A Formal Approach to Collaborative Modelling and Co-simulation for Embedded Systems DRAFT

J. S. Fitzgerald, P. G. Larsen, K. G. Pierce and M. H. G. Verhoef
Newcastle University, Aarhus School of Engineering, and CHESS Embedded Technology

August 2011

Abstract

The effective use of model-based formal methods in the development of complex embedded systems requires the integration of discrete-event models of controllers with continuous-time models of their environments. This paper proposes a new approach to the development of such combined models (co-models), in which an initial discrete-event model may include approximations of continuous-time behaviour that can subsequently be replaced by couplings to continuous-time models. An operational semantics of co-simulation allows the discrete and continuous models to run on their respective simulators, managed by a coordinating co-simulation engine. This permits the exploration of the composite co-model's behaviour in a range of operational scenarios. The approach has been realised using the Vienna Development Method (VDM) as the discrete-event formalism, and 20-sim as the continuous-time framework, and has been applied successfully to a case study based on the distributed controller for a personal transporter device.

1 Introduction

Embedded systems are those in which a digital computing element, typically running control software, interacts with devices, such as mechanical or electrical units, and external sources of inputs such as users and the physical environment. The development of such systems is challenging for several reasons. First, the embedded systems market demands rapid innovation and assessment of designs. Second, in critical applications, evidence must be provided to the assurance process that relevant operational scenarios have been considered from an early stage. Third, the development of embedded systems is multidisciplinary in that software engineers must design controllers that implement laws defined by specialists in the application technologies, such as mechanical or electronic engineering. Treating the disciplines separately runs the risk of miscommunication, and inefficiency in handling the many cross-cutting design concerns. Fourth, a major source of complexity, and hence risk, lies in the logic of the controller, and particularly of the supervisory controller that manages higher-level functions such

as mode switching, error detection and recovery[1], an aspect of particular importance in safety-related systems. The move to distributed multiprocessor control adds further urgency to the need to find ways of managing the complexity of controller software design.

Model-based development approaches have the potential to address some of the challenges of embedded systems development. Models produced in the early stages of product development may be analysed in order to identify the strongest design alternatives and provide evidence to the assurance process. Although models can provide a basis for collaboration between engineers, each discipline has its own established abstractions and analysis methods. For example, software is typically modelled using discrete-event formalisms, while mechanical and electrical systems use models based on continuous-time descriptions of phenomena expressed as differential equations. Effective model-based design for embedded control systems should bridge this gap. Our work concerns *co-modelling*, by which we mean the collaborative production of composite models (called *co-models*) that incorporate both a discrete-event (DE) model of a digital controller, and a continuous-time (CT) model of the environment, including the plant to be controlled.

Formal methods, which exploit modelling notations that have mathematically well-founded semantics, offer significant benefits, including improved communication between stakeholders, and the ability to analyse functional and safety properties before the commitment is made to a detailed design. Some of the best-known industrial applications of formal methods use them in a "lightweight" way, targeting them on system elements that carry high development risk because of their criticality or complexity, and using analysis methods that balance cost with the insight gained [35]. We focus on the exploitation of formal semantics to enable the analysis of executable formal models by *simulation*, and we refer to the simulation of co-models as *co-simulation*. Although simulation-based techniques are not as exhaustive as symbolic analysis by model-checking or proof, they do not generally entail the construction of special abstractions to manage the state space, allow a high level of automation, and are familiar to the majority of engineers.

It is possible to construct models that are entirely DE or entirely CT, so long as one is prepared to accept the approximations inherent in a discretized DE model, or the difficulty of describing realistic supervisory control logic in a CT-only formalism. We base our work on the proposition that co-modelling and co-simulation enable more rapid and easier production of accurate models by allowing the integration of CT models of the environment with expressive DE models for complex controllers. The value of co-modelling has been established by related work, and notably the BODERC project which focussed on the use of lightweight, approximate models that were built quickly and generated predictions that were accurate enough to allow a much faster evaluation of design options than could be obtained by building physical prototypes. The approach was demonstrated in a commercial digital copier design, and Océ Technologies reported that it enabled developers to skip at least one complete physical machine-build iteration, saving many person-years of effort [14].

[1] A rule of thumb is that the lower-level controllers that realise the basic control laws typically constitute less than 10% of the entire code base of the system.

The aim of our work is to develop formal methods and tools to support co-modelling and co-simulation. We use the *Vienna Development Method* (VDM) for modelling discrete-event controllers, and 20-sim as the continuous-time framework for modelling the environment. Both are well-established formalisms with stable tool support and a record of industry use. A proof-of-concept of the co-simulation of VDM with 20-sim has been reported, but but did not address co-model construction [11]. The contribution of this paper is to propose and demonstrate a pragmatic, tool-supported approach to the development of co-models, targeted particularly at complex distributed supervisory control. This starts from an exclusively discrete-event model that uses approximations of environment components, gradually replacing these with continuous-time models as greater fidelity is required and as CT models become available. The approach benefits from tool support, and has been applied to the modelling and co-simulation of a personal transportation device with a distributed controller architecture.

A brief introduction to co-modelling and co-simulation in VDM and 20-sim is given in Section 2. Our approach to model construction is outlined in Section 3. The personal transporter case study is described in Section 4. We link our work to related activities in embedded systems and formal methods in Section 5, and make concluding observations and discuss future work in Section 6.

2 Collaborative Modelling and Co-Simulation

This section introduces the technologies on which our work is based. VDM and 20-sim are described in Sections 2.1 and 2.2 respectively. The basic concepts and semantics of co-simulation are described in Section 2.3. Finally, Section 2.4 provides an overview of the tool support developed for co-simulation in our setting.

2.1 The Vienna Development Method

VDM's origins lie in the work on semantics of programming languages at IBM's Vienna Laboratory [2]. The basic modelling language and its denotational semantics have been standardised [25], and a proof theory has been defined, based on the typed logic of partial functions [18, 1]. Extensions to the language have introduced object-oriented structuring and concurrency [10]. A goal of VDM's development since the mid-1990s has been to offer modelling and analysis techniques that are readily accessible without requiring a deep understanding of the underlying semantics. Stress has been laid on robust and efficient tools for simulation rather than proof, and on links to established but less formal frameworks, such as UML. Tool support includes the commercial VDM-Tools, and the open-source tool Overture [12, 22], and there is an ongoing record of successful industry deployment [23, 21].

The application of VDM to embedded control required its extension to allow explicit modelling of computation times on virtual networked processors [34, 24]. However, a weakness in this work is the need to make simplifying assumptions about the environment in order to allow it to be expressed entirely within the discrete-event formalism. A co-simulation approach has been proposed [33], and forms a technical foun-

dation of the DESTECS project that is developing the associated tools and methods[2].

A modeller using VDM may define data by means of constructors for such abstractions as records, sets, sequences and mappings (finite functions) and primitive types such as Booleans, natural numbers and real numbers. Types may be restricted by invariant predicates. Persistent state is modelled by means of typed variables. Functionality may be described in terms of referentially transparent functions, or by operations that have side-effects on state variables. Functions and operations may be specified explicitly, or implicitly by means of logical postconditions over inputs, outputs and state variables. The assumptions on which functions and operations rely are recorded as logical preconditions. In the object-oriented extensions of VDM, a model is organised into class definitions, with each class optionally containing state in the form of instance variables. Association and inheritance relationships may be defined between classes, and the structure of such VDM models can be readily expressed in UML. The full language is described in the standard text [10]. The work described in this paper uses VDM-RT, which is the dialect of VDM that incorporates features necessary to describe asynchronous, object-oriented and distributed real-time systems. These features will be introduced as they arise in this paper.

As an example, consider the VDM-RT model of a controller that manages a water tank that is continuously filled by an input flow, and can be drained by opening a valve. The controller is required to maintain the level of water in the tank between specified low and high marks. When the water reaches the high level mark the valve must be opened, and when it reaches the low level mark, the valve must be closed. A model of one possible controller is shown in Figure 1 (a). This contains a single instance variable that models a valve actuator, and asynchronous operations that model the opening or closing of the valve. Concurrency is modelled by interleaved threads synchronised using predefined predicates. The **sync** section of Figure 1 (a) contains mutual exclusion constraints that prevent multiple threads accessing the interface simultaneously. This type of controller is termed *event-driven* because the events triggered by the level sensors cause the corresponding operations to be performed by the controller (for brevity, we omit the details of the binding of event names to operation calls). Figure 1 (b) shows an alternative *time-triggered* controller, in which an operation modelling the body of a control loop processes input data from a sensor object `level`, leading to the invocation of operations on a `valve` object. The control loop itself is modelled as a periodic thread invoking the operation. The arguments to the **periodic** keyword give the period, jitter, delay and offset (in nanoseconds) — in this model, the `CtrlLoop` operation is invoked every 5 ms exactly. This design requires a sensor that returns the current water level, instead of the two high- and low-level sensors in the event-driven version.

2.2 20-sim

20-sim is a package for modelling and simulating complex physical systems [5, 20, 32]. The approach is multi-domain in that models may describe any of a range of types of physical process, for example electrical, hydraulic or other mechanical systems. The

[2] `http://www.destecs.org/`

```
class Controller

instance variables
  private valve: Actuator

operations
  async public Open: () ==> ()
  Open() == duration(50)
    valve.SetValue(true);

  async public Close: () ==> ()
  Close() == cycles(1000)
    valve.SetValue(false);

sync
  mutex(Open, Close);
  mutex(Open);
  mutex(Close)

end Controller
```

```
class Controller_TT

instance variables
  private valve: Actuator;
  private level: Sensor

operations
  private CtrlLoop: () ==> ()
  CtrlLoop() ==
  ( --obtain sensed level
    dcl l: real := level.GetValue();
    -- calculate response
    if l >= MAX_LEVEL
    then valve.SetValue(true);
    if l <= MIN_LEVEL
    then valve.SetValue(false);
  );

thread periodic(5E6,0,0,0)(CtrlLoop);
end Controller_TT
```

Figure 1: Water tank example: two controllers in VDM (a) event-driven (left), and (b) time-triggered (right).

common underlying formalism uses *bond graphs* [19, 3, 4]. Bond graphs are directed graphs in which the vertices are submodels and the edges, called *bonds*, denote the ideal (or idealised) exchange of energy. Entry points of submodels are called *ports*. The exchange of energy through a port (p) is always described by two implicit variables, effort ($p.e$) and flow ($p.f$). The product of these variables is the amount of energy that passes through the port. The meaning of these two variables depends on the physical domain (examples include voltage and current, and force and velocity).

The 20-sim tool supports the creation and simulation of models that can be represented in a variety of forms, including basic bond graphs; collections of differential equations describing the behaviour of nodes; and iconic diagrams such as those shown in Figure 2. Although the tool is commercial, all the model libraries are open source. The package supports mixed-mode integration techniques to allow the modelling and simulation of computer controlled *physical systems* that contain continuous-time as well as discrete-event elements. The level of complexity of many modern controllers means that discrete-event elements are better modelled using a rich formalism such as VDM. The 20-sim package supports the connection of external software both for model construction and simulation (discrete-event, continuous-time or hybrid), and this connection is exploited in providing support for co-simulation.

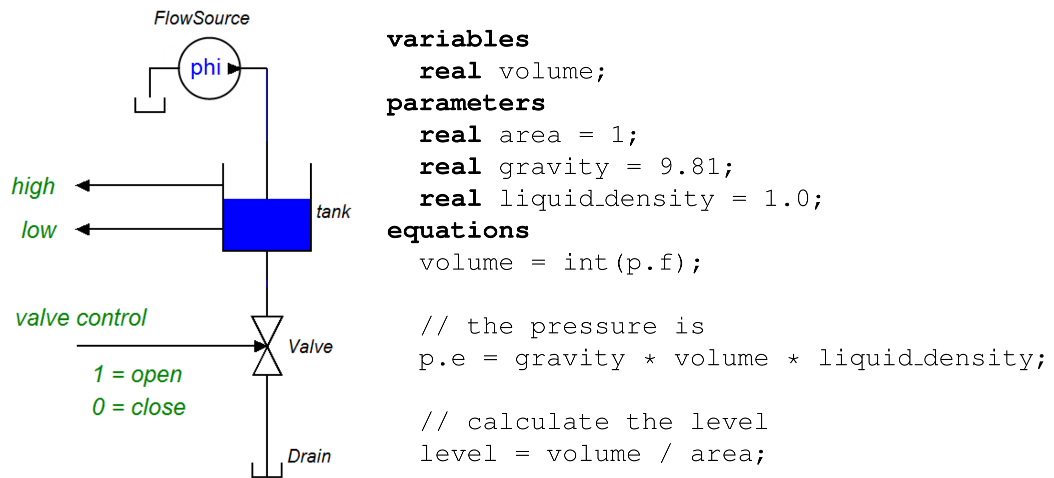


Figure 2: Water tank example: 20-sim model showing iconic diagram representation (left) and tank component equations (right).

2.3 Co-Simulation

In this section, we outline the semantics of co-simulation that underpins our approach. It is first necessary to introduce some terms. A co-model is composed of models of a controller and an environment containing the *plant* (that part of the system which is to be controlled) and environmental stimuli which include disturbances that tend to deflect the plant from desired behaviour. The controller can affect the plant by means of *actuators* and receive feedback via *sensors*. A symbolic execution of a model is called a *simulation*. The collection of timed inputs and stimuli provided to a model during a simulation is called a *scenario*. Note that, in order to perform a simulation of a model, tools may internally manipulate models in order to optimise execution.

Co-simulation involves coupling discrete-event (DE) and continuous-time (CT) simulations that are running in separate tools. Here we adopt Robinson's terminology [29]. In a *discrete-event simulation*, only the points in time at which the state of the system changes are represented. These state changes are *events*. In a *continuous-time simulation*, the state of the system changes continuously through time, and in order to model this behaviour on a computer, the simulator approximates continuous change by taking small discrete-time steps. A *design parameter* is a property of a model that affects the behaviour it describes, but which remains constant during a given simulation. A *variable* is an element of a model that may change during a given simulation.

Interaction between the DE and CT models is achieved by executing them simultaneously and allowing information to be shared between them through *shared variables* that appear in and can be accessed by both the DE and CT 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 that leads to an action in the other model.

Events can be scheduled to occur at a specific time: these are *time events*. Events can also occur in response to a change in a model: these are *state events*. State events are described by means of predicates such that the changing of the local truth value of the predicate during a co-simulation triggers the event. Events are referred to by name and can be propagated from one model to the other within a co-model during co-simulation. Shared variables, shared design parameters, and events are recorded in the *contract*. Only one constituent model (either the DE model or the CT model) may have write access to a shared variable. Shared variables written to by the DE model are called *controlled* variables and those written to the by the CT model are *monitored* variables.

The operational semantics of co-simulation is described in terms of synchronisation of the two simulators [33, 17]. The DE and CT simulators are coupled through a co-simulation engine that explicitly synchronises the shared variables, events and the simulation time in both linked simulators.

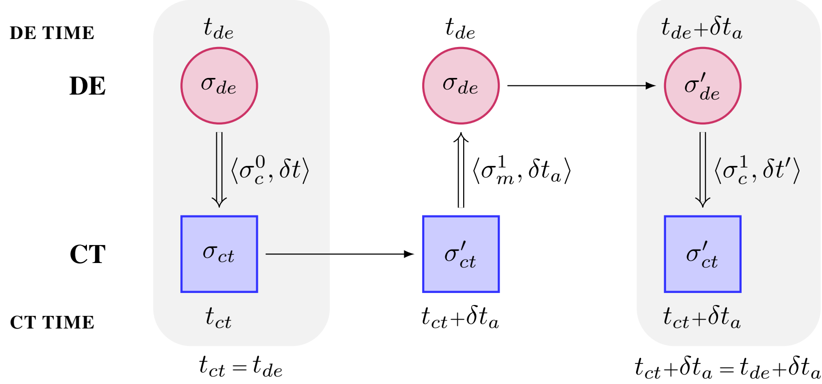


Figure 3: Example of the synchronisation scheme for DE-CT co-simulation

Figure 3 illustrates the synchronisation scheme underlying co-simulation between a DE simulation of a controller (top) and a CT simulation of the environment (bottom). Each simulation maintains its own local state and time at which the state is valid. Thus, let σ_{de} be the internal state of the DE simulation at simulation time t_{de}, and let σ_{ct} be the internal state of the CT simulation at simulation time t_{ct}. The controlled variables defined in the co-simulation contract (whose values are defined in σ_c) are set by the DE controller and read by the environment; the monitored variables (σ_m) are set by the environment model and read by the controller.

Consider a synchronisation cycle which starts with the two simulators having a common simulation time ($t_{ct} = t_{de}$). On each cycle, the DE controller simulation sets the controlled variables, and proposes a duration δt by which the CT simulation should, if possible, advance. As the CT simulation of the environment advances, it may encounter a state in which one of the event predicates defined in the contract becomes true. The state of the monitored variables σ_m^1 and the actual time that it reached δt_a are communicated back to the DE side. If no events occur in the CT simulation during this interval, $\delta t_a = \delta t$. While the CT simulation has been progressing, the DE simulation

remains unchanged, so its local simulation time remains at t_{de} and state σ_{de}. The DE simulation then advances by δt_a so that the both DE and CT are again synchronised at the same simulation time, and the controlled variables are updated (σ_c^1) and the next time step is proposed to CT. The performance of the DE state change takes place in two stages, with the calculations being performed first, separately from advancing the DE simulation time. The granularity of the synchronisation time step is always determined by the DE simulator. The scheme does not require resource-intensive roll-back of the simulation state in either of the simulators.

2.4 Co-Simulation Tool Support

A development environment for constructing VDM/20-sim co-models and running co-simulations is being developed in the DESTECS project. The environment supports the definition of contracts and scenarios for co-simulation. A central co-simulation engine orchestrates the DE (Overture) and the CT (20-sim) simulators, determining how far in time each simulator is allowed to proceed before returning control to the master (δt in Figure 3). It takes the desired arguments, including the scenario the user wishes to simulate, into account and instructs the DE and CT simulations respectively to take the next time step. If an event is raised by the CT simulator before δt has elapsed, control is passed back to the co-simulation engine so that the DE side can react as described above.

The DESTECS tool is constructed on the Eclipse platform[3]. In addition to the standard libraries supplied for VDM, libraries for commonly used feedback controllers are included in the development environment, and may be incorporated into co-models and tuned to the needs of each application.

3 A "Discrete-event First" Approach to Co-model Development

A usable formal modelling framework must support the construction of models, and not merely provide a language for their representation. Our approach to collaborative modelling is intended to promote interaction between disciplines. The process of arriving at a useful co-model is thus one of "bootstrapping" and negotiation, influenced by characteristics of the product such as the development risks of different components.

An ideal approach might be to develop both DE and CT models concurrently and to link them by a contract governing co-simulation. However, this requires the two constituent models to be ready before analysis can proceed. In practice, the development risk in a complex embedded system is not evenly distributed, and it may be advisable to focus early development efforts on the riskier components, whether DE or CT. This is particularly true of the supervisory control aspects of safety-related systems. In these circumstances, we propose a "*DE-first*" approach whereby models are produced initially using the DE formalism for both the controller and environment.

[3]Sources and executables can be found at SourceForge (`https://sourceforge.net/projects/destecs/`).

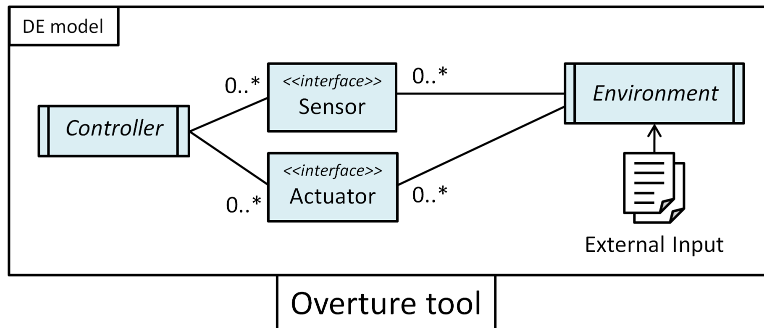
Some components, especially those in the environment, will ultimately be modelled more accurately using the CT formalism, but, initially, abstract and approximate DE models are used instead. Once initial versions of the controller have been produced and tested, CT models of environment components may be substituted for their DE approximations. Technical features needed to support the strategy are considered in Sections 3.1-3.3. We briefly discuss the complimentary "CT-first" approach —used when the performance of low-level loop controllers is initially seen as more important than complex supervisory control— in Section 6.

A "DE-first" development begins with the construction of a single DE model covering both the control system and the environment. This model structure is described in Section 3.1. It allows simulation of the initial DE model in order to gain confidence in the control approach, while permitting progress to be made on supervisory control. A generic style of interface between the two submodels is set up so that they can be readily decoupled when the CT model is introduced (Section 3.2). This structured DE model may be evaluated by executing scenarios, prior to the replacement of the environment submodel by a CT version, creating the co-model (described in Section 3.3) which is then subject to co-simulation.

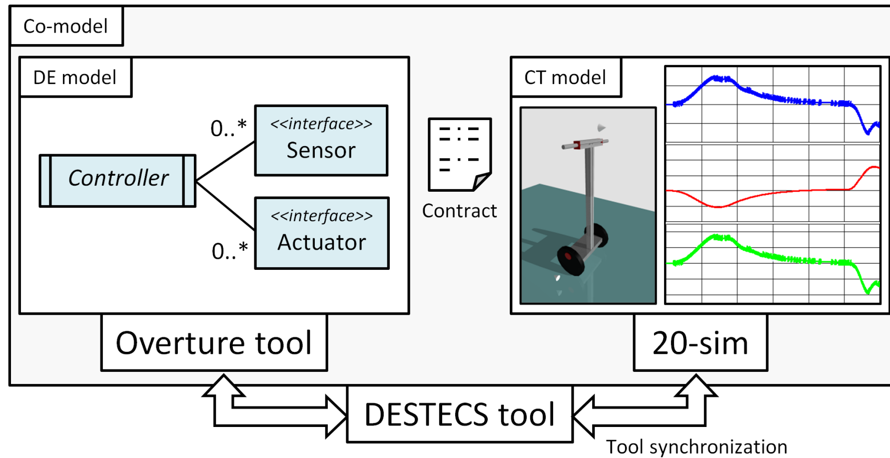
3.1 Initial DE Model Structure

Following a DE-first approach entails developing an initial model in VDM-RT that has elements of both the controller under development and the environment. The environment elements will ultimately be replaced by a CT model but, at this initial stage, the model is exclusively in VDM-RT and the scenario is run as a simulation in Overture. A representation of the structure of this type of model is given as a class diagram in Figure 4(a). It shows an abstract `Controller` class and an abstract `Environment` class that communicate via one or more `Sensor` and `Actuator` objects. The controller and environment are active classes, meaning that they define a thread. The controller and environment run concurrently during a simulation. There are clearly defined interfaces for sensors and actuators in the model. While the details of these interfaces will differ from model to model, we expect that sensor interfaces with a `GetValue` operation and actuator interfaces with a `SetValue` operation could be typical. We recommend programming to interfaces in this way because it allows different sensor and actuator implementations to be produced for the initial DE-only model and, later on, for the co-model. This is useful where a DE-only sensor may need to perform some simplified calculations, whereas a co-model sensor class may simply pass on values calculated by the CT model. Using interfaces in this way ensures that replacing the DE approximation of the CT model with the real CT model can be done with minimal alteration to the controller class. This can be achieved by simply replacing instantiations of the DE-only sensor and actuator classes with instantiations of their co-model counterparts.

Figure 4(b) shows the progression of the DE-first model to a full co-model, consisting of a DE model held in the Overture tool and the CT model held in 20-sim. A contract defines the communication between these models. The DESTECS tool allows handles tool synchronization as described in Sections 2.3–2.4.



(a) DE-only simulation of a controller and interface to a simplified environment model



(b) Co-simulation of a co-model containing the same DE controller and a more realistic CT plant model

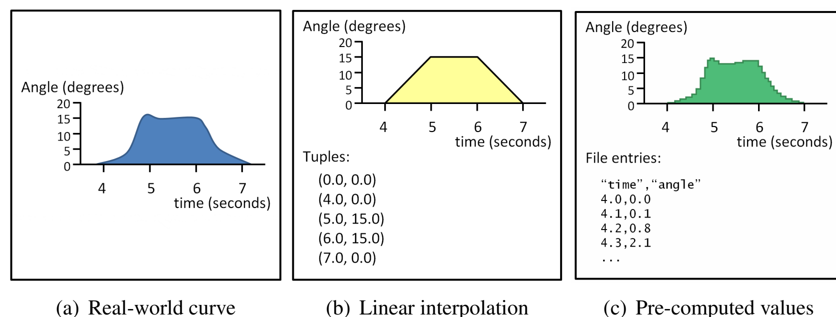
Figure 4: General overview of a DE-first approach

3.2 Approximation of CT Behaviour

In our approach, an environment class is used to mimic the behaviour of the continuous-time world in the discrete-event setting of VDM. In a sense, it implements a basic simulator running in a separate thread in order to provide stimuli to, and observe the responses of, the controller model. So the first approximation made in a DE-first environment model is that the simulation will run in fixed time steps at a much coarser resolution than a full CT model. The key is to find approximations that allow controller logic to be tested in a reasonable time frame. The environment model will be composed of elements that model devices in the physical environment of the controller. It must also handle input from other external sources such as the user or the operating conditions. It is therefore important to consider how to create DE approximations of both kinds of element.

Approximation strategies for defining components of the plant include replacing non-linear relationships with linear approximations and replacing complex differential equations with a mixture of simpler differential equations and constants. For example, in the water tank example shown in Figure 2, the rate of flow of water from the tank depends on the volume of water, therefore as the tank empties, the rate will decrease. A DE-first approximation might use a constant flow rate, where discrete quantities of water disappear from the tank at each time step. This is useful if we are primarily concerned with the correct logic of the controller at the high and low water marks at this stage of development, rather than with the exact time at which these crossings occur. We could then replace this approximate model with a more accurate CT model to better gauge the expected real-world performance of the controller.

Although linear approximation is also useful when defining external input, we may wish to define some erratic behaviours. For example, consider a user rotating a dial clockwise and back again, with the variation in rate and the overshoot that may be expected from a manual operation (Figure 5(a)). One approach to approximating the curve is to identify the key points at which the curve changes and interpolate between these points linearly, representing the curve as a sequence of pairs each giving a time and corresponding angle (Figure 5(b)). This gives a smooth, though approximate, change in angle. If a less linear approximation is required, values can be pre-computed and stored in a file that can be read by the environment simulator to provide the given values at the correct time (Figure 5(c)). The approaches could be combined, allowing the simulator to interpolate linearly between values where the time steps recorded in the file are too coarse. This method would be useful if measured values for components or behaviours were already available. Other sources for this data include using 20-sim itself (or other programs such as MATLAB[4]), which can easily export files containing the values of plotted data. This approach also offers the possibility that during the initial stages of a co-model development, the developer of the CT model could pass some initial test data to the developer of the DE model, even if neither model is sufficiently mature to perform a co-simulation.



(a) Real-world curve (b) Linear interpolation (c) Pre-computed values

Figure 5: Visualization of the change in angle of a dial as it is rotated 15 degrees clockwise and back (a), and two DE approximations (b), (c)

[4] http://www.mathworks.com/products/matlab/

3.3 Co-model Creation

When a DE-first model has reached sufficient maturity, the process of replacing components can begin. If the model structure outlined in Section 3.1 has been used, objects that implement the sensor and actuator interfaces can be substituted without changing the controller. The process of replacing a DE environment approximation with a CT model (and thus creating a co-model) takes place in three steps. First, the CT model of the environment is constructed. Second, entries are added to the co-model contract to describe the variables shared between the DE controller and the CT environment model. This will typically be one or more monitored variables (and/or events) for each sensor object and one or more controlled variables for each actuator object. Third, shared variables are accessed in the DE model by creating new implementations for sensor and actuator objects. Instead of performing calculations, these implementations share their data with the CT model.

When the DE model is incorporated into a co-model, the DE environment and the DE sensor and actuator objects are not instantiated. Instead, the new implementations of these objects are created and the DESTECS tool handles synchronisation of the variables within these objects to their CT counterparts, as described in Sections 2.3–2.4. By using interfaces and alternative implementations of sensor and actuator objects, the change is transparent to the controller. This means that there is a greater chance that the controller will behave correctly when interacting with the CT model, as the scope for introducing errors to the controller when modifying it is reduced. It also means that it is easy to revert to the DE approximation of the environment by simply changing which objects are instantiated and passed to the controller. This switching might be necessary if there are changes to requirements that require large updates to both models, for example.

The degree of maturity required before migrating a DE-first model to a co-model will depend on several factors. It is expected that a reasonable degree of confidence in the correctness of the controller logic should be sought first, as well as a CT model that is at least as mature as the DE approximation. If an entire CT model is ready, then the migration can be done in a single step. However it is also possible that only parts of the CT model may be ready. The modeller can decide at any time what subset of variables can be moved over, so it is possible to have a heterogeneous co-model in which some components use the abstract DE representation, while others use the co-simulation interface. This satisfies the requirements for early design-space modelling, since parts of the model can be detailed where others are specified very abstractly.

4 Case Study: the "ChessWay" Self-balancing Scooter

The DE-first method of co-model development is the result of experience gained in a series of case studies in co-modelling and co-simulation. This section presents one such study, based on the ChessWay self-balancing scooter developed by the company CHESS in the Netherlands as a demonstration and validation platform for model-driven design and distributed control system technology. Co-models of the ChessWay have been developed as a basis for analysing alternative controller architectures, includ-

ing alternative safety controllers. We first outline the ChessWay's main features (Section 4.1) before presenting the DE-first model, including an overview of the discrete approximations used (Section 4.2), and their replacement by a 20-sim CT model (Section 4.3). For brevity, only the most salient features of the VDM models are shown here. The full VDM model of the ChessWay can be downloaded from the Overture website[5].

4.1 The ChessWay

The ChessWay has two wheels, mounted on either side of a platform on which a rider stands, holding on to a handlebar (Figure 6). The system's weight is mostly positioned above the two powerful, direct-drive wheels. As such, it acts like an inverted pendulum and is therefore unstable. In order to stop the ChessWay from falling over and perhaps injuring the rider, it must be actively balanced by driving the wheels. The aim of the system controller is to keep the ChessWay upright, even while stationary. It can do this by applying torque to each wheel independently so that the base of the ChessWay is always kept directly underneath the centre of gravity of the entire system. The rider can move forward (or backward) by leaning forward (or backward). The controller measures the deviation angle of the handlebar and performs an immediate control action in order to keep the ChessWay stable by rapidly moving the wheels to counter the deviation, in much the same way as one might attempt to balance an upright broomstick on one's hand.

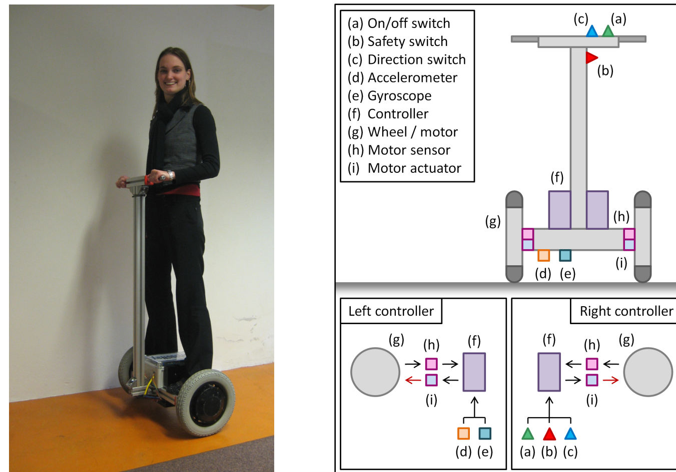


Figure 6: The ChessWay personal transporter

Although its control laws are relatively simple, the ChessWay presents a challenging control problem because the desired nominal system state is in fact metastable. Furthermore, the system dynamics require high frequency control in order to guarantee

[5] `http://www.overturetool.org/`

smooth and robust handling. Safety plays a crucial complicating role: there are circumstances in which the safest thing for the controller to do is to *allow* the ChessWay to fall over. For example, any sudden deviation exceeding 10 degrees from upright could result in the user being subjected to violent control actions. Moreover, it is intuitively clear that even small failures of the hardware or software could easily lead to the ChessWay malfunctioning. It should be possible to specify multi-modal controller behaviour for the ChessWay, which manages safe start-up and shut-down. For example, if the ChessWay is lying on the floor (90 degrees deviation from upright), then the controller needs a dangerously large torque to correct this. This would result in the handlebar swinging suddenly upright, possibly hitting the user. These factors suggest that supervisory control presents at least as significant a risk area as the lower-level control loops, and hence that it is a candidate for a DE-first development.

4.2 ChessWay Discrete-event Model

This section describes the discrete-event model of the ChessWay developed in order to clarify elements of controller behaviour prior to integrating with a CT model of the environment. The purpose of the DE model is to gain confidence in the control algorithm to be used for the ChessWay, including the deployment of system components to processors. Once some basic level of confidence has been obtained, it is possible to integrate a CT model, substituting more accurate models for the DE approximations.

The DE model follows the architecture presented in Section 3, with some minor deviations that will be identified as they arise. The ChessWay has two wheels, each equipped with Hall effect sensors that give each wheel's position. These, plus the user operating the handlebar, are all considered as belonging to the environment of the control system. The structure of the DE environment model is shown in Figure 7.

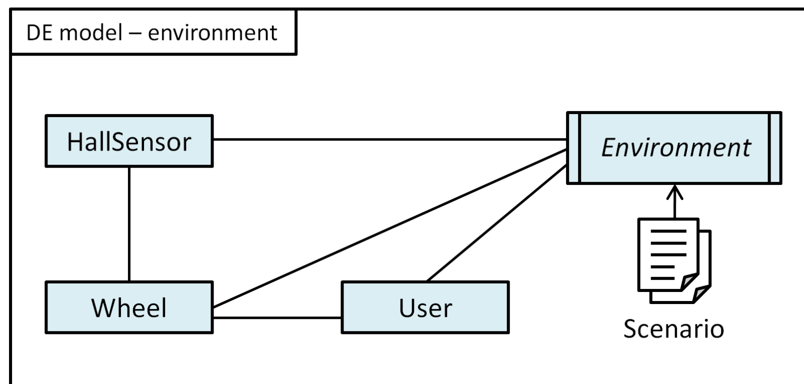


Figure 7: ChessWay VDM-RT class diagram - environment model

The ChessWay has two controllers distributed on two Field Programmable Gate Array (FPGA) devices with a connecting bus. These are called the left (`lctrl`) and right (`rctrl`) controllers because they control the motors for the left and right wheels

respectively. In the design considered here, the left controller also manages an accelerometer and a gyroscope, and the right controller has responsibility for a direction switch (to steer left and right), a safety switch (that is disconnected if the rider falls off the ChessWay), and an on/off switch. The controllers must communicate with each other in the overall control of the ChessWay. If the right controller requires access to the accelerometer and gyroscope, it must do so by communicating over the bus. Similarly, the left controller must access the three switches over the bus. The static topology and physical deployment of functionality to processors is described in the `ChessWay` system class discussed in Section 4.2.3. The overall structure of the interfaces for the controller can be seen in Figure 8. Note that, although all sensor classes are shown as implementing a common sensor interface in Figure 8, finding such a common interface for all sensors and all actuators may not always be practical. For example, a 3D accelerometer yields three real numbers, whereas a switch may yield a Boolean value.

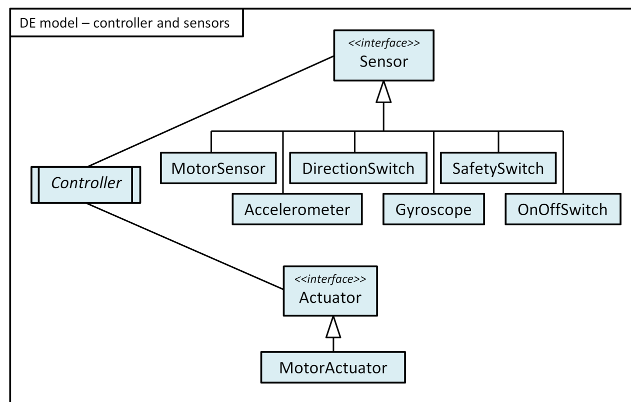


Figure 8: ChessWay VDM-RT class diagram - controller, sensors and actuators

4.2.1 Top-level model — the `World` class

A special `World` class is the top-level entry point of the ChessWay DE model. In this class, instances of the controller and environment are created, and then the `RunScenario` operation instantiates the entire simulation model. It loads a user-defined scenario which specifies the initial settings of all ChessWay devices (such as the safety, direction and on/off switches) that are controlled by external forces (such as the rider) and the evolution of those settings over time, during a simulation run. The environment model is linked to the `ChessWay` system class to facilitate simulation (the link has no counterpart in the final implementation of the system). Simulation commences through the `PowerUp` operations starting the periodic controller threads in the left and right controller objects. The environment simulator thread is started in order to execute the scenario.

```
operations
  public RunScenario: seq1 of char ==> ()
  RunScenario(fname) ==
  (  dcl env: Environment := new Environment(self, MAX_SIM_TIME);
     env.loadScenario(fname);
     ChessWay`lctrl.setEnvironment(env);
     ChessWay`rctrl.setEnvironment(env);
     ChessWay`lctrl.setRightController(ChessWay`rctrl);
     ChessWay`rctrl.setLeftController(ChessWay`lctrl);
     ChessWay`lctrl.PowerUp(); ChessWay`rctrl.PowerUp();
     start(env);
     waitForSimulationEnd()
  );
```

In this model, the coupling between the environment and controller objects is created by the `setEnvironment` operations of the left and right controllers. A more loosely coupled approach may be preferred in general, as shown in Section 3, where the connection is established through the sensor and actuator objects only.

4.2.2 The `Environment` Class

Following the DE-first approach, the `Environment` class contains discrete approximations of the plant and external elements that may ultimately be replaced by CT models. As described in Section 3.2, we develop sensor and actuator interface classes as discrete approximations. The `MotorSensor` interface defines an operation `GetValue` that returns data on the current state of a Hall effect sensor, and assigns its definition as a subclass responsibility. The functionality might be supplied by the class `MotorSensorDE` shown below. In this approximation, a private instance variable models the angular position of the wheel and the operation calculates appropriate sensor outputs depending on the wheel's position. The `Environment` will contain an instance variable for the right-hand motor sensor, and in its constructor create a `MotorSensorDE` linked to the appropriate wheel. Later, this may be substituted by an object that manages the link to the CT simulation via the co-simulation contract (see Section 4.3).

```
class MotorSensorDE is subclass of MotorSensor

instance variables

private mwheel: Wheel

operations

public GetValue: () ==> HallData
GetValue() ==
    let position = mwheel.GetPosition() in
      cases (position div 60):
```

```
        0 -> return mk_HallData(true,  false, true),
        1 -> return mk_HallData(true,  false, false),
        2 -> return mk_HallData(true,  true,  false),
        3 -> return mk_HallData(false, true,  false),
        4 -> return mk_HallData(false, true,  true),
        5 -> return mk_HallData(false, false, true),
        others -> error
      end;

end MotorSensor
```

External inputs include the state of the safety key, direction switch, on/off switch and user input (leaning forward/backward). The evolution over time of these environment variables is described by linear approximations, as described in Section 3.2. Here we describe in outline one approach used in the ChessWay DE model. The environment class defines a set of reserved names, one for each external input, and an instance variable that maps each of these input names to a sequence of readings from points in the approximated curve. For example, the following defines such an approximation mapping for four inputs of interest.

```
values
  reserved: set of seq of char =
   {"RIGHT_SAFETY", "RIGHT_DIRECTION", "RIGHT_ONOFF", "USER"}
types
  public tCtCurve = real * real * real;
  public tCtBehavior = map seq of char to seq1 of tCtCurve
  ...
instance variables
  private mCtBehavior: tCtBehavior := {|->}
```

The tCtCurve elements give a time, a value of the relevant input at that time, and the gradient at that point. A possible behaviour for a scenario to be described as a constant within the environment model. For example, consider a simple scenario in which the ChessWay is enabled after 2 seconds and disabled again at 8 seconds, and the handlebar is moved forward and backward in the period between 4 and 7 seconds. The following mapping defines the scenario in terms of the evolution of the four external inputs mentioned above. This is only one possible approximation structure, and modellers are free to choose more or less elaborate and detailed approximations as they see fit, and depending on the purpose of the simulation.

```
{ "RIGHT_SAFETY" |-> [ mk_(0.0, 1.0, 0.0) ],

  "RIGHT_DIRECTION" |-> [ mk_(0.0, 0.0, 0.0) ],

  "RIGHT_ONOFF" |-> [ mk_(0.0, 0.0, 0.0),
                      mk_(2, 1.0, 0.0),
                      mk_(8, 0.0, 0.0) ],

  "USER" |-> [ mk_(0.0, 0.0, 0.0),
```

```
                mk_(4.0, 0.0, 0.2618),
                mk_(5.0, 0.2618, 0.0),
                mk_(6.0, 0.2618, -0.2618),
                mk_(7.0, 0.0, 0.0) ]
}
```

The operation `mainLoop` implements the core functionality in the `Environment` class, which executes as a periodic thread started by the `RunScenario` operation. On each iteration, it determines the system time and updates the environment model in a manner that reflects the causal relationship between the external inputs and elements of the plant (e.g. the user's state affects the accelerometer and gyroscope). First, it evaluates the external inputs to the sensors by reading the scenario, then it updates the wheel, then the Hall effect sensors, and finally the user's state (including their deviation from upright).

```
operations
  private mainLoop: () ==> ()
  mainLoop() ==
  (  dcl ticks: nat := time,
         clock: real := ticks / World`SIM_RESOLUTION;
     evalSensors(clock);
     mLeftWheel.evaluate(); mRightWheel.evaluate();
     mLeftHall.evaluate(); mRightHall.evaluate();
     mUser.evaluate();
     if (ticks >= mMaxSimTime) then terminate()
  );
...
thread
  periodic (1E6, 0, 0, 0) (mainLoop) -- 1kHz frequency
```

The `mainLoop` operation also checks the current simulation "wall clock" against a preset maximum simulation target time. Once this is reached, the `terminate` operation is called, which will stop simulation and return control to the user.

4.2.3 Constructing the System Model — the `ChessWay` class

The ChessWay controller's distributed architecture, is composed of two CPUs connected by a bus which bus enables communication between the controllers deployed on each processor, for example to exchange information relating to their internal state. The VDM-RT model defines these in a system class that uses built-in CPU and BUS abstractions available in VDM-RT. The two FPGAs are defined as instance variables using the `CPU` constructor with parameters defining the scheduling policy, either fixed priority (FP) or first-come-first-served (FCFS), and processor capacity in terms of instructions per second. The `BUS` constructor's parameters are the type of bus (first-come-first-served is in fact the only kind available in the language at present), its bandwidth in bytes per second, and the identifiers of the set of CPU instances that the bus connects.

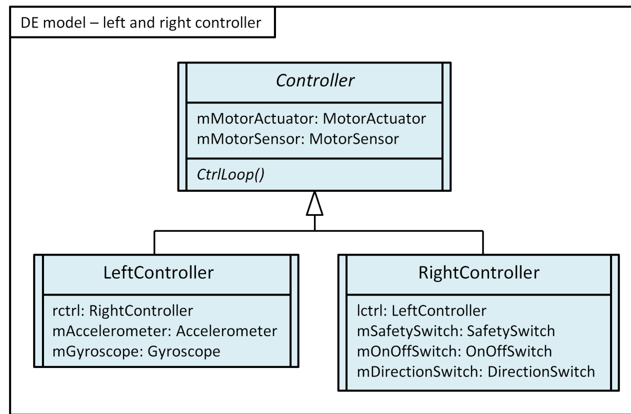


Figure 9: ChessWay controller class diagram

The constructor in the system class deploys the instances of the `LeftController` and the `RightController`, one to each CPU, which are named `LeftCtrl` and `RightCtrl` respectively.

```
system ChessWay

instance variables
  fpga1: CPU := new CPU(<FP>, 10E6);
  fpga2: CPU := new CPU(<FP>, 10E6);

  bus: BUS := new BUS(<FCFS>, 100E3, {fpga1, fpga2});

  static public lctrl: LeftController := new LeftController();
  static public rctrl: RightController := new RightController()

operations
  public ChessWay: () ==> ChessWay
  ChessWay() ==
  (  fpga1.deploy(lctrl, "LeftCtrl");
     fpga2.deploy(rctrl, "RightCtrl")
  );

end ChessWay
```

The `ChessWay` controller class hierarchy is illustrated in Figure 9 and the deployment of instances of these classes is illustrated in Figure 10. Alternative system architectures deploying the functionality to different processors can be explored by changing this part of the model.

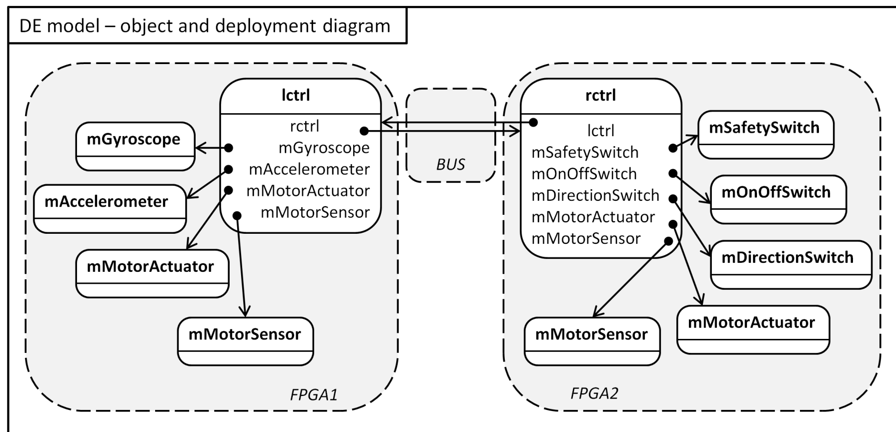


Figure 10: ChessWay controller deployment model (object diagram)

4.2.4 The `Controller` Class

For reasons of brevity, we only outline the controller class here. There are two controllers in the system model, so their common features are defined in a superclass from which the left and right controllers inherit. Each controller is linked to a `MotorActuator` and a `MotorSensor`.

```
class Controller

instance variables
  public mName : seq of char;
  public mMotorActuator : MotorActuator;
  public mMotorSensor : MotorSensor;
```

The right-hand controller in the ChessWay controls the right wheel and monitors the safety, direction and on/off switches. The `RightController` is created by subclassing the generic `Controller` class and overriding the operation prototypes for `CtrlLoop` and `PowerUp`. The `LeftController` class is similar to the `RightController` class with the exception that it manages an accelerometer and a gyroscope instead of the switches controlled by the `RightController`. For reasons of brevity, we omit details here.

4.3 Moving from a DE-only Model to a Co-simulation

A DE-only model of the ChessWay has been developed and run as a simulation in the Overture tool. A variety of scenarios and controller structures can be validated in this way, but the conclusions are limited by the fidelity of the DE approximated inputs

and plant models. As indicated in Section 3.3, these components may be replaced by links to CT models. A contract defines the controlled and monitored variables that link the two running co-models during a co-simulation. The simple contract for the basic sensors and actuator for the right wheel defines the three Hall Effect sensors as the monitored variables, and the PWM (Pulse Width Modulation) setting of the motor actuator (PWM is a common method for controlling the amount of power given to an electric motor) as the controlled variable.

```
contract ChessWay

monitored  real right_hall1;
monitored  real right_hall2;
monitored  real right_hall3;
controlled real right_pwm;

end ChessWay
```

In the co-model, the object of type `MotorSensorDE` that represented the right motor sensor is replaced by an object of class `MotorSensorCT`, whose instance variables are linked to the monitored variables `right_hall1-3` of the CT model in the DESTECS co-simulation tool. The `Environment` is modified so that the `rightMotorSensor` is now a `MotorSensorCT`.

An example of the output generated by co-simulation of the VDM model of the ChessWay controller the 20-sim model of the plant is given in Figure 11, which shows the values of monitored values in the CT model against time during the execution of a scenario. The output can further be visualised by means of a simple animation.

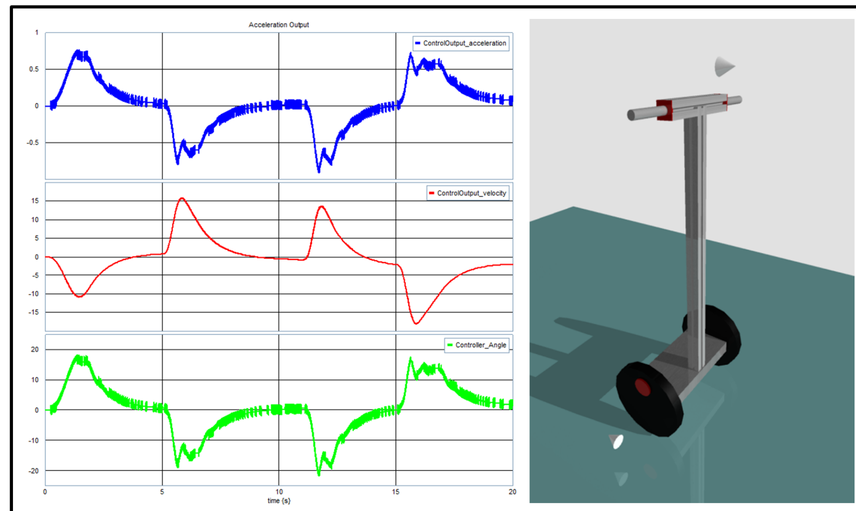


Figure 11: Example co-simulation output produced by 20-sim showing a graph of actuator values over time and a corresponding animation

5 Related Work

There is a substantial body of work on the provision of modelling and simulation for distributed control systems. Matlab/Simulink[6] provides a commercial tool suite for development of control systems, with extensions for multi-domain modelling, and explicit notions of architecture, most notably the tools TrueTime and JitterBug [7]. The semantics of these composed models rely on the underlying tool-specific semantics of Simulink, which forces a fixed time step (time-triggered) operational semantics that significantly influences model structuring and analysis effort. Furthermore, these tools are targeted towards code generation rather than the provision of abstract discrete-event models of supervisory control that motivates our work. Tool suppliers have attempted to bridge that gap by coupling UML software engineering tools such as IBM Rhapsody to Simulink. IBM Rational Rose Real-time software models have been co-simulated with control laws specified in Matlab/Simulink [16], using a platform-neutral notion of time. This is achieved by inserting an interface between the tools, which exposes the software simulator of Rose Real-time to the Simulink internal clock, but the principal limitations of the Simulink semantics still remain valid. Tools such as Scilab/Scicos [6], which have formally defined operational semantics, are candidates to overcome these issues, and are the subject of further study. Similar approaches with time-triggered models of computation have been studied in work coupling the synchronous language Signal to Simulink [31] and in the Giotto modelling approach [15].

Another approach to co-simulation comes from the MODELISAR project[7], which provides a *functional mockup interface (FMI)* for model exchange and tool coupling. The FMI basically constitutes a co-simulation contract, but this approach exposes significant detail from the co-models over this interface, to allow a common co-simulation execution platform. This differs from our approach in which tools are used "as-is" (running models in their own tool-specific and usually highly optimised interpreters) and only time, shared variables and events are exchanged over the co-simulation interface.

Nicolescu et al. propose a software architecture for co-simulation tools [28], for which they provide an operational semantics [13]. Our work is an instantiation of that architecture, with a difference. Their approach is aimed at connecting multiple simulators on a *simulation bus*, whereas we currently connect two simulators point-to-point. In addition, this framework does not address explicit handling of non-normative behaviour, or distribution onto multiple CPUs. However, the suggested framework with our extensions remains tool-independent, as demonstrated by Theelen et al., who showed that the Parallel Object-Oriented Specification Language (POOSL) is able to cope with models of distributed applications using a timed CCS paradigm [30].

Ptolemy-II proposes a component-based, actor-oriented approach in which components are concurrent and interact by message-passing [9], and which supports heterogeneous modelling and design [8]. Significant effort has been invested in formalising the semantics of this framework [26]. The Kepler extension provides first attempts to deal with distributed computing. There are several points of comparison with our approach, notably in the level of abstraction available for modelling the DE part.

[6] `http://www.mathworks.com/`
[7] `http://www.modelisar.com/`

Recently, Myers et al. have described an approach that integrates Behavior Engineering models with a component-based model of a virtual environment in the Modelica language [27]. There are important similarities with our approach, although we focus on the controller/environment partition and use a markedly different formalism on the discrete-event side.

6 Conclusions and Future Work

We have proposed a DE-first approach to the construction of co-models. We have shown how the approach can be supported using the VDM and 20-sim modelling languages, and we have described the first application of the approach to a substantive case study. A key feature of our approach is that it allows modellers to develop DE simulations with "rough" discrete event simulations of elements of the plant and external agents, freeing them to replace these by higher-fidelity CT models when they see fit. This in turn permits a concentration on specific high-risk aspects of the system under development, such as safety-related elements, or high-complexity components. Co-simulation is supported by a tool that manages the coordination of the collaborating simulations that are running within each of the two tools. Our approach differs from much of the existing research in that it exploits the formal semantic bases of the two notations to allow a principled co-simulation without the models having to be translated into a single common tool.

We have adopted and described a DE-first approach to co-model development. In a development that prioritises risks in the CT model, for example where the plant model is particularly complex, a "*CT-first*" approach might be preferred. Here, the core control laws that govern the basic response of the controller can be well-described by 20-sim. While definition of more complex supervisory control is not impossible within the CT model, definition of mode-switching behaviours must be realised through state machines expressed as monolithic conditional statements, which rapidly become hard to manage, particularly when describing error detection and recovery. Therefore in the CT-first approach, once it becomes necessary to model the data and logic of the controller in greater detail, the structuring notions such as functions, classes and objects present in the DE notation of VDM come into their own.

The DESTECS project is running pilot studies to better understand the communications between collaborating engineers "DE-side" and "CT-side", and provide support in the form of patterns and libraries of reusable component models. We also focus on modelling abnormal behaviour (whether caused by conventional faults or "malicious" users), and fault tolerance mechanisms. For example, in the ChessWay, we are modelling the use of safety kernel and safety monitor architectures, examining these alternatives under a common set of scenarios, in particular the potential failure modes resulting from the distributed controllers of the two wheels.

Our work aims to get the best value from currently under-exploited formal methods in selecting candidate designs. Formal verification costs are so high that the first priority is to provide a means of rapidly eliminating non-viable models at an early stage, and this motivates the focus on analysis by co-simulation rather than symbolic verification. Although this paper concerns the methods of co-model construction, our form

of co-simulation is nonetheless a formal method. The formal semantics of the DE and CT models are essential to defining the co-simulation semantics on which the tools have been built. We would argue that these give a more consistent and precise basis for confident decision-making during design, than would be provided by attempting to merge informally-defined models. Much remains to be done to develop the pragmatics further. We intend to develop a more expressive language for scenarios (currently limited to sequences of timed stimuli). The CT-side state is readily visualised by charts and animations but there is no corresponding visualisation of the DE controller state. In addition, techniques are needed to manage the complex set of co-models, scenario inputs and test results that grows as co-models evolve and the design space explored during development.

Co-modelling and co-simulation techniques aim to use formal methods in order to provide facilities that are useful to practising engineers, regardless of their depth of knowledge of the semantic foundations. Ultimately, the utility of this approach will become apparent through industry-based experience. Further case studies, in high-speed mail processing, the control of heavy machinery, and other areas, are in progress. These use a variety of approaches, including DE-first and CT-first model development, and present a diverse set of dependability characteristics. We hope to report on this increasing range of applications in future.

Acknowledgements

We are grateful to Bert Bos, Jan Broenink, Christian Kleijn, Kenneth Lausdahl, Angelika Mader, Yunyun Ni, Augusto Ribeiro, Yoni De Witte, Sune Wolff and Xiaochen Zhang for many useful colleagues. This work is supported by the EU FP7 DESTECS project; Fitzgerald's work is also supported by the UK EPSRC Platform Grant on Trustworthy Ambient Systems.

References

[1] Bicarregui, J., Fitzgerald, J., Lindsay, P., Moore, R., and Ritchie, B. (1994). *Proof in VDM: A Practitioner's Guide*. FACIT. Springer-Verlag. ISBN 3-540-19813-X.

[2] Bjørner, D. and Jones, C., editors (1978). *The Vienna Development Method: The Meta-Language*, volume 61 of *Lecture Notes in Computer Science*. Springer-Verlag.

[3] Breedveld, P. (1985). Multibond-graph elements in physical systems theory. *Journal of the Franklin Institute*, 319(1/2):1–36.

[4] Broenink, J. F. (1990). *Computer-aided physical-systems modeling and simulation: a bond-graph approach*. PhD thesis, Faculty of Electrical Engineering, University of Twente, Enschede, Netherlands.

[5] Broenink, J. F. (1997). Modelling, Simulation and Analysis with 20-Sim. *Journal A Special Issue CACSD*, 38(3):22–25.

[6] Campbell, S. L., Chancelier, J.-P., and Nikoukhah, R. (2006). *Modeling and Simulation in Scilab/Scicos with ScicosLab 4.4*. Springer. ISBN: 978-0-387-27802-5.

[7] Cervin, A., Arzen, K.-E., Henriksson, D., Lluesma, M., Balbastre, P., Ripoll, I., and Crespo, A. (2006). Control loop timing analysis using truetime and jitterbug. In *Computer Aided Control System Design, 2006 IEEE Intl. Conf. on Control Applications, 2006 IEEE Intl. Symp. on Intelligent Control*, pages 1194 – 1199. IEEE.

[8] Davis, J., Galicia, R., Goel, M., Hylands, C., Lee, E., Liu, J., Liu, X., Muliadi, L., Neuendorffer, S., Reekie, J., Smyth, N., Tsay, J., and Xiong, Y. (1999). Ptolemy-II: Heterogeneous concurrent modeling and design in Java. Technical Memorandum UCB/ERL No. M99/40, University of California at Berkeley.

[9] Eker, J., Janneck, J., Lee, E., Liu, J., Liu, X., Ludvig, J., Neuendorffer, S., Sachs, S., and Xiong, Y. (2003). Taming heterogeneity – the Ptolemy approach. *Proc. of the IEEE*, 91(1):127–144.

[10] Fitzgerald, J., Larsen, P. G., Mukherjee, P., Plat, N., and Verhoef, M. (2005). *Validated Designs for Object–oriented Systems*. Springer, New York.

[11] Fitzgerald, J., Larsen, P. G., Pierce, K., Verhoef, M., and Wolff, S. (2010). Collaborative Modelling and Co-simulation in the Development of Dependable Embedded Systems. In Méry, D. and Merz, S., editors, *IFM 2010, Integrated Formal Methods*, volume 6396 of *Lecture Notes in Computer Science*, pages 12–26. Springer-Verlag.

[12] Fitzgerald, J., Larsen, P. G., and Sahara, S. (2008). VDMTools: Advances in Support for Formal Modeling in VDM. *ACM Sigplan Notices*, 43(2):3–11.

[13] Gheorghe, L., Bouchhima, F., Nicolescu, G., and Boucheneb, H. (2006). Formal definitions of simulation interfaces in a continuous/discrete co-simulation tool. In *RSP '06: Proceedings of the Seventeenth IEEE International Workshop on Rapid System Prototyping*, pages 186–192, Washington, DC, USA. IEEE Computer Society.

[14] Heemels, M. and Muller, G. (2007). *Boderc: Model-based design of high-tech systems*. Embedded Systems Institute, Den Dolech 2, Eindhoven, The Netherlands, second edition.

[15] Henzinger, T. A., Horowitz, B., and Kirsch, C. M. (2003). Giotto: A Time-Triggered Language for Embedded Programming. *Proceedings of the IEEE*, 91(1):84–99.

[16] Hooman, J., Mulyar, N., and Posta, L. (2004). Coupling Simulink and UML Models. In Schnieder, B. and Tarnai, G., editors, *Proc. of Symposium FORMS/FORMATS 2004, Formal Methods for Automation and Safety in Railway and Automotive Systems*, pages 304 – 311.

[17] Hooman, J. and Verhoef, M. (2010). Formal semantics of a VDM extension for distributed embedded systems. In Dams, D., Hannemann, U., and Steffen, M., editors, *Concurrency, Compositionality, and Correctness, Essays in Honor of Willem-Paul de Roever*, volume 5930 of *Lecture Notes in Computer Science*, pages 142–161. Springer-Verlag.

[18] Jones, C. B. and Middelburg, K. (1993). A typed logic of partial functions reconstructed classically. Technical Report 89, Department of Philosophy, Utrecht University.

[19] Karnopp, D. C., Margolis, D. L., and Rosenberg, R. C. (2000). *System Dynamics: Modeling and Simulation of Mechatronic Systems*. Wiley-Interscience, third edition.

[20] Kleijn, C. (2006). Modelling and Simulation of Fluid Power Systems with 20-sim. *Intl. Journal of Fluid Power*, 7(3).

[21] Kurita, T. and Nakatsugawa, Y. (2009). The Application of VDM++ to the Development of Firmware for a Smart Card IC Chip. *Intl. Journal of Software and Informatics*, 3(2-3).

[22] Larsen, P. G., Battle, N., Ferreira, M., Fitzgerald, J., Lausdahl, K., and Verhoef, M. (2010). The Overture Initiative – Integrating Tools for VDM. *ACM Software Engineering Notes*, 35(1).

[23] Larsen, P. G. and Fitzgerald, J. (2007). Recent Industrial Applications of VDM in Japan. In Paul Boca, J. B. and Larsen, P. G., editors, *FACS 2007 Christmas Workshop: Formal Methods in Industry*, Electronic Workshops in Computing. British Computer Society.

[24] Larsen, P. G., Fitzgerald, J., and Wolff, S. (2009). Methods for the Development of Distributed Real-Time Systems using VDM. *Intl. Journal of Software and Informatics*, 3(2-3).

[25] Larsen, P. G. and Pawłowski, W. (1995). The Formal Semantics of ISO VDM-SL. *Computer Standards and Interfaces*, 17(5–6):585–602.

[26] Lee, E. A. and Zheng, H. (2007). Leveraging Synchronous Language Principles for Heterogeneous Modeling and Design of Embedded Systems. In *Proceedings of EMSOFT '07*, Salzburg, Austria. ACM.

[27] Myers, T., Dromey, G., and Fritzson, P. (2011). Comodeling: From requirements to an integrated software/hardware model. *IEEE Computer*, 44(4):62 –70.

[28] Nicolescu, G., Bouchhima, F., and Gheorghe, L. (2006). CODIS – A Framework for Continuous/Discrete Systems Co-Simulation. In Cassandras, C. G., Giua, A., Seatzu, C., and Zaytoon, J., editors, *Analysis and Design of Hybrid Systems*, pages 274–275. Elsevier.

[29] Robinson, S. (2004). *Simulation: The Practice of Model Development and Use*. John Wiley & Sons.

[30] Theelen, B., Florescu, O., Geilen, M., Huang, J., van der Putten, P., and Voeten, J. (2007). Software/hardware engineering with the parallel object-oriented specification language. In *Proceedings of the ACM-IEEE International Conference on Formal Methods and Models for Codeesign (MEMOCODE)*, pages 139–148, Los Alamitos (USA). IEEE Computer Society.

[31] Tudoret, S., Nadjm-Tehrani, S., Benveniste, A., and Strömberg, J.-E. (2000). Co-simulation of hybrid systems: Signal-simulink. In Joseph, M., editor, *FTRTRT 2000*, pages 134–151, Berlin, Heidelberg. LNCS, Springer-Verlag.

[32] van Amerongen, J. (2010). *Dynamical Systems for Creative Technology*. Controllab Products, Enschede, Netherlands.

[33] Verhoef, M. (2008). *Modeling and Validating Distributed Embedded Real-Time Control Systems*. PhD thesis, Radboud University Nijmegen.

[34] Verhoef, M., Larsen, P. G., and Hooman, J. (2006). Modeling and Validating Distributed Embedded Real-Time Systems with VDM++. In Misra, J., Nipkow, T., and Sekerinski, E., editors, *FM 2006: Formal Methods*, Lecture Notes in Computer Science 4085, pages 147–162. Springer-Verlag.

[35] Woodcock, J., Larsen, P. G., Bicarregui, J., and Fitzgerald, J. (2009). Formal Methods: Practice and Experience. *ACM Computing Surveys*, 41(4):1–36.