Chapter 2
Co-modelling and Co-simulation in Embedded Systems Design

John Fitzgerald and Kenneth Pierce

2.1 Introduction

This chapter introduces the first basic concepts of co-modelling and co-simulation, including notions of system, model and co-model, simulation and co-simulation, etc. It also describes the ways in which co-modelling and co-simulation can be integrated with established development processes such as IEEE 15288 (*Systems and Software Engineering—System Life Cycle Processes*, [45]) and IEEE 12207 (*Systems and Software Engineering—Software Life Cycle Processes*, [44]).

The collaborative development of an embedded system requires productive interaction between engineers from very different backgrounds. Control engineering and software engineering have matured over many decades, each with its own philosophy, methods and terminology, and so it is necessary to clarify the common ideas that underpin co-modelling and co-simulation. This chapter introduces these concepts, including the ideas of system (Sect. 2.2), model (Sect. 2.3), co-model (Sect. 2.4), co-simulation (Sect. 2.5) and design space exploration (DSE) (Sect. 2.6). Realising collaborative modelling and co-simulation within established development processes is considered in Sect. 2.7. Finally, Sect. 2.8 provides a summary of the chapter.

2.2 Systems and System Boundaries

We build models in order to assist in the design of *systems*. We regard a *system* as an entity that interacts with other entities, including hardware, software, humans and the physical world [6]. The system may itself be a group of interacting or

J. Fitzgerald (✉) • K. Pierce
Newcastle University, Newcastle upon Tyne, UK
e-mail: john.fitzgerald@newcastle.ac.uk; kenneth.pierce@newcastle.ac.uk

J. Fitzgerald et al. (eds.), *Collaborative Design for Embedded Systems*,
DOI 10.1007/978-3-642-54118-6_2,

interdependent items forming a coherent whole [4]. The *system boundary* defines a frontier between the system and the entities that form its *environment*. The developer can exercise some choice over the design of entities within the boundary of a system of interest. By contrast, the laws governing the behaviour exhibited by the environment are beyond the developer's direct control.

In an embedded system, the entities within the system boundary may be digital computing elements or physical elements such as machines. The environment provides *stimuli* to the system, and the resulting behaviour of the system, visible at its boundary, is termed its *response*. Embedded control systems are typically thought of as being composed of a *controller* and *plant* ("that part of the system which is to be controlled" [43]). The controller contains the control laws and decision logic that affect the plant directly by means of *actuators* and receive feedback via *sensors*.

Experience suggests that, while control engineers and software engineers might broadly agree on these definitions, they will have a natural bias towards some aspects of a system. For example, software engineers may see the environment as everything outside of the computing part of the system, including the plant, whereas control engineers may focus mainly on the plant as the system. Communication is therefore required in a co-modelling project to ensure common understanding of where the boundaries of influence and responsibility lie in the design process.

2.3 Models

In this book, we focus on the use of *models* to describe designs during product development. The act of creating models is called *modelling*. A model is an abstract description of the reality of a putative system [4]. The model is abstract in the sense that it omits details that are not relevant to the *purpose* for which the model is constructed. For example, a model of an aircraft flight control system intended to ensure smooth response to pilot commands may omit details of the cockpit layout, but would instead focus on the commands that can be generated to the control surfaces. Models that are expressed with sufficient clarity and precision may be analysed to confirm or refute the presence of desirable characteristics or the absence of undesirable properties. This helps developers to control risk by providing assurance of design characteristics before expensive commitments are made to implementation in target software and hardware.

A model may contain representations of the system, environment and stimuli. We regard a model as being *competent* for a given analysis if it contains sufficient detail to permit that analysis. Models may be analysed by inspection or by formal mathematical analysis. Many models are also *executable* in that they may be performed as a sequence of instructions on a computer; such an execution is termed a *simulation* because the behaviour exhibited is intended to simulate that of the system of interest.

A *design parameter* is a property of a model that can be used to affect the model's behaviour, but which remains constant during a given simulation. A *variable* is part of a model that may change during a given simulation. We consider *code generation* to be the process of implementing a controller by automatically translating a model into some programming language, which can then be executed on the real computer hardware of the system.

Embedded systems contain both computing and physical elements, and so we expect that the models describing these may be quite different in nature. In this book, we focus on two types of models: discrete-event and continuous-time. In a Discrete-Event (DE) model, "only the points in time at which the state of the system changes are represented" [83, p. 15]. Discrete-event modelling is typically used for digital hardware [5, 68, 93], communication systems [11] and embedded software [22]. In a *continuous-time simulation*, "the state of the system changes continuously through time" and the simulator "approximates continuous change by taking small discrete-time steps." [83, p. 15]. By contrast, Continuous-Time (CT) modelling uses differential equations and iterative numerical integration methods to describe dynamic behaviour. Continuous-time modelling is typically used for analogue circuits and physical processes [68].

In this book, we set out to answer the question of whether DE and CT models can be brought together in a sound but practically useful way to enable the early-stage collaborative design of embedded systems. The principles and experience that we present can be applied to a wide range of notations and tools for CT and DE modelling. However, we have realised the approach using two particular formalisms: *bond graphs* [16] for CT models and Vienna Development Method (VDM) [37, 50] for DE models. VDM models can be constructed, animated and analysed using the tool *Overture* [56], which provides natural features for describing software structures and behaviour. In the same way, bond graphs can be supported by the tool *20-sim* [53] which allows the plant to be modelled in several ways, including the powerful bond graph [52] notation which permits domain-independent description of the dynamic behaviour of physical systems. Overture and 20-sim are linked by a new tool Crescendo, which allows models expressed in the two formalisms to be developed and analysed together. 20-sim and VDM are introduced in depth in Chaps. 3 and 4.

2.4 Co-models

Our approach focuses on system models that are composed of a DE model of a controller and a CT model of a plant (called "co-models"). The DE and CT models are referred to as *constituent models*. Interaction between the DE and CT models is achieved by executing them simultaneously and allowing information to be shared between them. This is termed a *co-simulation*. In a co-simulation, a *shared variable* is a variable that appears in and can be accessed from both the DE and CT

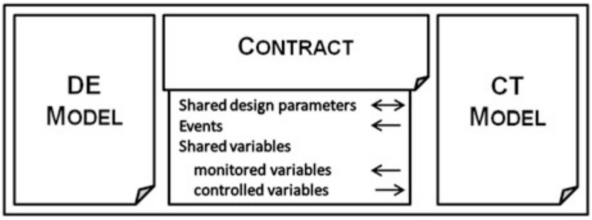

Fig. 2.1 A co-model contains a DE model, contract and CT model, where a contract may define shared design parameters, events and shared variables

models. Design parameters that are common to both models are called *shared design parameters*.

An *event* is an action that is initiated in one model and leads to an action in another model. Events can be scheduled to occur at a specific time (*time events*) or can occur in response to a change in a model (*state events*). State events are described with predicates (Boolean expressions), where the changing of the local value of the predicate during a co-simulation triggers the event. In our approach, events are referred to by name and can be propagated from the CT model to the DE model within a co-model during co-simulation.

Shared variables, shared design parameters and events define the nature of the communication between constituent models. These elements are recorded in a *contract*. For each shared variable, only one constituent model (either the DE model or the CT model) can be assigned write access to it. In the control-system paradigm, shared variables written to by the DE constituent model are called *controlled* variables and those written to by the CT constituent model are called *monitored* variables. A co-model is a model comprising a DE model, a CT model and a contract. Note that a co-model is itself a model and that a co-simulation can therefore be described succinctly as the simulation of a co-model. Figure 2.1 shows a hierarchy of the concepts relating to a co-model.

For a co-model to produce simulation results that can be trusted, the DE and CT models must be *consistent* with each other. Consistency can be broken down into two parts. If the models agree on the identities and data types of the variables, parameters and events they share, then they can be said to be *syntactically consistent* with each other. Achieving syntactic consistency alone does not guarantee that the simulation will produce trustworthy results. For that, the models must also agree on the semantics of the variables, parameters and events they share. If this agreement is reached, then the models can be said to be *semantically consistent*. Only when the DE and CT models are both syntactically and semantically consistent can we say that the co-model is consistent and only then can we place trust in its results.

We suggest that at a minimum, the following should be recorded about each contract entry: the SI unit[1] or a simple description of the value, the range of acceptable values, the datum against which a value is measured, and the direction

[1] The international system of units, abbreviated SI from French: *le Système Internationale d'unité*.

of positive values or frame of reference. For events, the condition under which the event will be raised should be recorded.

2.5 Co-simulation

Simulation of a co-model is called *co-simulation*. During a co-simulation, the DE and CT simulator have responsibility over their own constituent models. Overall coordination and control of the co-simulation is the responsibility of a *co-simulation engine* that is responsible for the progress of time in the co-simulation and the propagation of information between the two constituent models. Crescendo acts as such an engine.

Figure 2.2 shows the co-simulation engine interacting with the DE and CT simulators. The thin arrows indicate inputs and outputs. The DE simulator and CT simulator take a DE model and a CT model as input, respectively. The contract and scenario are inputs to the co-simulation engine. The co-simulation engine outputs a set of results (representing the outcome of the co-simulation). The large arrows indicate data exchange between the co-simulation engine and the two simulators. Note that the simulators do not communicate directly.

2.5.1 The Co-simulation Engine

In order to allow coherent co-simulations to be performed, it is important to reconcile the semantics of two simulation tools from different domains. This is covered in detail in Chap. 13. At this stage however, it is useful to understand the basic operation of a co-simulation and of the co-simulation engine.

Figure 2.3 presents an abstract view of the synchronisation scheme underlying co-simulation between a DE simulation of a controller (top) and a CT simulation of the plant (bottom). The DE and CT simulators are coupled through a co-simulation engine that explicitly synchronises the shared variables, events and simulation time in both linked simulators (the co-simulation engine is not shown explicitly in Fig. 2.3).

Each simulator maintains its own local state and internal simulation time. At the start of a co-simulation step, the two simulators have a common simulation time. The granularity of the synchronisation time step is always determined by the DE simulator. The scheme does not require resource-intensive rollback of the simulation state in either of the simulators, though rollback may occur inside the CT simulator in order to catch the precise time requested, i.e., when a zero crossing is detected in an equation.

At the start of a co-simulation step (t_n in Fig. 2.3), the DE controller simulation sets the controlled variables and proposes a duration by which the CT simulation should, if possible, advance. The co-simulation engine communicates this to the CT

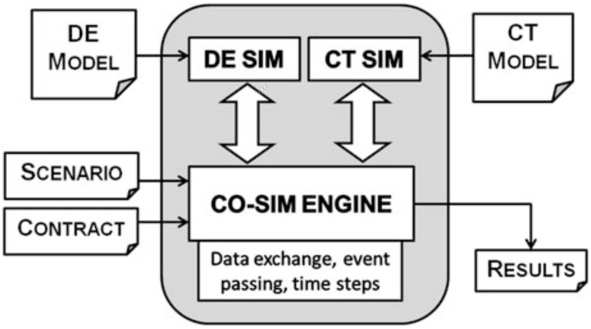

Fig. 2.2 Tool-oriented perspective of a co-model

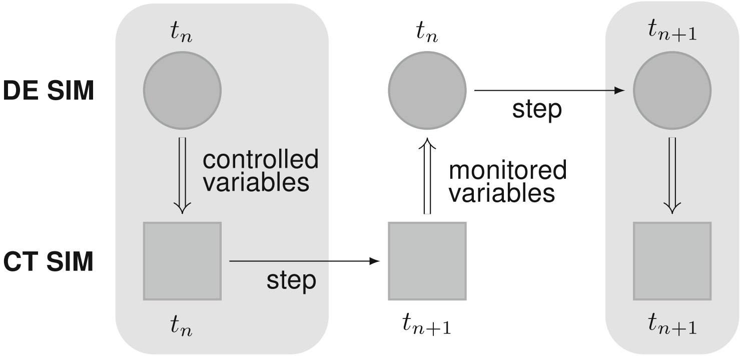

Fig. 2.3 Example of the synchronisation scheme for DE-CT co-simulation

simulator. The CT simulator then tries to advance its simulation time. If an event occurs before the proposed step time is reached, the CT simulator stops early so that the DE simulator can be notified of the event. Once the CT simulator has paused (reaching internal time t_{n+1}), the monitored variables and the actual time reached in the CT simulation are communicated back to the DE simulator. The DE simulation then advances so that both DE and CT are again synchronised at the same simulation time.

2.5.2 Scenarios

To predict a system's behaviour using a co-model, it is often desirable to try out a number of *scenarios*, in which certain aspects are varied, including the setup of the modelled system, simulated user inputs and faulty behaviours. Scenarios are realised through two features: co-simulation *settings* and *scripts*.

The *settings* configure a co-simulation before it begins. Settings include: selection of alternative components from within the co-model, setting of design param-

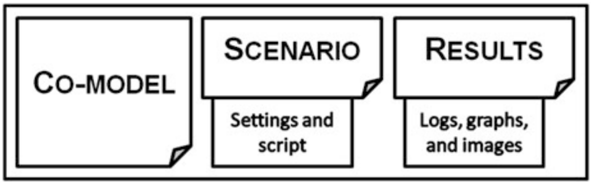

Fig. 2.4 A co-simulation run comprises a co-model, scenario and test results

eters and various tool settings such as co-simulation duration and choice of integration method. A *script* may influence a co-simulation during execution by changing selected values in the co-model. Values can be changed at a given time or in response to a change in the state of the co-model. Scripts are defined using a simple, domain-specific language and are contained in a *script file*. Scripts can be used for fault activation and for mimicking user inputs.

The output from a co-simulation is a *test result* that may take a number of forms, including a *log* of data collected during execution for post-simulation analysis and 2D and 3D plots to allow the simulation state to be observed more immediately. The combination of a co-model, a scenario and corresponding test results is called a *co-simulation run*. Figure 2.4 shows the elements that make up a co-simulation run.

2.6 DSE and Automated Co-model Analysis

As with other model-based techniques, our approach can be used to test a range of solutions while creating a design. We view the *design space* as the set of possible solutions for a given design problem, and *DSE* is an activity undertaken by one or more engineers in which they build and evaluate co-models in order to reach a design from a set of requirements. Where two or more co-models represent different possible solutions to the same problem, these are called *design alternatives*. Each choice involves making a selection from alternatives on the basis of criteria that are important to the developer (e.g. cost, performance). The alternative selected at each point constrains the range of designs that may be viable next steps forward from the current position. Figure 2.5 illustrates the concept of DSE.

Crescendo aids DSE by supporting the selection of a single design from a set of design alternatives. Ranges of values for co-model settings can be defined before the tool then runs co-simulations for each combination of these settings. Results are stored for each simulation and can be analysed. We call this feature *Automated Co-model Analysis (ACA)*. One way to analyse these results is to define a *ranking function*, which assigns a value to each design based upon its ability to meet the requirements defined by an engineer. After the co-simulation runs are complete, the ranking function can be applied to the test results, producing *analysis results* that contain the rank(s) for each design simulated.

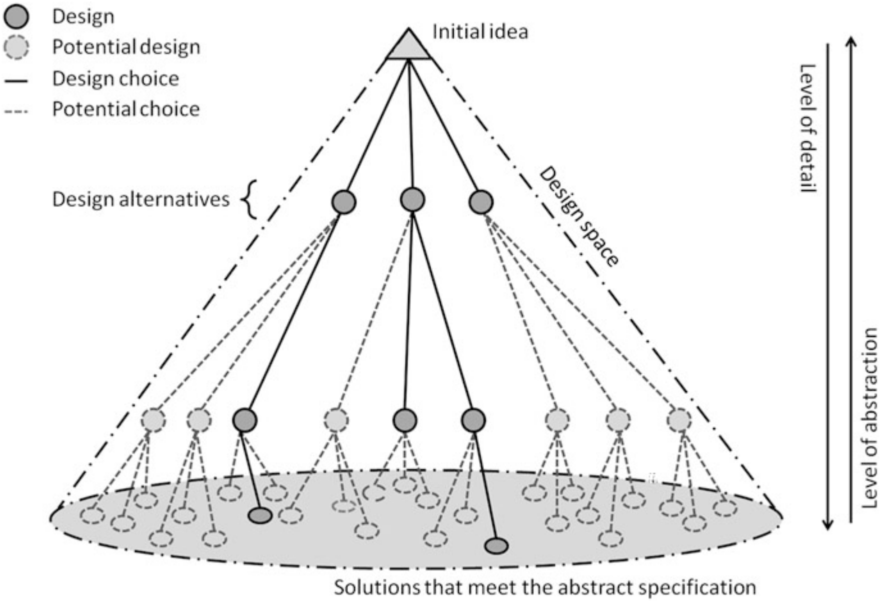

Fig. 2.5 A cone symbolising exploration of the design space and showing how a choice restricts further designs, inspired by [25]

2.7 Co-simulation in Practice

The approach proposed in this book may be applied in the concept definition phase in order to clarify the optimal system-wide requirements for the different parts of the system. However, if it is not just used at the very early stages, successful use of our approach may rest on integrating it with existing practices and design flows. This may involve identifying how our approach fits existing standards. In addition, decisions about how to bring co-simulation into a design flow where model-based design is already used will depend on the type of modelling done previously and the competencies of the team involved. These two aspects also influence the choice of how to build an initial co-model. We offer some insights into these points in the following sections.

2.7.1 Where Does Co-simulation Fit with Existing Practice?

To help with getting a feel for how our approach can fit into existing practices, we describe how our approach can map into the following existing standards: ISO/IEC and IEEE standards 15288 [45] and 12207 [44]; ECSS-E-40 [31] (Space Engineering—Software) and ECSS-Q-80 [32] (Space Product Assurance—

Software Product Assurance); and the Rational Unified Process (RUP) [82]. While these workflows differ in some ways, they have two key properties in common. First, none of them mandates a particular life cycle, but they do identify processes that form part of life cycles that can be implemented in specific projects and development organisations. Second, it is possible to identify a core progression of processes that holds across all of these frameworks.

The development process starts with something that needs to be designed. This is the *operational concept* in IEEE 12207 or the *vision* in the RUP. From here, each of the four workflows defines a set of ordered processes that occur in a development (described as *requirements for engineering* in ECSS-E-40, *technical processes* in IEEE 15288/12207 or *phases* in the RUP). All four workflows broadly adhere to the following pattern:

- Requirements definition
- Requirements analysis
- Architectural design
- Detailed design
- Implementation/integration
- Operations and maintenance

Collaborative modelling and co-simulation can have a role in several of these processes. In IEEE 12207 terms, Crescendo forms an "enabling system" supporting parts of the system life cycle, notably the more upstream technical processes. Relating to ECSS-E-40, Crescendo represents "tools and supporting environment". We would expect to see applications of collaborative modelling and co-simulation as follows:

Requirements definition: During elicitation, requirements can be expressed in terms of a co-model or less formally. Defining the stakeholder requirements includes the development of representative activity sequences or use cases that help to elicit requirements that may not have been explicitly stated. Co-models and co-simulation can help subsequent analysis and maintenance of stakeholder requirements to identify areas of ambiguity or incompleteness and the communication back to the stakeholders of these deficiencies. A collaborative model allows system elements, continuous and discrete, to be expressed in the appropriate formalism, and this in turn may make the model easier to communicate to stakeholders.

Requirements analysis: A representation of a technical system (for example, a co-model) that meets the requirements is built. It involves the definition of a system boundary and of the services delivered at the boundary. Here, we expect co-models to be valuable in considering in depth alternative boundaries and functions. IEEE 15288 states that "System requirements depend heavily on abstract representations of proposed system characteristics and may employ multiple modelling techniques and perspectives to give a sufficiently complete description of the desired system requirements" (IEEE 15288, Clause 6.4.2.3).

Architectural design: This process involves the allocation of responsibilities to units in a solution architecture, each unit having defined internal or external interfaces. From the perspective of co-simulation, the key part of this process is the evaluation of alternative design solutions. Expressed as co-models, these alternatives can form the basis of trade-off and risk analyses.

Detailed design: Here, the design of the units in a solution architecture is built. By this stage, a single design should have been chosen from the set of alternatives. The constituent models of the co-model can then be used to explore the detailed design of the chosen solution and co-simulation used to test the evolving design.

2.7.2 Developer Background and Legacy Models

The choice of how to begin co-modelling can be influenced by the skills of the development team and whether or not legacy models exist. Legacy models are models that already exist and that relate to the system under design. These might include existing models of the system as a whole in a single formalism; models of a part of the system, such as a CT plant model; models of potential components of the system; or models of other systems or components that relate to the system under design. Legacy models, such as existing plant models, might be used directly or could simply be used as a reference. Another potential source of modelling information is in the form of prior art, existing implementations or other prototypes; these can provide valuable measurements or simply inspiration. It is useful to identify these models and sources before modelling begins.

The skill set of the co-model development team is another factor that can influence the way in which our approach is adopted. Perhaps, the "ideal" make-up for a team would be a group of experienced modellers from both the DE and CT domains who understand enough of the mindset within the other domain to communicate and collaborate effectively. Naturally, the real-world environment is unlikely to be so idyllic; therefore, it is a good idea to consider the skills of the team upfront. Software engineers with experience of object-oriented language should not find the move to VDM-RT difficult. Similarly, experience of other CT formalisms such as Matlab should permit a smooth transition to 20-sim for modellers. Note, however, than a team entirely composed of DE or CT experts should be careful not to be overly biased by their backgrounds.

2.7.3 Paths to Co-modelling

Building a first co-model is a big step towards adopting our approach. We define three "standard" paths to reach a first co-model, which are based on the structure of a co-model. Chapter 8 explores the following paths in much greater detail:

DE-first: Here, initial models are produced in the discrete-event formalism before introducing a CT model to form the initial co-model. The focus is on developing the DE controller first.

CT-first: In this approach, initial models are produced in the CT tool, with a DE model being introduced later to form a co-model. The focus is on modelling the dynamics of the plant.

Contract-first: In this third approach, a contract is defined initially. The constituent models are then developed separately but concurrently, following the respective DE-first and CT-first approaches. The contract acts as a guide and target for constituent model development. This allows for early testing of constituent models without reliance on a competent counterpart model. The constituent models are then integrated into a co-model.

2.8 Conclusion

In order to realise the potential of co-modelling and co-simulation technology, we need to take account of established modelling techniques and practices, rather than abandoning trusted approaches. In this chapter, we have outlined the concepts, semantics and pragmatics of co-modelling and co-simulation in our framework. After introducing the basic concepts, we briefly discussed the mechanics of co-simulation between DE and CT simulation engines. We indicated the potential of this approach as a means of exploring alternative designs and described ways in which this can be aligned with existing design flows, with reference to standard development processes including IEEE 15288 and 12207. We have only described the bare bones of the approach; the remaining chapters flesh it out by describing the DE and CT formalisms on which it has been realised, the tool support developed and the practical experience of several substantial industrial applications.

http://www.springer.com/978-3-642-54117-9

Collaborative Design for Embedded Systems
Co-modelling and Co-simulation
Fitzgerald, J.; Larsen, P.G.; Verhoef, M. (Eds.)
2014, XXI, 385 p. 244 illus., 14 illus. in color., Hardcover
ISBN: 978-3-642-54117-9