that is, if $g(t) > 0$ then $f'(t) = g(t)$, else $f'(t) = 0$. Similarly, the second case defines the filling of a full tank: In this case, water can only flow out; that is, $f'(t) = g(t)$ if $g(t)$ is negative, else $f'(t) = 0$. (You could imagine that opening the drain on an empty tank has no effect, whereas opening the gain on a full tank makes the water spill over.) Finally, the third case defines the behaviour of the tank if it is neither empty nor full; in this case the filling level changes as indicated by $(*)$.

Fig. 11 gives a block diagram for $(**)$. In this diagram, the case distinction is done by two switches. A switch is a block where the output is either the first or third input, depending on whether the second input is larger than a threshold. The MIN and MAX blocks work as expected. The height h is set in the right

switch-block to $h = 8$; note that in the evaluation (with the same *gain* and *drain* as before) the filling level $f(t)$ stays between 0 and 8.
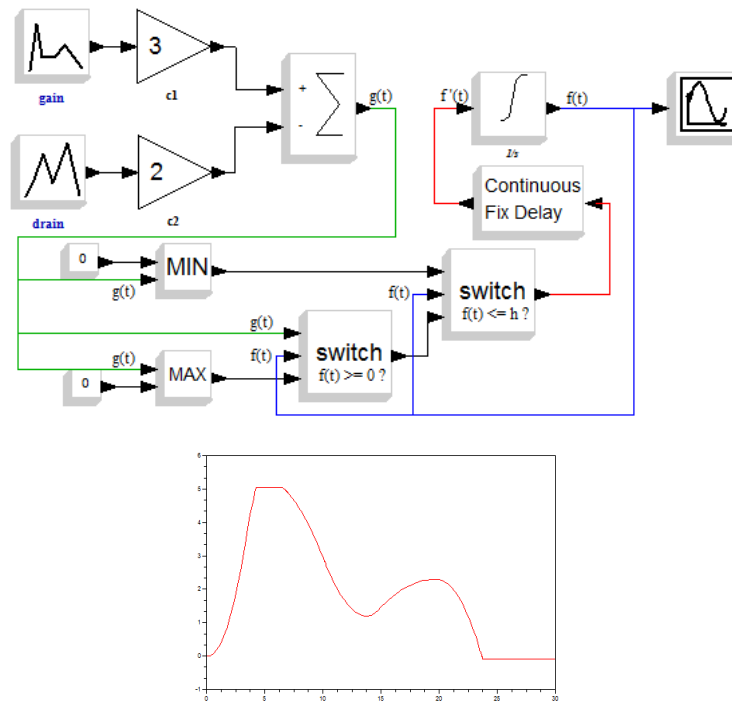
Fig. 11. Block diagram modeling a water tank

Equation $(**)$ is a "recursive definition" of the function f: the value of $f(t)$ is used in the definition of $f'(t)$. Such a recursive definition leads to a loop in the corresponding block diagram, as in Fig. 11. In order to avoid an "unguarded recursion", we have to put a delay block between definition and use of the signal f. This is necessary for the numerical evaluation of the diagram. In the evaluation, the solver tries to determine a numerical value (a real number) for each signal at each time point. If there is an "unguarded" loop, it means that the value of a signal f at a given time point depends *on itself* and cannot be determined. Putting a delay in the loop means that the value of f depends *on an earlier value* of f; thus it can be determined by calculating the values of all signals from the beginning in fixed steps. Of course, one should be aware that this calculates only an *approximation* to the solution of the corresponding differential equation. In general, differential equations need not have solutions; for example, consider

$$(***) \quad f'(t) = \begin{cases} -1, \text{ if } f(t) \geq 0, \\ 1, \quad \text{ if } f(t) < 0 \end{cases}$$

There is no function f defined on any subset of real numbers which is a solution to this equation. The numerical solution will oscillate depending on the chosen step size for the delay.

Block diagrams can also be used to model the behaviour of a controller. As shown in Fig. 8, a controller is a computational system influenced by and determining the behaviour of its technical environment, the plant. As an example, we consider our water tank as the plant which is regulated by a controller. That is, we extend our model of the water tank by a controller model. Since the drain valve is operated by the controller, it is no longer modeled by an arbitrary function; the *drain* signal is output from the controller to the plant. The gain-valve, which still is under control of some external user, and the rest of the tank model are left unchanged.

We assume that the controller can observe the actual water level by a sensor; thus, the signal $f(t)$ is fed from the tank model to the controller model. Consider the two marks *upper* and *lower*, mounted at the desired upper and lower filling level. The task of the controller is to keep the water level between the lower and upper mark, no matter how the gain-valve is opened or closed by the user. A simple strategy the controller could follow is
  – if the water level exceeds the *upper* marking, open the drain, and
  – if the water level falls below the *lower* marking, close the drain.
As mathematical formulas, this strategy could be written as

  – if $f(t) \geq upper$, then $drain(t) = 1$
  – if $f(t) \leq lower$, then $drain(t) = 0$
  – if $lower < f(t) < upper$, then $drain'(t) = 0$

A block diagram for this solution (with $lower = 3$ and $upper = 5$ is shown in Fig. 12. In this diagram, we have collapsed the model of the water tank from Fig. 11 into a *superblock*. The possibility to abstract several blocks into one is a very important structuring mechanism which can help to make large block diagrams more understandable.

Note that this simple controller keeps the water level only "approximately" between the desired limits; depending on the reaction time of the drain valve there may be over- or undershootings of the limit. More complex type of controllers can be modeled in this way, including so-called PID controllers, which can maintain the desired target value very accurately.

4.3   Discrete-state Modeling

Cyber-physical systems are networks of computational systems operating in a technical environment. Physical objects in this environment are mostly characterized by continuous parameters such as shape, size, position, movement, temperature, pressure, voltage, etc. Notable exceptions to this rule are switching elements such as mechanical switches, bistables, relays, and transistors. These can be seen as taking, at any given time, one of two or a finite set of possible values. In contrast, computations are usually discrete processes. The reason is
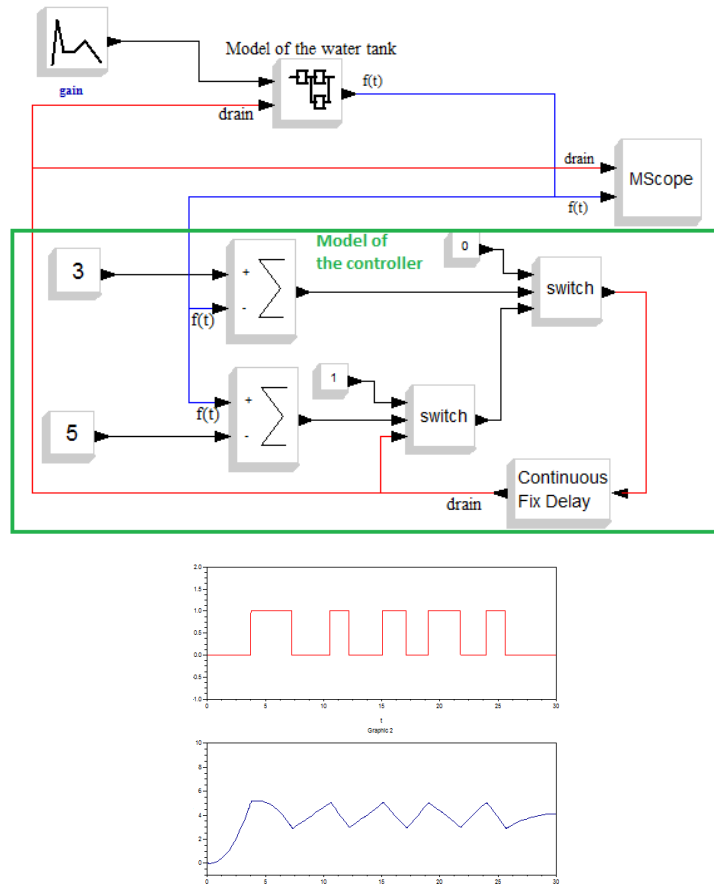
Fig. 12. Water tank with a simple controller

that present computers are largely built from switching elements. Thus, they can only execute programs in discrete steps, and they can represent data only in a discrete, finite way. (There was a time when people experimented with continuous or hybrid computers, but those days are long gone...) Thus, each computer can assume only a finite (although very large) number of states, and each process executed by a computer can go only through a finite or countably infinite number of states.

Many different formalisms have been proposed for the description of discrete processes and machines. Most of them are based on the notions of state and event. A *state* of a system is a mapping of its parameters to values. An *event* is an instantaneous change in some state component(s), causing a *transition* between states. Mathematically, an event is a discontinuity in the trajectory of some variable(s).

Formally, a *state-transition system* (or simply *transition system*) consists of an alphabet $\mathcal{A}$, a set S of states, a set T of transitions, and an initial state s_0. Each transition between two states is labelled by a symbol from $\mathcal{A}$. A *run* or *execution* of a state-transition system is a finite or infinite sequence of states from S starting from the initial state s_0, where each pair of adjacent states is related by a transition from T.

For example, reconsider the pacemaker specification from Fig. 3. From Requirement 5.7.2, we learn that there exists an activity threshold which determines whether the pacing rate is affected by activity data or not. This can be modeled by a state-transition system as shown in Fig. 13.
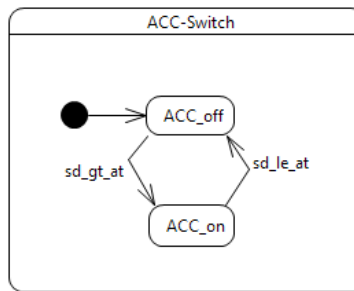
Fig. 13. A simple switch

In this transition system, there are two states, `ACC_off` and `ACC_on`, determinining whether the accelerometer sensor data shall be taken into account or not. The alphabet $\mathcal{A}$ consists of only two symbols: $\mathcal{A} = \{\texttt{sd\_gt\_at}, \texttt{sd\_le\_at}\}$. The transition from `ACC_off` to `ACC_on` occurs when `sd_gt_at` happens, that is, when the sensor data becomes greater than the activity threshold. Likewise, when the sensor data becomes less or equal to the activity threshold, `sd_le_at` happens and a transition from `ACC_on` to `ACC_off` is performed. Initially, the transition system is in state `ACC_off`. Therefore, there is only one possible run of the system:

$$\texttt{ACC\_off} \xrightarrow{\texttt{sd\_gt\_at}} \texttt{ACC\_on} \xrightarrow{\texttt{sd\_le\_at}} \texttt{ACC\_off} \xrightarrow{\texttt{sd\_gt\_at}} \texttt{ACC\_on} \xrightarrow{\texttt{sd\_le\_at}} \ldots$$

Historically, state-transition systems have been investigated as Moore- and Mealy-machines, finite automata, Petri nets, neural networks, StateCharts, and others. Traditional research questions have been the relation of these formalisms to formal language theory, logic, and term rewriting. Mainly, the *expressiveness* and *complexity* was in the focus of research. Their potential as a modeling language in a model-based development process has been emphasized with the advent of UML [OMG15b]. The "Unified Modeling Language" has been defined in order to combine and unify several of the above formalisms. In its current

version 2.5 (Mar. 2015), it contains 14 types of diagrams, for modeling both the structure and behaviour of computational systems.

State Machines are the UML diagram type which is closest to state-transition systems. In fact, our above Fig. 13 depicts a valid UML state machine, drawn with the Eclipse Papyrus Tool. However, in UML, the alphabet of state machines can be structured: each transition can have a number of triggers, a guard, and an effect. A trigger of a transition can be any event, e.g., the receipt of a message, or the execution of a message. The guard can be any boolean expression, formulated, e.g., in the Object Constraint Language OCL. The effect of a transition can be any behaviour, e.g. an assignment, an event, or even a state machine behaviour. A transition from s to s' which has event e as trigger, condition c as guard, and action a as effect is depicted as

$$\boxed{s} \xrightarrow{\;e\,[c]\,/\,a\;} \boxed{s'}.$$

Empty triggers, guards, and effects can be omitted. Thus, in Fig. 13, `sd_gt_at` and `sd_le_at` are events which occur due to an action of some outside component, and there are no transition guards and effects.

Besides the concept of "state" and "transition" UML state machines include concepts to include data and to structure states via hierarchies and parallelism. Let us explain these concepts via an example. Requirement 5.7.3 in Fig. 3 declares that there is a response factor for the accelerometer-induced increase of pacing rate. This factor has settings between 1 and 16. Assume that the setting can be increased by the event `inc` and decreased by `dec`. Then we can model this with 16 states, as shown in the upper half of Fig. 14 (drawn with the Eclipse Yakindu Statechart Tools).
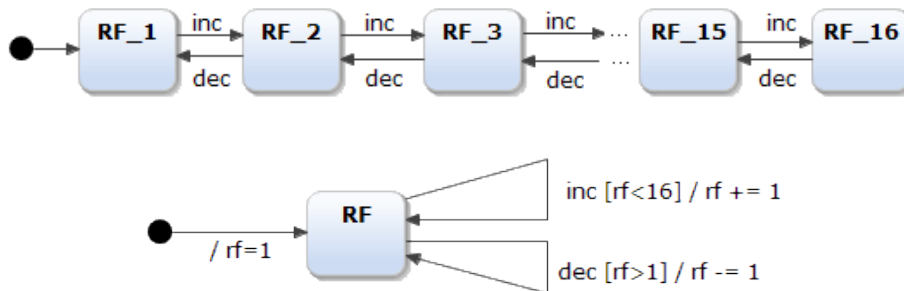
Fig. 14. Two different models of a counter

However, a more concise model can be obtained as shown in the lower half of this figure. Here, `rf` is an object of type `int`, which is set to 1 by entering the initial state, incremented upon event `inc` by the action `rf += 1` if its value

is less than 16, and decremented upon `dec` by `rf -= 1` if its value is greater than 1.

For parallelism and hierarchies, we note that a UML a state machine consists of a set of regions, where each region contains states and transitions. All regions in a state machine are executed in parallel. A state itself can be simple or composite, where a composite state is one which again contains a number of regions. This way, a state can contain states, which contain states, etc.; this arbitrary nesting is similar to the nesting of blocks and superblocks which we have seen in Fig. 12.

As an example, consider the model in Fig. 13. Assume that there is a process `sdac` (sensor data acquisition) which is triggered in fixed intervals and reads the current activity level from the sensor into variable `sd`. Assume further that in order to smoothen this sensor reading, we are to switch into state `ACC_on` only if two consecutive sensor readings have been above the threshold, and likewise for state `ACC_off`. Then we can extend the model as shown in Fig. 15. Here, the events `ACC_off` and `ACC_on` are generated by the parallel region at the right if two successive readings are above or below threshold, respectively.
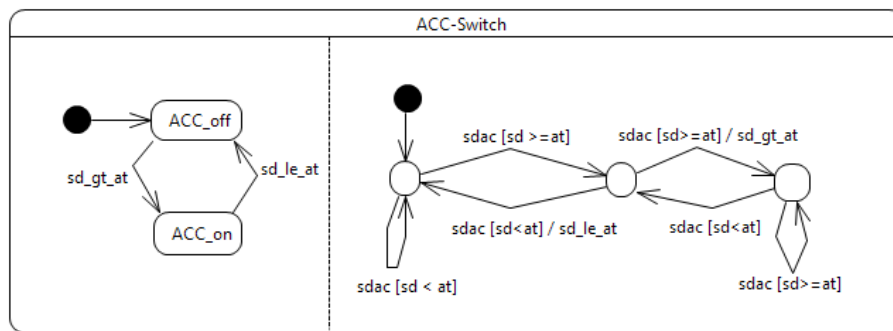

Fig. 15. Two parallel regions

5   Model Transformation and Code Generation

A major advantage of a formal model, in cyber-physical systems engineering, is that it enables the engineer to automatically generate code from it. A prerequisite for this is, of course, that the model is treated as a "first-class citizen" in the development process. That is, the model is not just a means of documentation and illustration, but is on the same level of importance as, e.g., requirements, code segments, and test cases. Syntactic and semantic correctness of the model must be ensured similarly to the development of code. Furthermore, the model

must be integrated, maintained, put under version control, and evolved, as the system is being further developed.

Code generation from a model is a special type of *model transformation*. Here, executable code according to some programming language syntax is generated from a syntactically correct model. The transformation is described via the modeling concepts. The situation is similar to that of a classical compiler, which translates programs from a high-level programming language into machine code. The actions of the compiler are described via the syntax of the source language. Similarly, a code generator can be considered as a "model compiler".

5.1 Code Generation from Block Diagrams

According to the two kinds of models which we have met, block diagrams with continuous flows and state-machine diagrams with discrete transitions, there are two major ways to generate code. For continuous models, numerical solvers are employed which construct an approximation to the trajectories of all signals in the model.
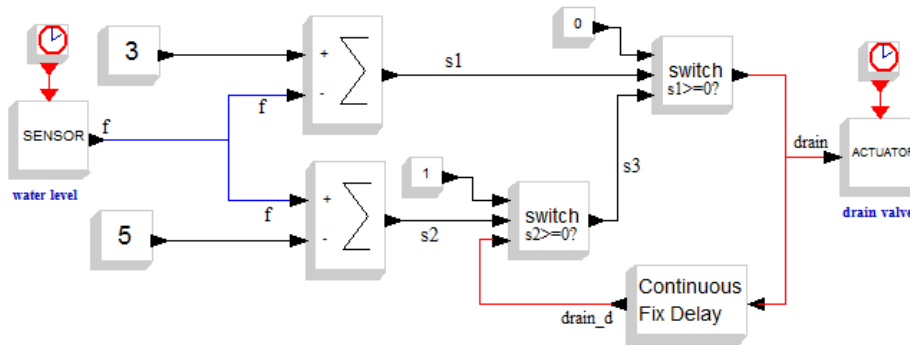
Fig. 16. Controller model for the water tank

Consider a block-diagram model of a controller, e.g., for our water-tank simulation. Fig. 16 is the boxed (control) part of Fig. 12, where we replaced the feedback loop from and to the model of the water tank by a sensor and an actuator. The sensor represents an input function $f(t)$ to the controller, viz. the observed water level of the tank at a given moment. The actuator is an output function $drain(t)$ of the controller, viz. the controlled setting of the drainage valve at time t. Both of these models are time-triggered, i.e., they are read and written at designated times only (indicated by the clock input). This controller model has internal signals (s1, s2, s3, drain and drain_d) connecting the blocks, plus certain constant parameters. The code generator will generate code from this model which calculates, for a given time interval $[0, t_{\max})$ the values of the

output function from the values of the input function. It does so by starting with the initial values at time $t = 0$, and incrementing the time t in discrete steps. The values in between these steps are numerically approximated. In general, the translation follows the scheme

```
initialize all signals (default value is 0);
for (t:=0; t< t_max; t+=t_step)
    for all blocks
        out-signal := block-function(in-signals)
    end for all blocks
end for
```

For the evaluation of the **for all**-statement, the code generator has to construct an order of the blocks such that all in-signals are computed before the block-function itself is called. For a source block (e.g., a sensor), the block function has no input arguments. For a sink block (e.g., an actuator), the block function returns *void*. The calculation of the block function may involve previous values of this function, which must be stored in a buffer. As an example, the output of an integrator block can be approximated by

out-signal($t+t_{\mathrm{step}}$) := out-signal(t)+t_{step}*in-signal(t)

This method was first described by L. Euler in 1768; meanwhile, more exact mathematical methods have been developed. As another example, consider the `continuous fix delay` block which realizes a time shift of the input signal. It can be approximated by storing "sufficiently many" values of the input signal in a queue and outputting the earliest one. The following pseudo-code roughly describes the code generated from the controller model in Fig. 16; the actual code contains several optimizations.

```
s1 := 0; s2 := 0; s3 := 0; drain := 0; drain_d := 0;
for (t:=0; t< t_max; t+=t_step)
    if (trigger1) then f := sensor_read;
    s1 := 3 − f;
    s2 := f − 5;
    s3 := if (s2 ≥ 0) then 1 else drain_d;
    drain := if (s1 ≥ 0) then 0 else s3;
    drain_d := buffer1 . dequeue (); buffer1 . enqueue (drain);
    if (trigger2) then actuator_write(drain);
end for
```

The maximal simulation time t_{max} and step size t_{step} are parameters of the simulator. On a host computer, the maximal simulation time usually is finite, whereas for execution on an embedded target system, the upper limit usually is infinite. During a simulation, the step size may be adjusted in the main loop according to the dynamics of the system. For code which is to run on an embedded system, the step size should be chosen according to the speed of the target processor.

5.2 Code Generation from Transition Systems and State Machines

When programming an embedded system, an often used scheme is the so-called *simple control loop*:

```
while (true){
    sense: ⟨read sensor values⟩;
    think: ⟨calculate action⟩;
    act: ⟨write actuator values⟩;
}
```

Most code generators are constructed such that the code generated from a state machine follows this scheme. For example, for a transition system with alphabet $\mathcal{A}$, states S, transitions T and initial state s_0, the generated code could be as follows.
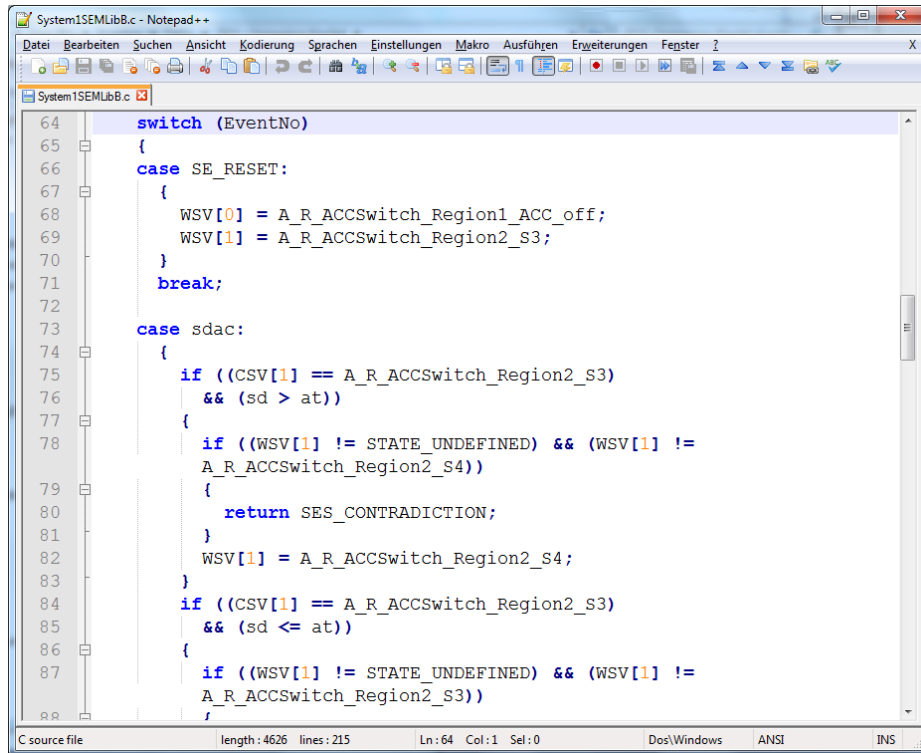
```
state s = s₀;
while (true){
    ⟨get trigger a ∈ 𝒜⟩;
    if ∃s'((s, a, s') ∈ T) then s = s';
    ⟨display state s ∈ S⟩;
}
```

Here, we assume that the transition system is *deterministic*, i.e., for any $s \in S$ and $a \in \mathcal{A}$ there is at most one s' such that $(s, a, s') \in T$. For nondeterministic transition systems, the **if**-statement must be replaced by a nondeterministic **choose**-statement. For execution on a (deterministic) machine, of course, the nondeterminism must be resolved according to some strategy.

For UML state machines, the trigger of a transition could be either an external event or a signal generated as the result of an action. Hence, at any given instant, there may be several events which can be processed by the state machine The semantics of UML determines that the processing of all internal events must be completed before the next external event is considered. Thus, occurrence and processing of an event are treated separately: If an event occurs either from an external source or as the consequence of an internal action, it is put into an *event pool*. From this pool, events which are the result of an internal action are prioritized whenever a new trigger is dispatched. This feature is called ""run-to-completion" semantics: once an action is started, it will run until it is done, before any external signals are processed.

Furthermore, UML state machines may contain parallel and nested regions. Thus, there are some further extensions to the above scheme. For state machines containing parallel regions, there is not one overall "current state" s. Each region has its own "current state". A *configuration* is a tuple which lists for each region the current state of this region. Furthermore, if states and regions are nested, then each transition leaving a superstate exits all regions within that state. Thus, the current state of all sub-regions is affected by a transition from the enclosing state.

Fig. 17. Code generated from the model in Fig. 15

As an example for code generation from UML state machines, we use the industrial tool IAR visualSTATE. This tool is a "front end" for the IAR embedded workbench tool suite, which offers debugging and profiling support for embedded software. However, visualSTATE is able to generate C code for other tools as well. Fig. 17 depicts part of the "readable" code generated from the model in Fig. 15 with this tool. This code can be compiled with standard C compilers, and executed on an embedded target.

6 Conclusion

In these notes, we have discussed the principles of model-based engineering of cyber-physical systems. We have seen the particular challenges which these systems pose to the software development. Then, we considered systems- and requirements-analysis techniques which are the basis of a successful design process. We studied different modeling concepts: systems modeling with SysML, continuous modeling with Scicos, and state transition modeling with UML. Fi-

nally, we have seen how these models can be used for the generation of executable code for embedded controllers.

Of course, these notes cover just a first encounter with this subject. For each of the topics addressed, there is extensive further literature which helps to get a more profound knowledge. Many pointers on textbooks have been given within the various chapters. For current research, the reader is referred to the proceedings of the MBEES (Model-Based Engineering of Embedded Systems, [GHPS14,GHPS15]) and MODELS (Model-Driven Engineering Languages and Systems, [Let15]) conference series.

There are several areas in the development of cyber-physical systems which we have *not* considered. For example, we did not discuss the wide field of quality assurance, i.e., verification and testing of embedded systems. Also here, extensive literature exists (see [ZSM11] for some pointers). Other important aspects include safety and security, fault tolerance, domain- and platform-specific methods, communication and autonomy of systems, and many more. Each of these aspects is continually evolving, and new theories and research directions appear frequently.

However, no theory is of any use without practice: To thoroughly understand and learn the subject, the reader is strongly advised to experiment with tools and platforms which are readily available. Most producers of embedded hardware will give away evaluation boards for free or at low cost. It is a very worthwile exercise to conduct a medium-sized example from the requirements elicitation through the modeling process up to the code generation and the deployment onto the embedded target(s). Doing this will give you a hands-on experience of the problems, but also of the fun which the development of a modern cyber-physical system can bring.

References

[Bos07]   Boston Scientific Inc. PACEMAKER System Specification, 2007. http://sqrl.mcmaster.ca/_SQRLDocuments/PACEMAKER [accessed Oct. 2015].

[Coc01]   Alistair Cockburn. *Writing Effective Use Cases*. Addison-Wesley, 2001.

[Ecl15]   Eclipse Foundation. Papyrus 1.1.0, June 2015. https://eclipse.org/papyrus [accessed Oct. 2015].

[GHPS14]   Holger Giese, Michaela Huhn, Jan Phillips, and Bernhard Schätz, editors. *Dagstuhl-Workshop MBEES: Model based engineering of embedded systems X, Dagstuhl, Germany*. fortiss GmbH, München, 2014. https://www4.in.tum.de/~schaetz/papers/MBEES2014.pdf [accessed Oct. 2015].

[GHPS15]   Holger Giese, Michaela Huhn, Jan Phillips, and Bernhard Schätz, editors. *Dagstuhl-Workshop MBEES: Model based engineering of embedded systems XI, Dagstuhl, Germany*. fortiss GmbH, München, 2015. https://www4.in.tum.de/~schaetz/papers/MBEES2015.pdf [accessed Oct. 2015].

[INC00]   INCOSE (International Council on Systems Engineering). *Systems Engineering Handbook*, volume 2.0. 2000.

[ISO08]   ISO (International Organization for Standardization). *ISO/IEC 15288:2008 — Systems engineering – System life cycle processes*, 2008.

[KHCD13] F. Kordon, J. Hugues, A. Canals, and A. Dohet. *Embedded Systems: Analysis and Modeling with SysML, UML and AADL*. ISTE. Wiley, 2013.

[Let15]   Timothy Lethbridge, editor. *Proceedings of the 18th International Conference on Model Driven Engineering Languages and Systems, Ottawa*. ACM/IEEE, 2015.

[OMG15a] OMG (Object Management Group). SysML 1.4, June 2015. http://www.omgsysml.org [accessed Oct. 2015].

[OMG15b] OMG (Object Management Group). UML 2.5, June 2015. http://www.omg.org/spec/UML [accessed Oct. 2015].

[PBvdL05] Klaus Pohl, Günter Böckle, and Frank van der Linden. *Software product line engineering: foundations, principles, and techniques*. Birkhäuser, 2005.

[PHAB12] Klaus Pohl, Harald Hönninger, Reinhold Achatz, and Manfred Broy. *Model-Based Engineering of Embedded Systems: The SPES 2020 Methodology*. Springer, 2012.

[Poh10]   Klaus Pohl. *Requirements Engineering: Fundamentals, Principles, and Techniques*. Springer, 2010.

[Sch14]   Bernd-Holger Schlingloff. Towards a curriculum for model-based engineering of embedded systems. In Giese et al. [GHPS14]. https://www4.in.tum.de/~schaetz/papers/MBEES2014.pdf [accessed Oct. 2015].

[Som10]   Ian Sommerville. *Software Engineering*. Addison-Wesley, 9 edition, 2010.

[ZSM11]   Justyna Zander, Ina Schieferdecker, and Pieter J. Mosterman, editors. *Model-based testing for embedded systems*. Computational analysis, synthesis, and design of dynamic systems. CRC Press, Boca Raton, 2011.