Collaborative Modelling and Co-simulation in the Development of Dependable Embedded Systems

John Fitzgerald, Peter Gorm Larsen, Ken Pierce, Marcel Verhoef, Sune Wolff

Collaborative Modelling and Co-simulation in the Development of Dependable Embedded Systems

John Fitzgerald [1], Peter Gorm Larsen [2], Ken Pierce [1],
Marcel Verhoef [3], and Sune Wolff [2,4]

[1] Newcastle University, UK, {John.Fitzgerald,K.G.Pierce}@ncl.ac.uk
[2] Aarhus School of Engineering, Denmark, {pgl,swo}@iha.dk
[3] Chess, Haarlem, The Netherlands, Marcel.Verhoef@chess.nl
[4] Terma A/S, Denmark, sw@terma.dk

Abstract. This paper presents initial results of research aimed at developing methods and tools for multidisciplinary collaborative development of dependable embedded systems. We focus on the construction and analysis by co-simulation of formal models that combine discrete-event specifications of computer-based controllers with continuous-time models of the environment with which they interact. Basic concepts of collaborative modelling and co-simulation are presented. A pragmatic realisation using the VDM and Bond Graph formalisms is described and illustrated by means of an example, which includes the modelling of both normal and faulty behaviour. Consideration of a larger-scale example from the personal transportation domain suggests the forms of support needed to explore the design space of collaborative models. Based on experience so far, challenges for future research in this area are identified.

1 Introduction

Whether viewed from a technical or a commercial perspective, the development of embedded systems is a demanding discipline. Technical challenges arise from the need to develop complex, software-rich products that take the constraints of the physical world into account. Commercial pressures include the need to innovate rapidly in a highly competitive market and to offer products that are simultaneously resilient to faults and highly efficient.

Traditional development approaches are mono-disciplinary in style, in that separate mechanical, electronic and software engineering groups handle distinct aspects of product development and often do so in sequence. Contemporary concurrent engineering strategies aim to improve the time to market by performing these activities in parallel. However, cross-cutting system-level requirements that cannot be assigned to a single discipline, such as performance and dependability, can cause great problems, because their impact on each discipline is exposed late in the development process, usually during integration. Embedded systems, in which the viability of the product depends on the close coupling between the physical and computing disciplines, therefore calls for a more multidisciplinary approach.

High-tech mechatronic systems are complex (a high-volume printer typically consists of tens of thousands of components and millions of lines of code) and so is the

associated design process. There are many design alternatives to consider but the impact of each design decision is difficult to assess. This makes the early design process error-prone and vulnerable to failure as downstream implementation choices may be based on it, causing a cascade of potential problems. Verhoef identifies four causes of this problem [24]. First, the disciplines involved have distinct methods, languages and tools. The need to mediate and translate between them can hamper development by introducing opportunities for imprecision and misunderstanding. The inconsistencies that result are difficult to detect because there is usually no structured process for analysing system-level properties. Second, many design choices are made implicitly, based on previous experience, intuition or assumptions, and their rationale is not recorded. This can lead to locally optimal but globally sub-optimal designs. Third, dynamic aspects of a system are complex to grasp and there are few methods and tools available to support reasoning about time varying aspects in design, in contrast to static or steady-state aspects. Fourth, embedded systems are often applied in areas where dependability is crucial, but design complexity is compounded by the need to take account of faults in equipment or deviation by the environment (plant or user) from expected norms. Non-functional properties associated with dependability can be hard to capture and assess. This typically leads to designs that are over-dimensioned, making implementation impractical due to the associated high costs.

A strong engineering methodology for embedded systems will be *collaborative*[5]. It will provide notations that expose the impact of design choices early, allow modelling and analysis of dynamic aspects and support systematic analysis of faulty as well as normal behaviour. The hypothesis underpinning our work is that lightweight formal and domain-specific models that capture system-level behaviour can supply many of these characteristics, provided they can be combined in a suitable way and be evaluated rapidly. Formal techniques of system modelling and analysis allow the precise modelling of desired behaviour upstream of expensive commitments to hardware and code. The support for abstraction in formal modelling languages allows the staged introduction of additional sources of complex behaviour, such as fault modelling. A recent review of the use of formal methods [27] suggests that successful industry applications often make use of tools that offer analysis with a high degree of automation, are lightweight in that they are targeted at particular system aspects, are robust and are readily integrated with existing development practices.

We conjecture that a collaborative methodology based on lightweight formal modelling improves the chances of closing the design loop early, encouraging dialogue between disciplines and reducing errors, saving cost and time. Throughout the paper, we term this approach "collaborative modelling" or "co-modelling". In previous work [24], Verhoef has demonstrated that significant gains are feasible by combining VDM and Bond Graphs, using co-simulation as the means of model assessment. Andrews et al. have suggested that collaborative models are suited to exploring fault behaviours [1]. In this paper, we build upon these results, indicating how a collaborative modelling

[5] Collaboration is "United labour, co-operation; esp. in literary, artistic, or scientific work." [23]. Simple coordination between disciplines, for example, bringing mechatronic and software engineers together in the same space, or enabling communication between them, is necessary but not sufficient to achieve collaboration.

approach can be realised in existing formally-based technology, how models can be extended to describe forms of faulty behaviour, and identifying requirements for design space exploration in this context.

In Section 2 we introduce concepts underpinning co-modelling and co-simulation. Section 3 shows how we have so far sought to realise these ideas using the VDM discrete event formalism and the Bond Graph continuous-time modelling framework. We outline some of the design decisions that face collaborative teams in the area of fault modelling in particular. Section 4 looks towards the application of co-simulation in exploring the design space for larger products. Section 5 identifies research challenges that spring from our experience so far.

2 Collaborative Modelling and Design Space Exploration

In our approach to collaborative development, a *model* is a more or less abstract representation of a system or component of interest. We regard a model as being *competent* for a given analysis if it contains sufficient detail to permit that analysis. We are primarily concerned with analysis by execution, so we will generally be interested in formal models that, while they are abstract, are also directly executable. A test run of a model is called a *simulation*. A *design parameter* is a property of a model that affects its behaviour, but which remains constant during a given simulation. A simulation will normally be under the control of a *script* that determines the initial values of modelled state variables and the order in which subsequent events occur. A script may force the selection of alternatives where the model is underspecified and may supply external inputs (e.g. a change of set point) where required. A *test result* is the outcome of a simulation over a model.

A goal of our work is to support the modelling of faults and resilience mechanisms. Adopting the terminology of Avizienis et al. [2], we regard a *fault* as the cause of an *error* which is part of the system state that may lead to a *failure* in which a system's delivered service deviates from specification. *Fault modelling* is the act of extending the model to encompass faulty behaviours. *Fault injection* is the act of triggering faulty behaviour during simulation and is the responsibility of a script.

A *co-model* (Figure 1 (a)) is a model composed of:

- Two component models, normally one describing a computing subsystem and one describing the plant or environment with which it interacts. The former model is typically expressed in a discrete event (DE) formalism and the latter using a continuous-time (CT) formalism.
- A *contract*, which identifies shared design parameters, shared variables, and common events used to effect communication between the subsystems represented by the models.

A co-model is itself a model and may be simulated under the control of a script. The simulation of a co-model is termed *co-simulation*. A co-model offers an interface that can be used to set design parameters and to run scripts to set initial values, trigger faulty behaviour, provide external inputs and observe selected values as the simulation progresses. Our goal is to provide modelling and simulation techniques that support

design space exploration, by which we mean the (iterative) process of constructing co-models, co-simulation and interpretation of test results governing the selection of alternative models and co-models as the basis for further design steps.

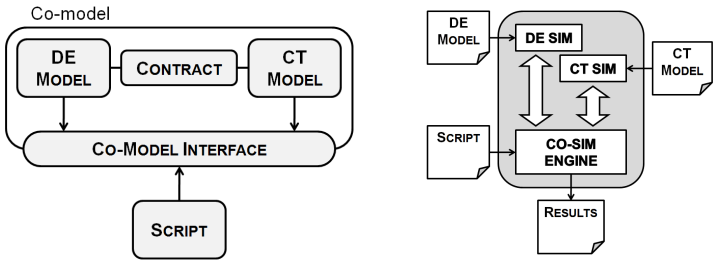


Fig. 1. (a) conceptual view of a co-model (left) and (b) execution of a co-model realised using a co-simulation engine (right).

In a co-simulation, a *shared variable* is a variable that appears in and can be accessed from both component models. Predicates over the variables in the component models may be stated and may change value as the co-simulation progresses. The changing of the logical value of a predicate at a certain time is termed an *event*. Events are referred to by name and can be propagated from one component model to another within a co-model during co-simulation. The semantics of a co-simulation is defined in terms of the evolution of these shared variable changes and event occurrences while co-model time is passing. In a co-simulation, the CT and DE models execute as interleaved threads of control in their respective simulators under the supervision of a *co-simulation engine* (Figure 1 (b)). The DE simulator calculates the smallest time Δt it can run before it can perform the next possible action. This time step is used by the co-simulation engine in the communication to the CT simulator which then runs the solver forward by up to Δt. If the CT simulator observes an event, for example when a continuously varying value passes a threshold, this is communicated back to the DE simulator by the co-simulation engine. If this event occurred prior to Δt, then the DE simulator does not complete the full time step, but it runs forward to this shorter time step and then re-evaluates its simulator state. Note that it is not possible (in general) to roll the DE simulation back, owing to the expense of saving the full state history, whereas the CT solver can work to specified times analytically. Verhoef et al. [25] provide an integrated operational semantics for the co-simulation of DE models with CT models. Co-simulation soundness is ensured by enforcing strict monotonically increasing model time and a transaction mechanism that manages time triggered modification of shared variables.

3 Co-modelling and Co-simulation in 20-Sim and VDM

The work reported in this paper is aimed at demonstrating the feasibility of multidisciplinary collaborative modelling for early-stage design space exploration. As a proof

of concept, methods and an open tools platform are being developed to support modelling and co-simulation, with explicit modelling of faults and fault-tolerance mechanisms from the outset. This activity is undertaken as part of the EU FP7 Project DESTECS [4] [6].

The proof of concept work uses continuous-time models expressed as differential equations in Bond Graphs [17] and discrete event models expressed using the Vienna Development Method (VDM) [16,10] notation. The simulation engines supporting the two notations are, respectively, 20-sim [6] [7] and Overture [18] [8]. Complementary work investigates the extension of the co-simulation approach to other modelling languages [26]. An open, extensible tools platform will be developed, populated with plug-ins to support static analysis, co-simulation, testing and fault analysis. Trials will be conducted on industrial case studies from several domains, including document handling, heavy equipment and personal transportation. Aspects of the latter study are introduced in Section 4. In this section we first introduce the two modelling notations and their tools (Sections 3.1–3.2) before discussing a simple example of co-simulation (Section 3.3) and fault modelling (Section 3.4).

3.1 VDM

VDM is a model-oriented formal method that permits the description of functionality at a high level of abstraction. The base modelling language, VDM-SL, has been standardised by ISO [15]. Extensions have been defined for object-orientation (VDM++ [11]) and real-time embedded and distributed systems (VDM-RT [19]). VDM-RT includes primitives for modelling deployment to a distributed hardware architecture and support for asynchronous communication.

VDM is supported by industrial strength tools: VDMTools [8,12] and the open source Overture tool [18] (being developed with the Eclipse Platform). Both tools have been extended with the capability to generate logfiles derived from execution of VDM-RT models [9,19].

3.2 20-sim

20-sim [6], formerly CAMAS [5], is a tool for modelling and simulation of dynamic systems including electronics, mechanical and hydraulic systems. All models are based on Bond Graphs [17] which is a non-causal technology, where the underlying equations are specified as equalities. Hence variables do not initially need to be specified as inputs or outputs. In addition, the interface between Bond Graph elements is port-based where each port has two variables that are computed in opposite directions, for example voltage and current in the electrical domain. 20-sim also supports graphical representation of the mathematical relations between signals in the form of block diagrams and iconic diagrams (building blocks of physical systems like masses and springs) as more user friendly notations. Combination of notations is also possible, since Bond Graphs

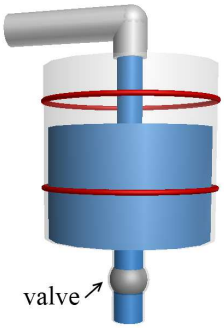
[6] http://www.destecs.org/

[7] http://www.20sim.com/

[8] http://www.overturetool.org/

provide a common basis. It is possible to create sub-models of multiple components or even multiple sub-models allowing for a hierarchical model structure.

3.3 Basic Co-simulation in 20-sim and VDM



$$\frac{dV}{dt} = \varphi_{in} - \varphi_{out} \quad (1)$$

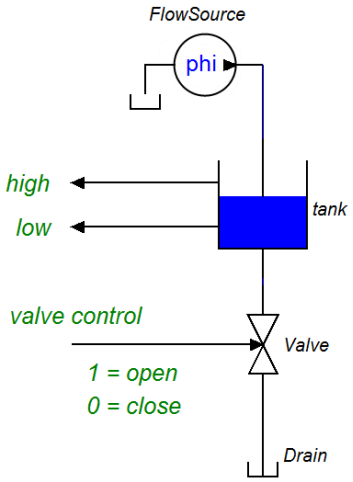
$$\varphi_{out} = \begin{cases} \frac{\rho * g}{A * R} * V & \text{if valve open} \\ 0 & \text{if valve closed} \end{cases} \quad (2)$$

Fig. 2. Water tank level controller case study system overview

In this section, co-simulation between a VDM and 20-sim model is illustrated by means of a simple example based on the level controller of a water tank (Figure 2). The tank is continuously filled by the input flow φ_{in}, and can be drained by opening the valve, resulting in the output flow φ_{out}. The output flow through the valve when this is opened or closed is described by Equation 2 in Figure 2, where ρ is the density of the water, g is acceleration due to gravity, A is the surface area of the water tank, R is the resistance in the valve and V is the volume. An iconic diagram model of this system created in 20-sim is shown in Figure 3 (a). There are two simple requirements for the discrete-event controller: when the water reaches the "high" level mark the valve must be opened, and when the water reaches the "low" level mark, the valve must be closed. A VDM model of the controller is in Figure 3 (b).

The controller model is expressed in VDM-RT. An instance variable represents the state of the valve and the asynchronous `Open` and `Close` operations set its value. Both operations are specified explicitly in the sense that they are directly executable. In order to illustrate the recording of timing constraints in VDM-RT, the **`duration`** and **`cycles`** statements constrain the time taken by the operations to 50 ms in the case of `Open` and 1000 processor cycles in the case of `Close`. The time taken for a `Close` operation is therefore dependent on the defined speed of the computation unit (CPU) on which it is deployed (described elsewhere in the model). The synchronisation constraints state that the two operations are mutually exclusive.

A co-model can be constructed consisting of the 20-sim model and VDM model shown above. The co-simulation contract between them identifies the events from the CT model that are coupled to the operations in the DE model and indicates that `valve` is shared between the two models. The contract indicates which state event triggers




```
class Controller

instance variables
 private i : Interface

operations
 async public Open:() ==> ()
 Open() == duration(50)
   i.SetValve(true);

 async public Close:() ==> ()
 Close() == cycles(1000)
   i.SetValve(false);

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

end Controller
```

Fig. 3. (a) 20-Sim model (left) and (b) event-driven controller in VDM (right).

which operations. In the case the water level rises above the upper sensor, the `Open` operation shall be triggered and respectively when the water level drops below the lower sensor, the `Close` operation shall be called. Note that `valve` represents the actual state of the valve, not merely the controller's view of it. These facets are explored in Section 3.4 below.

3.4 Modelling of Faults

The water tank case study has so far considered only the nominal behaviour of the environment and controller. This section considers the types of faults that one might wish to explore using co-simulation and how they might be modelled, taking the water tank as an example. The choice of which faults to model depends on the purpose of the overall modelling process. For example, where the purpose is to identify mechanisms for recovering from or tolerating faults that lead to significant system-level failures (those with unacceptable likelihood or severity), an analytic technique such as Fault Tree Analysis might be used to identify particular faults that lead to significant system failures. However faults of interest are identified, the model must be made competent to express them and to describe any recovery or tolerance measure to be put in place.

Sensor and actuator faults are particularly interesting because they typically cross the boundary between components of the co-model. In the water tank example, the sensors (high and low) and the actuator (valve) are not modelled as distinct units. Their connection is realised purely through the co-simulation framework and is essentially

perfect — the controller is always notified of events and the valve always reacts correctly to the controller.

The designer faces a choice of alternative ways in which to represent sensor and actuator faults. One approach is to specify failures of communication in the co-simulation contract. This is somewhat disconnected from reality however, since a contract will not necessarily map directly to the physical sensors and actuators of the real system. In our current work, we advocate keeping the contract pure and instead introducing explicit models of sensors and actuators to either the controller or plant model (or both where necessary). These models can then exhibit faulty behaviour as required. Since the controller and plant are modelled in far richer languages than the contract, this approach provides scope for describing complex faults. In addition, the capability to exhibit faults then exists statically in the controller and/or plant model and not simply dynamically as part of the co-simulation.

Bearing this in mind, we can introduce sensors and actuators into the model shown above. Let us first consider a faulty valve that can become *stuck*. That is, if the valve is open it will not close when commanded and if it is closed it will not open. We make the co-model competent to exhibit this fault by introducing a `ValveActuator` class into the VDM controller model, representing the actuator (Figure 4).

First, note the instance variable `stuck`. This represents an internal error state: when `stuck` is false, the valve actuator will behave correctly. An operation called `SetStuckState` is defined to control when the valve actuator becomes stuck. In this model, no logic is included to control exactly when faults occur. A more complex fault model could include parameters to tune when faults occurred, for example, based on stochastic measures. Alternatively, the `SetStuckState` operation could be exposed to the co-simulation tool, which could then activate the fault during a co-simulation, in response to some scripted behaviour. Both methods have benefits and drawbacks, so it is suggested that the DESTECS approach will allow both, leaving it up to the user to decide.

Second, consider the main operation of the class called `Command`. This operation should open and close the valve (using `i.SetValve`), depending on the command given. Note that in Figure 3 (b), the controller was directly responsible for operating the valve. In this new model however, the controller must call the `Command` operation. The body of `Command` gives an explicit definition for the operation, which is reasonably intuitive — if the valve is not stuck, the valve will open if the command given was `<OPEN>` and close if the command was `<CLOSE>`. If the valve is stuck, nothing will happen.

In addition to the explicit definition, a precondition, postcondition and errors clause are also given. The precondition records assumptions about the state and input parameters when an operation is invoked. In this case, the precondition states that the operation will only behave correctly if the valve is not stuck. The postcondition captures the behaviour of the operation as a relation between the initial and final state. Postconditions must hold if the precondition holds. It is obvious however that the precondition may not be met, i.e. when the valve is stuck. In this case, it is not necessary for the operation to meet the postcondition and its behaviour is typically undefined. We can however introduce an errors clause to capture the behaviour when the valve is stuck. Here, the **`errs`**

```
class ValveActuator

types
  ValveCommand = <OPEN> | <CLOSE>;

instance variables
 private i : Interface;
 private stuck : bool := false

operations
 public Command: ValveCommand ==> ()
 Command(c) == duration(50)
   if not stuck then
    cases c:
     <OPEN>  -> i.SetValve(true),
     <CLOSE> -> i.SetValve(false)
    end
 pre not stuck
 post i.ReadValve() <=> c = <OPEN> and
      not i.ReadValve() <=> c = <CLOSE>
 errs STUCK : stuck -> i.ReadValve() = ~i.ReadValve();

 private SetStuckState: bool ==> ()
 SetStuckState(b) == stuck := b
 post stuck <=> b and not stuck <=> not b;

end ValveActuator
```

Fig. 4. Explicit model of a valve in VDM, which can exhibit a stuck fault.

clause records that the valve's state will remain unchanged (note ˜-prefixed instance variable names indicate the initial value of the variable).

Another valve fault that might be considered is a leak. A simple model of a leaky valve would be one in which a constant amount of water flows out of the tank, even when the valve is closed. It is much more natural to model this fault in 20-sim, since it involves altering the flow rate (part of the plant model). Reproducing this on the VDM side would (at least) involve modifying the co-simulation contract to allow direct modification of the flow rate, which is an inelegant solution. Thus a modification of the 20-sim plant model is required. A diagram of the modified model is given in Figure 5 (based on Figure 3).

The leak is modelled as an alternative route for water to flow from the tank to the drain, bypassing the valve. The rate of flow is a constant (K). As with the VDM model, the fault can be active or dormant (zero flow). In 20-sim however, activation of the fault is modelled as an input signal to the leaky component, in much the same way that the

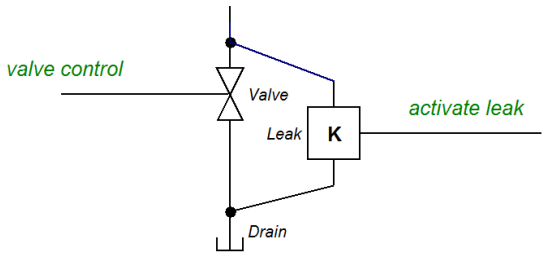


Fig. 5. Block diagram model of a valve which can exhibit a leak.

valve control activates the flow of water through the valve itself. As with the previous fault, this activation signal could be sent by the co-simulation framework executing a script to allow for fault injection. Using 20-sim to model this fault also allows for more complex models, for example a leak where the rate of flow depends on the pressure.

A scenario that could be explored with this leaky valve in the plant model would be to see whether or not the controller could discover that such a leak existed, based on the sensors with which it can observe the plant. Currently, the controller would only discover a potential leak if the low water sensor was activated while the valve was closed. Based on this discovery, we might suggest additional sensors that would allow the controller to discover a leak sooner.

4 Towards the Exploration of Design Alternatives

As indicated in Section 1, our practical approach to formal modelling of embedded systems must address the need to support the comparison and selection of design alternatives. Section 3.4 discussed some of the choices that developers can face in constructing a co-model for a simple control system that includes fault behaviour. In this section, we look towards the support for design space exploration based on the use of co-models, using a more substantial case study.

The ChessWay is an industrial challenge problem originated by Chess. It is a self-balancing personal transporter, much akin to the well-known Segway [9]. The device has two wheels, mounted on either side of a base platform on which the rider can stand, holding on to a handlebar (Figure 6). The systems 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 therefore is to keep the ChessWay upright, even while stationary. It can do this by applying torque to each wheel independently, such that the base of the ChessWay is always kept directly underneath the centre of gravity of the entire system. The rider can move forward (and backward) by leaning forward (or backward). The controller measures the deviation angle of the handlebar and performs an immediate control action

[9] `http://www.segway.com/`

in order to keep the ChessWay stable, similar to the way that you might try to balance a pencil on the tip of your finger.

Fig. 6. The ChessWay personal transporter

The control laws for this kind of system are relatively simple, but the ChessWay remains 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 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, 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. In fact, any sudden deviation exceeding 10 degrees from upright could result in similarly violent control correction subjected to the user. This obviously diminishes the driving experience, which should be smooth and predictable. Moreover, it is intuitively clear that even small failures of the hardware or software could easily lead to the ChessWay malfunctioning.

The challenge is to find a modelling methodology that allows the system developer to define a controller strategy (based on several possible user scenarios), while reasoning about the suitability of the system under possibly changing environmental conditions and in the presence of potential faults. The need for this methodology is easily demonstrated by the well-known public debate regarding the legality of allowing the Segway on public roads. For the device to become street legal, its usability had to be demonstrated to several third parties (such as government road safety inspectors) and to private insurance companies. This is of course a significant challenge and representative for many industrial products being developed today.

In the ChessWay case study, we should be able to specify multi-modal controller behaviour. For example, the controller should contain a start-up procedure in which the user must manually hold the ChessWay upright for a certain amount of time, before the

controller will begin to balance the device actively. Similarly, the user may step off the platform and the controller needs to be turned off at some point. Furthermore, we wish to model an independent safety controller which monitors and intervenes in the case of extreme angles, hardware failure, sensor failure and so on. In addition, we wish to model: a joystick, allowing the user to turn the ChessWay; degraded behaviour, based on low battery level; a safety key, in case the user falls off; a parking key, allowing the user to safely stop the ChessWay; and feedback to the user, in the form of LED indicators. It is clear that even this simple case study demonstrates the intrinsic complexity of modern real-time control systems.

There are numerous faults which we would hope to explore by developing and co-simulating a ChessWay co-model. These include sensor failures (e.g. missing, late, or jittery data) of the accelerometer, gyroscope, safety key and steering joystick. Other issues can arise with hardware, such as battery degradation, communication bus faults, and CPU crashes. In addition, complex environmental factors are not currently modelled. For example, uneven surfaces or those where the wheels experience different friction; or scenarios in which the ChessWay collides with obstacles or loses ground contact. Users can also cause faults, such as rapidly leaning forward on the ChessWay, which can lead to a dangerous overreaction of the controller [10].

The trade-off between safety, functionality and cost price is a typical bottleneck during system design. There are numerous factors to be explored in the ChessWay study. Low-cost accelerometers may be sufficient to meet the basic system requirements, while a more expensive IMU (Inertial Measurement Unit) could deliver a wider safety margin, reduced complexity and better performance, but at a higher cost. Choice of motor, desired top speed and desired running time will affect choice of battery capacity. In turn, battery capacity affects the size and weight of the battery, which affects the design of the frame and so on. Electronics and processors must be selected to meet the timing requirements. Deciding between the myriad options is envisioned as a typical design space exploration task in the DESTECS project.

The large variety in usage scenarios, functional requirements, environmental conditions and fault types described above makes it clear that suitable analysis of any design can only be sensibly done semi-automatically. It is also clear that current state of the art modelling technology does not provide efficient means to do so (at the appropriate level of abstraction). Rapid co-model analysis by simulation will be used in the DESTECS project to explore the design space. This iterative process rates each possible design on a number of predefined quantitative and qualitative (and possibly conflicting) design objectives for predefined sets of scenarios, environment conditions and faults. This ranking provides objective insight into the impact of specific design choices and this guides, supports and logs the decision making process.

5 Concluding Remarks

We have presented an approach to collaborative modelling and co-simulation with an emphasis on exploring the design space of alternative fault models. Our work is in

[10] Search `www.youtube.com` for "segway crashes" for examples of malicious users.

very early stages, but we have already demonstrated the feasibility of coupling discrete event models in VDM with continuous-time models in 20-Sim using existing tools. We believe that we have viable apparatus on which to build tools that will allow design exploration in terms co-models (and explicit fault models in particular).

Several authors have argued that the separation of the physical and abstract software worlds impedes the design of embedded systems. Henzinger and Sifakis conclude that a new discipline of embedded systems design, developed from basic theory up, is required [14]. Lee argues that new (time-explicit) abstractions are needed [20]. Our approach brings relevant timing requirements into otherwise conventional formal controller models. We expect that developing methods and tools for collaborative modelling and co-simulation, especially for fault-tolerant systems, will yield insights into the mathematical frameworks required for a unified discipline.

Our work aims for a pragmatic, targeted exploitation of formal techniques to get the best value from currently under-exploited formal methods and tools. Note that in early design stages, we are not interested in a design that is provably correct under all circumstances, but in finding the class of system models that is very likely to have that property when studied in more detail. This is sound engineering practice, because in reality the cost involved in performing this detailed analysis for a specific design is usually significant and can be performed at most once during the design of a system.

The concept of co-simulation has also attracted interest. Nicolescu et al. [22,21] propose CODIS, a co-simulation approach based on a generic "co-simulation bus" architecture. Verhoef's semantics [24] appears to differ from CODIS in that there is a direct connection between the CT interface and the operational semantics of the DE system. The COMPASS project [3] aims to support co-engineering of critical on-board systems for the space domain, using probabilistic model checking of AADL models extended with an explicit notion of faults. The MODELISAR[11] project shares many goals with the DESTECS project, particularly in using co-simulation to aid collaborative design. It is focused on the automotive industry and uses the Modelica [13] modelling language. The main output is the definition of a "Function Mock-up Interface" (FMI), which is essentially a specification for co-simulation. Models can be co-simulated by generating C-code which implement this FMI. There is less focus on explicit fault modelling.

Ptolemy [7] is a radically different actor-based framework for the construction of models of complex systems with a mixture of heterogeneous components, where different models of computation can be used at different hierarchical modelling levels. The Ptolemy execution semantics provides facilities that address similar goals to co-simulation – this is especially true for the built-in tool HyVisual.

Our exploration of co-models is already beginning to raise interesting questions for future work. Even the simple water tank example shows the choices facing modellers – some of which might not be explicit in more conventional development processes. For example, faults can be modelled on either side of the co-model and there is flexibility in how much fault behaviour is encoded within the model, as opposed to the external script driving a co-simulation. Practical experience will yield guidelines for modellers in future. Further, our goal is to develop patterns that allow the automatic (or semi-automatic) enhancement of normative models with descriptions of faulty behaviour.

[11] `http://www.modelisar.org/fmi.html`

We have argued that co-simulation should not be decoupled from design space exploration. A necessary activity is to develop forms of ranking and visualisation for test outcomes in order to support model selection. Further, a means of validating system-level timing properties by stating validation conjectures is required [9]. Regarding the co-simulation framework, there are open questions. Here there are open questions about how open the co-model's interface to the script should be. For example, should private variables and operations be made available to the script, or should the information hiding (in the DE models in particular) be enforced on the script? These are currently open points.

Finally, collaborative modelling and co-simulation have the same strengths and weaknesses as all formal modelling. The predictive accuracy of models depends in turn on the accuracy with which properties, especially timing properties, of components can be determined. The linking of heterogeneous models by co-simulation contracts brings into the light many of the choices and conflicts that are currently only poorly understood, or are made implicitly or with weak justification. It is to be hoped that exposing these decisions to scrutiny will help to reduce the late feedback and rework that characterises so much embedded systems development today.

Acknowledgements We are grateful to our colleagues in the EU FP7 project DESTECS, and we especially acknowledge the contributions of Jan Broenink. In addition we would like to thank Nick Battle for providing input on this paper. Fitzgerald's work is also supported by the EU FP7 Integrated Project DEPLOY and by the UK EPSRC platform grant on Trustworthy Ambient Systems (TrAmS).

References

1. Andrews, Z.H., Fitzgerald, J.S., Verhoef, M.: Resilience Modelling through Discrete Event and Continuous Time Co-Simulation. In: Proc. 37th Annual IFIP/IEEE Intl. Conf. on Dependable Systems and Networks (Supp. Volume). pp. 350–351. IEEE Computer Society (June 2007)
2. Avizienis, A., Laprie, J.C., Randell, B., Landwehr, C.: Basic concepts and taxonomy of dependable and secure computing. IEEE Transactions on Dependable and Secure Computing 1, 11–33 (2004)
3. Bozzano, M., Cimatti, A., Katoen, J.P., Nguyen, V.Y., Noll, T., Roveri, M.: The compass approach: Correctness, modelling and performability of aerospace systems. In: SAFECOMP '09: Proceedings of the 28th International Conference on Computer Safety, Reliability, and Security. pp. 173–186. Springer-Verlag, Berlin, Heidelberg (2009)
4. Broenink, J.F., Larsen, P.G., Verhoef, M., Kleijn, C., Jovanovic, D., Pierce, K., F., W.: Design support and tooling for dependable embedded control software. In: Proc. of Serene 2010 International Workshop on Software Engineering for Resilient Systems. ACM (April 2010)
5. Broenink, J.F.: Computer-aided physical-systems modeling and simulation: a bond-graph approach. Ph.D. thesis, Faculty of Electrical Engineering, University of Twente, Enschede, Netherlands (1990)
6. Broenink, J.F.: Modelling, Simulation and Analysis with 20-Sim. Journal A Special Issue CACSD 38(3), 22–25 (1997)
7. Eker, J., Janneck, J., Lee, E., Liu, J., Liu, X., Ludvig, J., Neuendorffer, S., Sachs, S., Xiong, Y.: Taming heterogeneity – the Ptolemy approach. Proceedings of the IEEE 91(1), 127–144 (January 2003)

8. Elmstrøm, R., Larsen, P.G., Lassen, P.B.: The IFAD VDM-SL Toolbox: A Practical Approach to Formal Specifications. ACM Sigplan Notices 29(9), 77–80 (September 1994)
9. Fitzgerald, J.S., Larsen, P.G., Tjell, S., Verhoef, M.: Validation Support for Real-Time Embedded Systems in VDM++. In: Cukic, B., Dong, J. (eds.) Proc. HASE 2007: 10th IEEE High Assurance Systems Engineering Symposium. pp. 331–340. IEEE (November 2007)
10. Fitzgerald, J., Larsen, P.G.: Modelling Systems – Practical Tools and Techniques in Software Development. Cambridge University Press, The Edinburgh Building, Cambridge CB2 2RU, UK, Second edn. (2009), ISBN 0-521-62348-0
11. Fitzgerald, J., Larsen, P.G., Mukherjee, P., Plat, N., Verhoef, M.: Validated Designs for Object–oriented Systems. Springer, New York (2005), http://www.vdmbook.com
12. Fitzgerald, J., Larsen, P.G., Sahara, S.: VDMTools: Advances in Support for Formal Modeling in VDM. ACM Sigplan Notices 43(2), 3–11 (February 2008)
13. Fritzson, P., Engelson, V.: Modelica - a unified object-oriented language for system modelling and simulation. In: ECCOP '98: Proceedings of the 12th European Conference on Object-Oriented Programming. pp. 67–90. Springer-Verlag (1998)
14. Henzinger, T., Sifakis, J.: The Discipline of Embedded Systems Design. IEEE Computer 40(10), 32–40 (October 2007)
15. Information technology – Programming languages, their environments and system software interfaces – Vienna Development Method – Specification Language – Part 1: Base language (December 1996)
16. J. S. Fitzgerald and P. G. Larsen and M. Verhoef: Vienna Development Method. Wiley Encyclopedia of Computer Science and Engineering (2008), edited by Benjamin Wah, John Wiley & Sons, Inc
17. Karnopp, D., Rosenberg, R.: Analysis and simulation of multiport systems: the bond graph approach to physical system dynamic. MIT Press, Cambridge, MA, USA (1968)
18. Larsen, P.G., Battle, N., Ferreira, M., Fitzgerald, J., Lausdahl, K., Verhoef, M.: The Overture Initiative – Integrating Tools for VDM. ACM Software Engineering Notes 35(1) (January 2010)
19. Larsen, P.G., Fitzgerald, J., Wolff, S.: Methods for the Development of Distributed Real-Time Systems using VDM. International Journal of Software and Informatics 3(2-3) (October 2009)
20. Lee, E.A.: Computing needs time. Communications of the ACM 52(5), 70–79 (May 2009)
21. Nicolescu, G., Boucheneb, H., Gheorghe, L., Bouchhima, F.: Methodology for efficient design of continuous/discrete-events co-simulation tools. In: Anderson, J., Huntsinger, R. (eds.) High Level Simulation Languages and Applications. pp. 172–179. SCS, San Diego, CA (2007)
22. Nicolescu, G., Bouchhima, F., Gheorghe, L.: CODIS – A Framework for Continuous/Discrete Systems Co-Simulation. In: Cassandras, C.G., Giua, A., Seatzu, C., Zaytoon, J. (eds.) Analysis and Design of Hybrid Systems. pp. 274–275. Elsevier (2006)
23. Oxford English Dictionary Online. Oxford University Press (2010)
24. Verhoef, M.: Modeling and Validating Distributed Embedded Real-Time Control Systems. Ph.D. thesis, Radboud University Nijmegen (2008), ISBN 978-90-9023705-3
25. Verhoef, M., Visser, P., Hooman, J., Broenink, J.: Co-simulation of Real-time Embedded Control Systems. In: Davies, J., Gibbons, J. (eds.) Integrated Formal Methods: Proc. 6th. Intl. Conference. pp. 639–658. Lecture Notes in Computer Science 4591, Springer-Verlag (July 2007)
26. Wolff, S., Larsen, P.G., Noergaard, T.: Development Process for Multi-Disciplinary Embedded Control Systems. In: EuroSim 2010. EuroSim (September 2010)
27. Woodcock, J., Larsen, P.G., Bicarregui, J., Fitzgerald, J.: Formal Methods: Practice and Experience. ACM Computing Surveys 41(4), 1–36 (October 2009)