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

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

J. S. Fitzgerald, P. G. Larsen, K. G. Pierce and M. H. G. Verhoef

Abstract

The development of embedded computing systems poses significant challenges. The increasing complexity of distributed control and the need to provide evidence to support assurance of safety suggest that there is merit in adopting model-based formal methods. However, such approaches require effective collaboration between the engineering disciplines involved, and in particular 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 later be replaced by couplings to continuous-time models. An operational semantics of co-simulation then 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.
Abstract

The development of embedded computing systems poses significant challenges. The increasing complexity of distributed control and the need to provide evidence to support assurance of safety suggest that there is merit in adopting model-based formal methods. However, such approaches require effective collaboration between the engineering disciplines involved, and in particular 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 later be replaced by couplings to continuous-time models. An operational semantics of co-simulation then 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.

About the authors

John Fitzgerald is Reader in Computing Science and Deputy Director of the Centre for Software Reliability at Newcastle University. He is the Newcastle Principal Investigator in the DESTECS project, in which he leads work on methods for exploring design spaces for dependable embedded systems in terms of co-simulation. He is Chairman of Formal Methods Europe.

Peter Gorm Larsen is a professor at the Engineering College of Aarhus, and convener of the Overture tools initiative for VDM. He leads the DESTECS project's work on the development of open tools for co-modelling and co-simulation.

Ken Pierce is Postdoctoral Research Associate in the School of Computing Science at Newcastle, where he works principally on co-modelling and co-simulation of faults and fault tolerance mechanisms in DESTECS. Prior to joining DESTECS, he completed a doctorate on rely-guarantee conditions for atomicity refinement.

Marcel Verhoef has been with CHESS (Computer Hardware & System Software) in the Netherlands since 1999, latterly as a member of the innovation and iTech teams. In DESTECS, he leads work on industry applications of co-simulation. He is also an active contributor to the Overture initiative for VDM.

Suggested keywords

FORMAL METHODS
EMBEDDED SYSTEMS
CYBER-PHYSICAL SYSTEMS
DESIGN METHODOLOGY
FAULT MODELLING
A Formal Approach to Collaborative Modelling and Co-simulation for Embedded Systems

J. S. FITZGERALD\textsuperscript{1}, P. G. LARSEN\textsuperscript{2}, K. G. PIERCE\textsuperscript{1}, and M. H. G. VERHOEF\textsuperscript{3}

\textsuperscript{1} Newcastle University, UK
\textsuperscript{2} Aarhus School of Engineering DK,
\textsuperscript{3} CHESS Embedded Technology BV, NL

Received June 2011

The development of embedded computing systems poses significant challenges. The increasing complexity of distributed control and the need to provide evidence to support assurance of safety suggest that there is merit in adopting model-based formal methods. However, such approaches require effective collaboration between the engineering disciplines involved, and in particular 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 later be replaced by couplings to continuous-time models. An operational semantics of co-simulation then 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 an environment that may contain controlled devices, such as mechanical, electrical or hydraulic units, and external sources of inputs such as users and the physical environment. For example, an automobile cruise controller takes the current speed as sensor input, sends control commands to engine actuators, and reacts to inputs such as a new speed setting from the driver, and changes in road conditions.

The development of embedded control systems is challenging for several reasons. First, the volatile character of the embedded systems market demands rapid innovation and assessment of alternative product designs for increasingly complex and distributed systems. Second, in critical applications, evidence must be provided to the assurance process that relevant operational scenarios have been considered from an early development stage. Third, the development of embedded systems is inherently multidisciplinary. Software engineers may need to design controllers that implement control laws defined by specialists in the application technologies, such as mechan-
ical or electronic engineering. Development approaches that treat the disciplines separately run the risk of mis-communication, and this proves to be an inefficient way of handling the many cross-cutting design concerns.

Model-based development approaches have the potential to address some of the challenges of embedded systems development. Models produced in the early stages of development may be analysed with a view to identifying weak and strong design alternatives. Their analysis, and subsequent evolution into product specifications, provide evidence to the assurance process. Further, models can provide a common basis for collaboration between engineers. However, each discipline has its own culture of modelling and analysis, using its own established abstractions. For example, software is typically modelled using discrete-event formalisms, while mechanical and electrical systems use models based on continuous-time descriptions of phenomena defined in terms of differential equations. Effective model-based design for embedded control systems should bridge this gap. Our work concerns collaborative modelling, by which we refer to the production of composite models (called co-models) incorporating both a discrete-event (DE) model of a digital controller, and continuous-time (CT) elements describing the environment, including the plant to be controlled.

Formal methods applied to software development use modelling notations that have mathematically well-founded semantics. The benefits claimed for the use of formal methods include improved communication between stakeholders, including those involved in certification, and the ability to analyse functional, safety and performance properties, identifying flaws before the expensive commitment is made to a detailed design. Some of the most successful applications use formal methods in a “lightweight” way, applying fully formal notations, but targeting them on parts of systems that carry a high development risk because of their criticality or complexity (Jackson and Wing, 1996; Woodcock et al., 2009).

Our work concerns the use of formal methods to assist collaborative modelling in embedded systems design. The existence of a formal semantics for modelling notations can be expected to allow the principled coupling of heterogeneous models from different disciplines. We focus on the exploitation of formal semantics to enable the analysis of executable formal models by simulation. Although simulation-based techniques are not as exhaustive as symbolic analysis using model-checking or proof, they do not generally entail the construction of special abstractions to manage the state space, but allow a high level of automation, and are familiar to the majority of engineers. As a result, they may be applied rapidly in early development stages in order to eliminate infeasible designs, or select promising ones for further analysis. We refer to the simulation of heterogeneous co-models as co-simulation.

The aim of our work is to develop methods and tools to support the construction and co-simulation of co-models that contain discrete-event and continuous-time models together, in order to assist in the rapid selection of design alternatives at early stages of development. The Vienna Development Method (VDM) is used for modelling discrete-event controllers, and 20-sim (a modelling and simulation environment for bond graph models) forms the continuous-time framework for modelling the environment. Both are well-established formalisms with good tool support and a record of industry use, suggesting that they could form a viable basis for collaborative modelling in practice. A basic proof-of-concept of the co-simulation of VDM with 20-sim has been reported previously, but has not addressed the process of co-model construction (Fitzgerald et al., 2010). The contribution of this paper is to propose and demonstrate a
pragmatic, tool-supported approach to the construction of co-models. This starts from an exclusively discrete-event model that uses discrete approximations of environment components, gradually replacing these with continuous-time models when greater fidelity is required and as these 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. This substantial example is also described.

A brief introduction to collaborative modelling and simulation in VDM and 20-sim is given in Section 2. Our approach to the staged process of model construction is outlined in Section 3. We describe a case study based on the design of the “ChessWay”, a personal transporter device with interesting safety challenges, in Section 4. We link our work to related activities in co-simulation 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. The formal modelling notations of 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 the co-simulation in our setting.

2.1. The Vienna Development Method

We use the Vienna Development Method (VDM) as the formalism for discrete-event models of controllers. VDM’s origins lie in the definitions of semantics of programming languages, notably at IBM’s Vienna Laboratory, in the 1970s (Bjørner and Jones, 1978; Jones, 1999). The basic modelling language (VDM-SL) and its denotational semantics have been standardised (ISO, 1996; Larsen and Pawlowski, 1995). A proof theory has been defined, based on the typed logic of partial functions (Jones and Middelburg, 1993; Bicarregui et al., 1994). Extensions to the language have introduced object-oriented structuring and concurrency (Fitzgerald et al., 2005). A further extension of VDM++ called VDM Real Time (VDM-RT) is the dialect used in this paper.

A goal of VDM’s development since the mid-1990s has been to develop modelling and analysis techniques that are accessible to the majority of systems and software engineers, and do not require deep understanding of the form of the underlying semantics. Stress has been laid on the development of robust and efficient tools for simulation of executable models, rather than proof, and on links to other less formal modelling frameworks, such as UML. Current tool support for VDM includes the commercial VDMTools, and the more recent open-source tool Overture (Fitzgerald et al., 2008; Larsen et al., 2010). The emphasis on simulation has led to the development of a very efficient interpreter for the executable subset of VDM (Lausdahl et al., 2011). As a consequence of this approach, there is an ongoing record of successful industry deployment (Larsen and Fitzgerald, 2007; Kurita and Nakatsugawa, 2009).

Work on the application of VDM to controllers required the extension of the language to allow explicit modelling of computation times on virtual networked processors (Verhoef et al., 2006; Hooman and Verhoef, 2010). An approach to the construction of VDM models of distributed real-time systems has been developed (Larsen et al., 2009). 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 couples VDM with a continuous-time modelling formalism was proposed by Verhoef and others (Verhoef et al., 2007; Verhoef, 2008) and forms part of the a technical foundation of the EU DESTECs research project that is developing the associated tools and methodology†.

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. In the remainder of this section, we introduce the basic characteristics of that formalism, using the simple and common example of a controller that manages a water tank (Figure 1). The tank is continuously filled by the input flow $\varphi_{in}$, and can be drained by opening the valve, resulting in the output flow $\varphi_{out}$. The output flow through the valve when this is opened or closed is described by Equation 2 in Figure 1, where $\rho$ 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 2 (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.

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

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

Fig. 1. Water tank level controller example

Models in VDM focus on the logical specification of data and functionality. The user may define data types by means of constructors for such abstractions as records, sets, sequences and mappings (finite functions) and primitive data types such as Booleans, natural numbers and real numbers. Types may be restricted by invariants recorded as predicates. Persistent state is modelled by means of typed variables. Functionality may be described in terms of referentially transparent functions, or by operations that may have side-effects on state variables. Functions and operations may be specified explicitly, or implicitly by means of postconditions. 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 exist between classes. The architecture of such VDM models can be readily expressed using a subset of UML.

†http://www.destecs.org/
VDM-RT contains abstractions that allow threads to be deployed to models of CPUs (each potentially with different capacities and scheduling policies) linked by buses, and for time budgets to be allocated to operations (Verhoef et al., 2006). The declaration of the CPUs and buses, and the deployment of components, take place in a special system class. Threads can also be declared to be periodic with specified jitter, period and offset from the start of the thread. Operations may be asynchronous. Time is explicit in VDM-RT models, and the duration of a computation can be specified in terms of an absolute time or in terms of the number of cycles required from the CPU on which it is deployed. The system class is implicitly constructed by the VDM interpreter at start-up, after which the internal clock starts.

A VDM model of one possible controller for the water tank is shown in Figure 2 (b). This contains a single instance variable that models the interface of the controller to the tank (the definition of Interface is elsewhere), 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 2 (b) contains mutual exclusion constraints that prevent multiple threads accessing the interface simultaneously.

```
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
```

Fig. 2. Water tank example: (a) 20-Sim model (left) and (b) event-driven controller in VDM (right).

The style of controller model shown in Figure 2 (b) 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). By contrast, in a time-triggered controller an operation modelling a control loop processes input
data and decides on an appropriate response. This response is usually in the form of modified actuator settings. The example in Figure 3 shows a time-triggered alternative to the event-driven controller in Figure 2 (b). It is a simple periodic thread in a discrete-event controller. Input from a sensor object \texttt{levelsensor} is obtained and processed, leading to the invocation of operations on a \texttt{valveActuator} object. Note that this model requires a sensor that returns the current water level required, rather than the two high- and low-level sensors in the event-driven version. The arguments to the \texttt{periodic} keyword give the period, jitter, delay and offset; in this example, the thread executes the \texttt{controlLoop} operation every 5ms.

2.2. \textit{20-sim}

\textit{20-sim} is a package for modelling and simulating complex physical systems (Broenink, 1997; Kleijn, 2006; van Amerongen, 2010). 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 common underlying formalism uses \textit{bond graphs} (Karnopp et al., 2000; Breedveld, 1985; Broenink, 1990). Bond graphs are directed graphs in which the vertices are submodels and the edges, called \textit{bonds}, denote the ideal (or idealised) exchange of energy. Entry points of submodels are called \textit{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. For each physical domain, such a pair of variables can be specified, for example: voltage and current, force and velocity.

The \textit{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

```
operations

async private controlLoop : () ==> ()
controlLoop() ==
  cycles(2)
  ( --obtain sensed level
    dcl level : real := levelSensor.GetValue();

    -- calculate response
    if level >= maxLevel
      then valve.setValue(true);
    if level <= minLevel
      then valve.setValue(false);
  );

thread
  periodic(5,0,0,0)(controlLoop); -- 200Hz
```

Fig. 3. Water tank example: Time-triggered controller in VDM
the behaviour of nodes; and iconic diagrams. 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 parts. 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.

2.3. Co-Simulation

For our purposes, a typical co-model is composed of a controller and an environment containing the plant ("that part of the system which is to be controlled" (IEEE, 2000)) and environmental stimuli. These stimuli include those related to normal functioning of the system as well as disturbances that tend to deflect the plant from desired behaviour. The controller can affect the plant directly by means of actuators and receive feedback via sensors.

A model is an abstract representation of a system of interest. A model is competent for a given analysis if it contains sufficient detail to permit that analysis. A symbolic execution of a model is called a simulation. A model which can be simulated is called an executable model. The inputs and stimuli provided to a model during a simulation run is termed 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 the coupling of discrete-event (DE) and continuous-time (CT) simulations running in separate tools. In a discrete-event simulation, "only the points in time at which the state of the system changes are represented" (Robinson, 2004). 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." (Robinson, 2004). 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. This is termed a co-simulation. In a co-simulation, a shared variable is a variable that appears in and can be accessed from both the DE and CT 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 with predicates (Boolean expressions), where the changing of the local 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.

A co-model is a model comprising a DE model, a CT model, and a contract describing the communication between them. Shared variables, shared design parameters, and events define the nature of the communication between models. These elements are recorded in the contract. Only one submodel (either the DE model or the CT model) may have write access to a shared
variable. In the control-system paradigm, shared variables written to by the DE submodel are called *controlled* variables and those written to by the CT submodel are called *monitored* variables.

The operational semantics of co-simulation is described in terms of synchronisation of the two simulators (Verhoef, 2008; Hooman and Verhoef, 2010). 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 4 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 $\sigma_{de}$ be the internal state of the DE simulation at simulation time $t_{de}$, and let $\sigma_{ct}$ 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 $\sigma_{c}$) are set by the DE controller and read by the environment; the monitored variables ($\sigma_{m}$) 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 $\delta t$ 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 $\sigma_{m}$ and the actual time that it reached $\delta t_a$ 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 $\sigma_{de}$. The DE simulation then advances by $\delta t_a$ so that the both DE and CT are again synchronised at the same simulation time, and the controlled variables are updated ($\sigma_{c}^{1}$) 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 semantics 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. The co-simulation follows a master/slave paradigm in which a central engine orchestrates the DE (Overture) and the CT (20-sim) simulators. The co-simulation engine determines how far in time each of the simulators is allowed to proceed before returning control to the master ($\delta t$ in Figure 4). 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 $\delta t$ 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. In addition to the standard libraries supplied for VDM, libraries for the commonly used feedback controllers (P, PI and PID) are included in the DESTECS development environment. These may be incorporated into co-models and tuned to the needs of each application.

3. A “Discrete-event First” Approach to Co-model Development

The use of co-models is intended to improve the concurrent development of embedded control systems by promoting the interaction of engineers from distinct 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. This section proposes a pragmatic approach to co-model construction that starts from the discrete-event side, allowing the gradual introduction of continuous-time models of elements of the system when ready and appropriate.

Several factors in the development of modern embedded systems create the need to consider the design of the controller from an early stage. Environment models can be complex. Consequently, a significant part of the risk in developing modern control systems lies in the logic of the controller, and particularly the supervisory controller that manages higher-level functions such as the switching of control modes, and identification and recovery from errors. This aspect of supervisory control assumes importance if systems under development are safety-related. Where controllers are distributed, this adds further to the complexity.

In these circumstances, and based on experience gained in the case studies identified in the DESTECS project, we propose using a “DE-first” approach, whereby models are produced initially using the DE formalism for both the controller and the environment. Some of the system components, especially those in the environment, might 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. A complimentary approach from

---

$\dagger$ The sources and the executables can be found at SourceForge (https://sourceforge.net/projects/desteecs/)

$\S$ An informally stated rule of thumb, albeit one for which we have no empirical evidence, is that the lower-level loop controllers that realise the basic control laws typically constitute less than 10% of the entire code base of the system.
the continuous-time side may be considered equally well, with CT approximations of the DE controller being substituted for more developed DE models at a later stage. We return to this point in Section 6.

Section 3.1 introduces the approach, while technical features needed to support the strategy are considered in Sections 3.2-3.4.

3.1. Approaches to Co-model Construction

Consideration of the co-model structure suggests three alternative pragmatic approaches to the construction of initial co-models within the development processes outlined above. An ideal “contract-first” 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, especially one with distributed control, is not evenly distributed, and it may be advisable to focus early development efforts on the riskier components, whether DE or CT.

The process of developing a co-model “DE-first” begins with the construction of a single DE model covering both the control system and the environment. This is composed of two submodels describing the proposed controller and an approximation of the environment; the environment submodel will ultimately be replaced by a more accurate CT model. This model structure is described in Section 3.2. It should be possible to run a simulation of the initial DE model in order to gain a basic level of confidence in the validity of the control approach, so the behaviour of environment elements that are ultimately to be modelled in the CT formalism are mimicked by DE data and operations. 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 3.3). 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 3.4) which is then subject to co-simulation.

3.2. 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 this type of model is given as an abstract class diagram in Figure 5(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 (indicated by the double vertical lines at their edges) meaning that they define a thread. This shows that the controller and environment run concurrently during a simulation.

The sensors and actuators are shown as interfaces, which implies that 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 implementation 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 5(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 nature of communication between these models (for instance, the types of the data passed via sensors and actuators). Co-simulation is enabled by the DESTECS tool, which allows co-models to be defined and handles tool synchronization as described in Section 2.3–2.4.
3.3. 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-RT. In a sense, it implements a basic simulator running in a separate thread of the model 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 typically run in fixed time steps at a much coarser resolution than a full CT model.

Since VDM-RT is a general language, it might be possible to build a realistic model of the environment given enough time and effort, but for the DE-first approach the key is to find suitable approximations that allow controller logic to be tested in a reasonable time frame. The environment model will be composed of elements that model the devices in the physical environment of the controller. It must also handle input that comes 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.

For defining components of the plant, possible approximation strategies include replacing non-linear relationships with linear approximations and replacing complex differential equations with a mixture of simpler differential equations and constants. For example, consider the water tank example shown in Figure 1 (from Section 2.1). Here, the rate of flow of water from the tank is dependent on the volume of water, therefore as the tank empties, the rate will decrease. A DE-first approximation of such a system might involve a linear approximation using 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 state, 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.

Another example to consider is that of a DE approximation of a motor, with a controller that can request a power level between 0% (no power) and 100% (full power). In an electric motor, a current is applied to it which induces a magnetic field. This in turn gives rise to a moment of force (torque) that causes the motor to rotate. The speed of the motor is determined by many factors including its internal structure, mass and various elements of friction. While a CT model could take all these factors into account, a DE approximation could assume that 100% power yields a maximum acceleration and that lower power levels yield an acceleration that scales linearly (e.g. 50% power gives half the acceleration of full power).

When defining external input, the principle of using linear approximations also applies. Here however we will typically wish to define some behaviours that can be used as scenarios in simulations, for example, user input. We will consider two different approaches to the same example, specifically a user rotating a dial fifteen degrees clockwise and back again. A real measurement of such an action is likely to initially show a large increase in rotation angle, slowing down as the user reaches the desired angle, perhaps with some overshoot if the user turns too far. A visualization of this is given in Figure 6 as a plot of angle against time.

One approach to approximating the curve in Figure 6 is to identify and record the key points at which the curve changes and then have the environment simulation interpolate between these points linearly. Hence we could represent this curve as a sequence of pairs (2-tuples) each giving
a time and corresponding angle. A visualization of this approach is given in Figure 7(a). Using linear interpolation ensures a smooth, though approximate, change in angle.

If a less linear approximation of behaviour is required, then values can be pre-computed in some way and stored in a file. This can be read by the environment simulator to provide the given values at the correct time. A visualization of this approach is given in Figure 7(b). Depending on the number of values given in the file, this approach could yield a less smooth approximation than the previous method. These approaches could be combined of course, 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 some 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\(^5\)), which can easily export files containing the values of plotted data. This approach also offers the possibility that during the initial stages of a collaborative, concurrent 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.

---

\(^5\) [http://www.mathworks.com/products/matlab/]
Experience in producing DE-first approximations of the environment is currently limited, however we believe the above suggestions form a good basis for expansion with further recommendations as experience grows.

3.4. Co-model Creation

In the DE-first approach described in this paper, the eventual aim is to produce co-models by replacing the DE approximations of the environment and plant with more accurate CT models. When a DE-first model has reached sufficient maturity, the process of replacing components can begin. The modelling process outlined in Section 3.2 advocates the use of interfaces for sensors and actuators. This simplifies the migration because 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) broadly follows the following pattern:

1. A CT model of the environment is built.
2. 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.
3. 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 CT model.

When the DE model is incorporated into a co-model, the DE environment and 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 Section 2.3–2.4.

By using interfaces and alternative implementations of sensor and actuator objects in this way, 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 is is easy to revert back to using 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 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.

The degree of maturity required before migrating a DE-first model to a co-model will depend on a number of 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, in which case a co-model can be created where only some of the DE approximations are replaced by CT models.
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 case study, based on the ChessWay self-balancing scooter developed by 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, including 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).

We choose to present the ChessWay study "warts and all", rather than as an idealised application of the DE-first method. Consequently, it contains some deviations from the process and model structures described in Section 3, but we believe that these give more insight into the pragmatics of co-model construction than an artificial and perfect example. The deviations are relatively minor, and the production of a more conformant set of models is in progress. 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\[1\].

4.1. The ChessWay

The ChessWay has two wheels, mounted on either side of a platform on which the rider can stand, holding on to a handlebar (Figure 8). 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 therefore 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, in much the same way as one might attempt to balance an upright pencil on the end of one’s finger.

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 the user being subjected to similarly violent control actions. 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.

It should be possible to specify multi-modal controller behaviour for the ChessWay. For example, the controller should contain a start-up procedure in which the user must manually hold the ChessWay upright for a given period before the controller will begin to balance the device actively. Similarly, the user may step off the platform and hence the controller needs to be turned off at some point. Furthermore, we wish to model an independent safety controller that monitors and intervenes in the case of extreme angles, hardware failure, sensor failure and so on. In addition, we wish to model: a direction switch allowing the user to turn the ChessWay; degraded behaviour, based on low battery level; a safety key, in case the user falls off; 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.

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 DE model is built in VDM, using the VDM-RT extensions introduced in Section 2.1.

The purpose of the VDM-RT model presented here is to gain confidence in the control algorithm to be used for the ChessWay, including the deployment of different system components to different 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. Once the co-model is in place, it can be used to investigate potential error states, and to incorporate detection and recovery mechanisms for those judged as posing significant risks.

The structure of the actual ChessWay DE model follows the principles 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 9.

![Diagram of DE environment model](image)

**Fig. 9. ChessWay VDM-RT class diagram - environment model**

The ChessWay system 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 request left or right motion), a safety switch (a key attached by a cord to the rider’s wrist that becomes disconnected if the rider falls off the ChessWay), and an on/off switch. The controllers must communicate with each other and to collaborate in the overall control of the ChessWay. If the right controller requires to access the accelerometer and gyroscope, it must do so by communicating over the bus. The same is true of the left controller, which must access the three switches over the bus. The static topology and the 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 10.

Note that although all sensor classes are shown as implementing a common sensor interface in Figure 10, this is mainly used to indicate the methodological point that interfaces should be used between the controller and sensors. The same is true for actuators. In a full model, finding a common interface for all sensors and all actuators may be impractical. For example, a 3D accelerometer yields three real numbers, whereas a switch may yield a Boolean value.

4.2.1. Top-level model – the World class

A special World class is the top-level entry point of the ChessWay discrete-event specification. Within this class, instances of the controller and environment are created, then the RunScenario operation is called. This operation 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. These scenarios provide an abstract, discrete-time representation of their continuous-time behaviours.

The `RunScenario` operation instantiates the entire simulation model. The `ChessWay` class is implicitly constructed by the VDM interpreter at startup since it is declared as a system, as explained in Section 2.1 above. The model developed in the original ChessWay study, as shown here, deviates from the generic structure discussed in Section 3 in that a direct connection is established here between the environment and the controller, rather than via the sensor and actuator models. Although this facilitated rapid construction of the original model, the more decoupled
approach is preferred in general, and a version of the controller showing this architecture is under development.

The environment model is an abstraction of those elements in the physical, continuous-time world that have an impact on the behaviour of the control system. The primary elements we are concerned about here are the wheels and the rider standing on the ChessWay. Their behaviours are best described in continuous time, but at this stage we make a discrete-event approximation in order to evaluate the DE model of the controller.

First, the user scenario is loaded and the environment model is linked to the ChessWay system class. These links are static and help to facilitate the modelling and simulation process; they have no counterpart in the final implementation of the system. Once these links are put in place, simulation can commence by starting the ChessWay controllers using the PowerUp operations on both the left and right controller instances. These operations in turn will start the periodic controller threads internal to those objects. The penultimate step is to start the environment simulator thread in order to execute the scenario. Finally, the main Overture debug thread will wait for the simulation to finish, by calling waitForSimulationEnd, and control is given back to the user running the simulation.

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. For the ChessWay, plant elements include the motors, accelerometer and gyroscope. External elements include the rider.

As described in Section 3.3, we develop sensor and actuator interface classes as discrete approximations. The MotorSensor interface shown below defines an operation that returns data on the current state of a Hall effect sensor.

```plaintext
class MotorSensor

types
  public HallData = bool * bool * bool

operations
  public GetValue: () ==> HallData
  GetValue() == is subclass responsibility

end MotorSensor
```

In the DE model, the functionality defined by this interface might be supplied by the class MotorSensorDE as follows:

```plaintext
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

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:

mRightMotorSensor: MotorSensor
...
mRightMotorSensor := new MotorSensorDE(mRightWheel)

When it becomes appropriate to attach the CT model, the right motor sensor, instead of being a MotorSensorDE, can be an object that manages the link to the CT simulation via the co-simulation contract (see Section 4.3).

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 indicated in Section 3.3, a number of approximation schemes are possible. Here we describe in outline one approach used in the ChessWay DE model. The environment class includes a definition of 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 := {}
Fig. 11. Scenario of a user leaning forward $\pi/12$ radians (15 degrees) and then returning to an upright position

of the wheel, followed by updating 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
deriodic (1, 0, 0, 2.5E5) (mainLoop) -- 1kHz frequency
```

The `mainLoop` operation also determines whether or not the end of the simulation run has been reached, by checking the current simulation “wall clock” against a preset maximum simulation target time. Once this is reached, the `terminate` operation is called, which will stop execution of the model and return control back to the user.

4.2.3. Constructing the System Model – the `ChessWay` class

The `ChessWay` system class defines the distributed architecture on which the controller software is deployed. The system is composed of two CPUs connected by a bus. The bus enables communication between the controllers deployed on each processor, for example to exchange information relating to their internal state. The VDM model uses the built-in CPU and BUS abstractions available in the real-time extensions of VDM. The instance variables in the system class are defined as follows:
The CPU constructor takes 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 set of CPU instances connected together by the bus. The system constructor deploys the instances of the LeftController and the RightController, one to each CPU, which are named LeftCtrl and RightCtrl respectively:

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

This ChessWay system class is illustrated in Figure 12. Alternative system architectures deploying the functionality to different processors can be explored by changing this part of the model.

4.2.4. The Controller Class

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

```plaintext
class Controller

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

The periodic thread in the Environment and the periodic threads of the two controllers have the same parameters except for the initial offset. This causes the simulation to be clearly ordered, since the environment is always evaluated before the controller threads.

thread
    periodic (1, 0, 0, 0) (CtrlLoop) -- 1kHz

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 and by overriding the operation prototypes for CtrlLoop and PowerUp de-
A private instance variable mLeft provides the link to the left controller:

```plaintext
class RightController
  is subclass of Controller

instance variables
  public mSafetySwitch : SafetySwitch;
  public mOnOffSwitch : OnOffSwitch;
  public mDirectionSwitch : DirectionSwitch;
  ...
  private mLeft : [LeftController] := nil
```

The constructor of the RightController initialises the safety switch, the on/off switch and the direction switch. A separate operation is used by the World class to set mLeft to the actual left controller object created at start-up.

```plaintext
operations

public CtrlLoop: () => ()
CtrlLoop () ==
  duration (100)
    ( dcl hall : bool * bool * bool :=
      mMotorSensor.getHallSensorData(),
      safe : bool := mSafetySwitch.getStatus(),
      onoff : bool := mOnOffSwitch.getStatus(),
      dir : DirectionSwitch\'tDirectionStatus :=
        mDirectionSwitch.getStatus();
    let pwm = computeResponse(hall, safe, onoff, dir) in
    mMotorActuator.setPWM(pwm) )
```

The CtrlLoop operation reads the values of each sensor and calculates the motor response, which is then passed to the setPWM†† operation of the MotorActuator. The execution time of the control loop is defined in this model to be 100 msec. The computeResponse operation is used to calculate the motor response. It processes all of the input parameters obtained from the sensors in the environment. As a simple illustration of the model structure, we consider an extremely crude response computation that only uses the on/off switch. If the switch is set to on, then both motors will be set to run forward at 10 percent of their maximum power. If the on/off switch is reset, then the motor is set to idle (<FREERUNNING>):

```plaintext
†† PWM (Pulse Width Modulation) is a common method for controlling the amount of power given to an electric motor.
```
public computeResponse: (bool * bool * bool) * bool *
  bool * DirectionSwitch \tDirectionStatus => real
computeResponse (-, -, onoff, -) ==
  ( if onoff
    then ( mMotorActuator.setActuated();
      mLeft.mMotorActuator.setActuated() )
    else ( mMotorActuator.setFreeRunning();
      mLeft.mMotorActuator.setFreeRunning() );
  return 0.1 )
end RightController

Obviously a more sophisticated response would take the other inputs into account (and the safety
key in particular). 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 on the Overture
VDM tools. 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 3.4, 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 contract for the basic sensors and actuator for the right wheel has the following
simple form:

```plaintext
contract ChessWay

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

end ChessWay
```

The object of type MotorSensorDE that represented the right motor sensor is replaced by an
object of class MotorSensorCT.

```plaintext
class MotorSensorCT is subclass of MotorSensor

instance variables
```
Co-simulation for Embedded Systems

```plaintext
-- link to monitored variables
private hall1: bool;
private hall2: bool;
private hall3: bool;

operations

public GetValue: () ==> HallData
GetValue() ==
  return mk_HallData(hall1, hall2, hall3)
end MotorSensor
```

The instance variables hall1-3 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:

```
mRightMotorSensor: MotorSensor
...
mRightMotorSensor := new MotorSensorCT()
```

The co-simulation of the VDM model of the ChessWay controller and 20-sim model of the plant has been demonstrated. An example of the output generated from the CT side is shown in Figure 13. The graphical output 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.

5. Related Work

There is a substantial body of work on the provision of modelling and simulation for distributed control systems. The Mathworks, Inc.\textsuperscript{‡‡} provides a well-known commercial tool suite for development of control systems, Matlab/Simulink, with extensions that cater for multi-domain modelling such as StateFlow and SimMechanics, and explicit notions of architecture, most notably the academic tools TrueTime and JitterBug\textsuperscript{§§}. 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

\textsuperscript{‡‡} http://www.mathworks.com/
\textsuperscript{§§} http://www.md.kth.se/RTC/ARTIST2/publications/cervin+CACSD06.pdf
Fig. 13. Example co-simulation output produced by 20-sim showing a graph of actuator values over time and a corresponding 3D animation

such as IBM Rhapsody to Simulink\footnote{http://www.ibm.com/developerworks/offers/lp/demos/summary/r-rhapsodysim.html}. This is a notable step forward, the limited semantics of the overall approach does not remove the problem of complex model structuring and analysis reliability. IBM Rational Rose Real-time software models have been co-simulated with control laws specified in Matlab/Simulink (Hooman et al., 2004), using a platform-neutral notion of time. This is achieved by inserting an interface between Rose Real-time and Simulink, 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 (Campbell et al., 2006), which have a formally defined operational semantics, are likely candidates to overcome the issues mentioned before with Simulink, and this is 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 (Tudoret et al., 2000) and in the Giotto modelling approach (Henzinger et al., 2003).

Another approach proposed to support co-simulation comes from the ITEA2 project MODELISAR\footnote{http://www.modelisar.com/}, which provides a notion of a functional mockup interface (FMI) for model exchange and tool coupling. The FMI basically constitutes a co-simulation contract, but this approach exposes a significant amount of detail from the co-models over this interface, so that a common simulation execution platform is able to run the co-simulation. This differs from our approach in which tools are used “as-is” (that 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 the design of continuous-time / discrete-
Co-simulation for Embedded Systems

6. Conclusions and Future Work

In this paper 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 demonstration of the approach on a substantive case study based on the ChessWay. 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, in which CT controller models mimic DE event co-simulation tools (Nicolescu et al., 2006), for which they provide an operational semantics (Gheorghe et al., 2006). Our work is in fact an instantiation of that architecture, with a difference. Their approach is aimed at connecting multiple simulators on a simulation bus, much akin to MODELISAR, whereas we currently connect two simulators using a point-to-point connection. 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 (Theelen et al., 2007). In this area, experiments with co-simulation have been started up with 20-sim.

Ptolemy-II proposes a component-based, actor-oriented approach (Eker et al., 2003) in which components are concurrent and interact by message-passing. Recently a significant effort has been invested to formalise the semantics of this framework (Lee and Zheng, 2007). The communication protocol and the concurrency policies together provide the model of computation. Ptolemy-II is a system-level design environment that supports heterogeneous modelling and design using this approach (Davis et al., 1999). It supports several domains, each of which is based on a particular model of computation, such as for example discrete event, synchronous data flow, process networks, finite state machines and communicating sequential processes. The Kepler extension provides first attempts to deal with distributed computing. In Ptolemy, co-simulation is carried out within the same tool. The approach differs from ours in the limitations on the level of abstraction available for modelling the DE part.

Recently, Myers et al. have described an approach to co-modelling and co-simulation for analysing hardware/software partitioning that integrates Behavior Engineering models with a component-based model of a virtual environment in the Modelica language (Myers et al., 2011). 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.
behaviour. Once it becomes necessary to model the data and logic of the controller in greater detail, it would be natural to replace these by DE controller models expressed using a rich notation such as VDM. Within the DESTECS project we are running pilot studies to better understand the communications between collaborating engineers “DE-side” and “CT-side” and provide appropriate support in the form of patterns and libraries of reusable component models.

In our current work, we also focus on ways in which we can model abnormal behaviour, whether caused by conventional faults or “malicious” users, and defences against these, including fault tolerance mechanisms that may protect against these. For example, we are modelling approaches to ensuring safety in the ChessWay, including the use of safety kernel architectures and having a separate safety monitor. Our intention is to examine such alternatives under a common set of scenarios, in particular the potential failure modes resulting from the distributed controllers of the two wheels.

As stated in Section 1, the aim of our work is to develop methods and tools to support the collaborative modelling and co-simulation of discrete-event and continuous-time models together, in order to assist in the rapid selection of design alternatives at early design stages. The progress reported in the paper concerns the methods of co-model construction and, although the approach appears to be promising, 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 (e.g. Figure 13) 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 area are in progress, using a variety of approaches, including DE-first and CT-first model development, and with a diverse set of dependability characteristics. We hope to report on this increasing range of applications in future.

Acknowledgements

This work is supported by the EU FP7 DESTECS Project. We are grateful to our colleagues for many useful discussions, notably Nick Battle, Bert Bos, Jan Broenink, Christian Kleijn, Kenneth Lausdahl, Angelika Mader, Yunyun Ni, Augusto Ribeiro, Yoni De Witte, Sune Wolff and Xiaochen Zhang. Fitzgerald’s work is additionally supported by the UK EPSRC Platform Grant on Trustworthy Ambient Systems (TrAmS).

References


