Advanced Engineering Informatics 28 (2014) 232–240

Contents lists available at ScienceDirect

Advanced Engineering Informatics

journal homepage: www.elsevier.com/locate/aei

A co-modelling method for solving incompatibilities during co-design of mechatronic devices

Yunyun Ni [*], Jan F. Broenink

Robotics and Mechatronics Group, CTIT Institute, University of Twente, P.O. Box 217, 7500 AE Enschede, The Netherlands

A R T I C L E I N F O

Article history:
Received 30 September 2013
Received in revised form 7 May 2014
Accepted 31 May 2014
Available online 11 July 2014

Keywords:
Cyber-physical systems
Co-design
Co-modelling
Event detection
Controller layered structure
Fault modelling

A B S T R A C T

The design process of mechatronic devices, which involves experts from different disciplines working together, has limited time and resource constraints. These experts normally have their own domain-specific designing methods and tools, which can lead to incompatibilities when one needs to work together using these those methods and tools. Having a proper framework which integrates different design tools is of interest, as such a framework can prevent incompatibilities between parts during the design process. In this paper, we propose our co-modelling methodology and co-simulation tools integration framework, which helps to maintain the domain specific properties of the model components during the co-design process of various mechatronic devices. To avoid expensive rework later in the design phase and even possible system failure, fault modelling and a layered structure with fault-tolerance mechanisms for the controller software are introduced. In the end, a practical mechatronic device is discussed to illustrate the methods and tools which are presented in this paper in details.

1. Introduction

Designing mechatronic devices is a challenging task for various reasons: (1) There are limited time and resources (money, human) for new innovation and assessment of mechatronic devices in current commercial markets; (2) New devices with novel functionality and an acceptable reliability need to reach the market before other competing products; (3) Mechatronic device development is a multi-disciplinary process, involving those who have domain-specific knowledge in their own field, such as electrical engineers, software engineers and mechanical engineers. In order to accomplish a design process in shorter design cycles, lower cost and better quality, engineers often use their own domain-specific design tools to perform simulations to avoid expensive physical prototypes in early design stages. Mechatronic devices are currently also known as *Cyber-Physical Systems* (CPS), as they consist of digital devices (computer, micro-controller, etc.) interacting with analogue (continuous-time) machines. Intrinsically, CPS have their incompatibilities, as the arithmetically and logically (binary) computed controller software is executed in discrete time while the dynamic plant is modelled in continuous time (i.e.: differential equations). Experts from different domains have different terms for the same concept or have the same term for different concepts. Both these cases are problematic, which can lead to serious problems later [1].

To handle possible fatal design flaws of mechatronic devices, modelling possible faulty behaviour of these devices and designing software that deals with this faulty behaviour at an early design stage is helpful to construct the actual devices “First-Time-Right”.

Methodologically, there are two major directions to perform modelling and simulation for CPS [2]: (1) use a *homogeneous* system model, i.e. using a single modelling language to express the whole CPS, and consequently use a single simulator and (2) use a *heterogeneous* system model, i.e. using different domain-specific modelling languages to model components from different domains, each simulated with their own simulator, and thus need a means to couple the involved simulators.

Using the *homogeneous* modelling approach, a model transformation from one domain to another is needed in order to model CPS in one single language. This regularly involves more abstractions and simplifications than originally planned,which in general compromises model fidelity. Furthermore, engineers from different specific domains often have the conceptual incompatibilities as mentioned above, causing misunderstandings and abstracting away relevant aspects, resulting in incompetent model parts. However, the single modelling formalism approach does work in case one of the domains is most relevant for the design: When one domain behaviour of the system is dominant, a system model

[*] Corresponding author. Tel.: +31 53 489 2626; fax: +31 53 4892223.
E-mail addresses: y.ni@utwente.nl (Y. Ni), j.f.broenink@utwente.nl (J.F. Broenink).

http://dx.doi.org/10.1016/j.aei.2014.05.004
1474-0346/© 2014 Elsevier Ltd. All rights reserved.

can be made in which the other domains are either ignored or simplified and formulated in the formalism of the dominant domain. For example, when the continuous-time (CT) part behaviour of the system is dominant, a purely CT representation can be made, in which the discrete-event (DE) part is abstracted away, or modelled very concise.

The *heterogeneous* system modelling approach preserves the hybrid properties of the systems by modelling the components in their own most suitable formalism. In this way, the CT components of CPS are modelled in one language which is best suitable for physical-systems dynamics modelling, while the DE components are modelled in an other appropriate modelling language. In this case, no sacrifice in any modelling domain needs to be made. This approach, however, has the risk that since each of the modelling formalisms and thus simulators has its own notion of time, they simply do not work together naturally. A proper synchronisation scheme to couple these different simulators is therefore needed. Simulation of such a combination takes in general more simulation time than when a homogeneous approach was used.

The proposed approach in this paper is to perform a co-modelling methodology (heterogeneous system modelling methodology) supported by a co-simulation tool framework which can address the incompatibilities described in the previous paragraphs.

Other research work has been done related to co-modelling methods and implementing the methods using tool frameworks: *Modelica* [3] is an object-oriented, equation based multi-domain language for simulating controlled physical systems, and provides a number of open and closed source libraries of physical components. However, in general, Modelica simulators cannot perform co-simulations that combine DE and CT computation domains together. The Functional Mockup Interface (FMI) [4] is a tool-independent standard for exchanging data between dynamic models, which is executed by implementing Functional Mock-up Units (FMU) that contain concrete mathematical models describing possible events in the related models. However, as it is indicated in [5], due to the fact that FMU is at a lower abstraction level comparing to Modelica and more target-oriented, it is less flexible. *Ptolemy* II [6] supports heterogeneous simulation from a methodology point of view, where per diagram a Model of Computation must be specified. It is implemented as a single tool. However, in [7,8], it was shown that dynamic plant modelling in Ptolemy II is less intuitive than 20-sim[1] (details about this tool will be mentioned later in Section 2.2.3).gCSP [9] is to graphically model concurrent process-oriented software based on the CSP formalism [10]. Co-simulation of networked control systems has been tried out [11], but the tool never reached maturity. *Cosimate*[2] is a backplane co-simulation tool offering interfaces to tools like Simulink, Modelsim, Modelica. Only time synchronisation is supported as exchanging data between simulators every time step. *Cosimate* has been tried out on the control of a mechatronic test set up [12]. The two discipline-specific models involved have to be connected in a rather cumbersome way.

In this paper, we discuss how our co-modelling method can help to solve the incompatibilities coming along with the designing process of mechatronic devices. In Section 2, we present some essential modelling and simulation concepts, explaining our modelling top-level structure, the details about the co-simulation tool integration framework. In order to make the design process of mechatronic devices "First-Time-Right", details about fault modelling and its corresponding controller software fault handling will also be introduced. Section 3, the introduced methods and the tools are demonstrated by using an existing mechatronic device as an example. Finally, conclusions are drawn and directions for future work are indicated in Section 4.

2. Approach

In Section 2.1, the proposed co-modelling concepts and methods are introduced, following by details about the co-simulation tool integration framework in Section 2.2. Fault modelling and fault handling in controller software are presented in Section 2.3.

2.1. Modelling methodology

To avoid unnecessary misunderstanding about commonly used terms in this paper for readers from different background (domains), a short explanation about concepts related with co-modelling is included.

Co-modelling is a heterogeneous modelling approach in which different, domain-specific modelling methodologies are used. It is supported by a co-simulation tool framework which integrates different domain-specific tools. The simulators under the co-simulation framework are connected through a co-simulation engine. The details about the synchronisation schemes among different tools are explained in Section 2.2.

Collaborative modelling is one step of the whole *co-design* process, which means more than one person is working together. Engineers from different domains can perform collaborative modelling, but this process does not necessarily need to be a co-modelling process unless the tools can synchronise with each other. Details about collaborative modelling on a pilot study can be found in [1].

Our proposed *co-modelling* approach is one of the options to perform collaborative modelling. It is considered as less error-prone than those depending purely on human communication, since there, the human factors involved easily introduce unnecessary faults.

2.1.1. System top-level model

In our methodology, a mechatronic device is divided into several top-level components as shown in Fig. 1: *Controller*, *IO* and *Plant*. The `Controller` (DE domain) block represents the control algorithm and/or logic, which ultimately get implemented in a control computer. The `Plant` (CT domain) block models plant dynamics, which involve the relevant physics domains, like electrical, mechanic, pneumatic, hydraulic, and thermal. The data conversion between the controller and the plant (as is needed to connect the two different modelling domains) is modelled in the `IO` block, as it is also there in the real system. I/O components such as A/D, D/A converters, Samplers, Zero-Order Holds are modelled inside this `IO` block as well. The discrete-event parts of the `IO` block (i.e. which will eventually be implemented in the control computer) are modelled in DE domain, while its continuous-time parts are modelled in CT domain. This is indicated in Fig. 1 by the `IO` block having two different background shades.

2.2. Tool integration framework

2.2.1. General concepts

In this paper, we use the concept of co-model and co-simulation to express and execute CPS models [13]. A co-model is a model

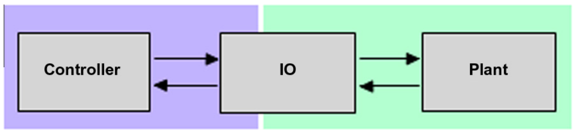


Fig. 1. Top-level structure of the system model.

[1] www.20sim.com.
[2] www.chiastek.com/products/cosimate.html.

which consists of two component models, one formulated in a DE formalism, the other in a CT formalism and one contract. In a co-simulation, the DE and CT models are executed in their own simulators with the steering of a co-simulation engine as shown in Fig. 2. This co-simulation engine handles the exchange of time, parameters and variables at the interface of DE and CT models.

2.2.2. Co-simulation synchronisation scheme

A system model representation is defined as the following [14,15],

$$x(\sigma) = \phi(\tau, \sigma, x(\tau), u) \tag{1}$$

where

- τ: initial time $\tau \in$ time set $\mathcal{T}$;
- σ: final time $\sigma \in \mathcal{T}$ with $\sigma \geqslant \tau$;
- x: state variable in time set $[\tau, \sigma]$;
- u: a function that maps $[\tau, \sigma]$ to control inputs $\mathcal{U}$;
- ϕ: a mapping from the initial state x, the initial time τ, the final time σ and the function u to the value of the state at time σ.

This formulation covers both time-triggered and (discrete)-event triggered cases. Here the time-triggered case means the state of the system changes as time progresses. In other words, when x and u belong to infinite sets, this representation is a time-driven system. In this case, $\mathcal{T} \in \mathbb{R}$ the system is continuous time, while when $\mathcal{T} \in \mathbb{Z}$ the system is discrete time. The (discrete)-event triggered case means the state of the system changes due to the occurrence of an event. When u belongs to a finite/countable set, this representation is an (discrete)-event triggered system, while x still belongs to infinite sets (dynamic behaviour of the system). In this paper, we adopt the notation given above.

For a proper co-simulation framework, state events, being detected and precisely localised in the CT simulator, need to be communicated to the DE simulator. This functionality must be supported by the co-simulation engine as shown in Fig. 2.

There are two options to achieve a synchronisation for this purpose [16]:

- In an optimistic co-simulation, each simulator proceeds at its own pace. If a signal is received from the other simulator, the time at which the event occurred is determined. A problem occurs if the simulator's internal clock has passed the time at which the event occurred. If this happens, a roll back of the simulator has to be performed to put it in the state it was in at the time of the received event. This roll-back mechanism is not available in all simulators.
- In a lock-step co-simulation, all simulators calculate synchronously and at equal time-steps. There is no need for a roll-back mechanism.

We use the lock-step synchronisation scheme in this paper. Fig. 3a illustrates the synchronisation scheme underlying the co-simulation between a DE simulator (top) and a CT simulator (bottom). The DE and CT simulators are coupled through a co-simulation engine that explicitly synchronises the shared variables, events and the simulation time.

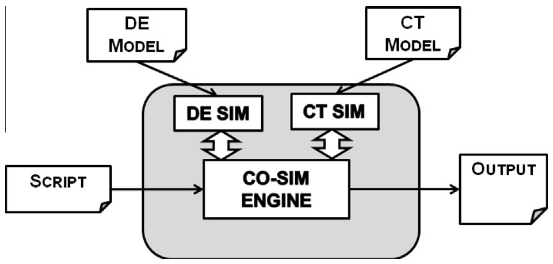


Fig. 2. Co-model and co-simulation.

At initial time $\mathcal{T}_\tau$, the DE simulator has processed all internal zero-time transitions and it wants to move time forward by $\mathcal{T}_i - \mathcal{T}_\tau$, shown as a co-simulation step in Fig. 3a. Instead of actually performing this time step, transfer is given to the CT simulator through the co-simulation engine.

The state of the shared variables in the CT model is updated at $X_{\mathcal{T}_\tau}$ and the CT simulator tries to move forward to $X_{\mathcal{T}_i}$. Assuming that no state events have been triggered in the CT-part during that time step, control is given back to the DE simulator through the co-simulation engine and the shared variables and DE simulation time are updated in $U_{\mathcal{T}_i}$. The DE simulator again processes all internal zero-time transitions until it needs to perform another time step, in this case $\mathcal{T}_\sigma - \mathcal{T}_i$. Fig. 3a illustrates an iterative synchronisation scheme for solving Eq. (1), in the sense that there can be n intermediate co-simulation steps in between.

If one or more state events happen at the CT side in between $\mathcal{T}_\tau$ and $\mathcal{T}_i$ as seen in Fig. 3b, the CT simulator detects and localises that event, and hands over control to the DE simulator, which adjusts its co-simulation step accordingly.

In the situation that the event is a special case, such as so-called even root problem [17], if the integration step is too large, there is a danger of missing both events, see Fig. 3b. CT simulator has the possibility to specify a maximum integration time step in order to avoid this problem.

2.2.3. DESTECS tool

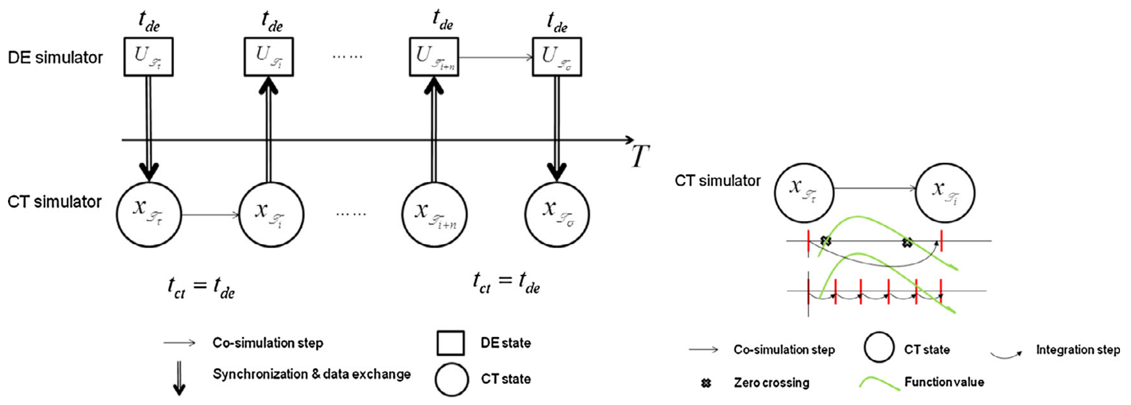
In DESTECS (Design Support and Tooling for Embedded Control Software) project,[3] two domain-specific tools were involved: VDM and 20-sim.

VDM [18] is a formal method that permits description of functionality at a high level of abstraction. It is an object-oriented language. In this paper, we use VDM-RT, a special version of VDM, to specify time in DE models. VDM-RT includes primitives for modelling deployment to a distributed hardware architecture and support for asynchronous communication, supported by the Overture[4] tool.

20-sim is a multi-domain modelling and simulation tool for the dynamic behaviour of physical systems. It supports mixed-mode integration techniques to allow the modelling and simulation of computer-controlled physical systems that contain continuous as well as discrete-time parts. 20-sim supports bond-graph modelling [19,20] which is a port-based approach. Bond-graph submodels can be re-used elegantly, since bond graphs are non-causal. The submodels can be seen as objects as well, with hierarchy, thus bond-graph modelling is a form of object-oriented physical-systems modelling. 20-sim has handy graphic visualisation functionality (diagram based).

The DESTECS tool takes advantages from both VDM and 20-sim: VDM's high level abstraction and 20-sim's intuitive physical-systems dynamics modelling property as well as the object-oriented feature. Using the lock-step synchronisation scheme, the DESTECS tool can handle not only time-triggered cases but also event-triggered cases. These are improvements compared to the tools listed in Section 1. The DESTECS tool was also tested by industry, using industrial case studies [21].

[3] www.destecs.org.
[4] www.overturetool.org.



(a) The whole synchronization scheme (b) CT simulator in one co-simulation step

Fig. 3. Co-simulation synchronisation scheme.

2.3. Fault modelling and corresponding controller software fault handling

As we already know that modelling is an abstraction of the reality, which means that a model is not 'exactly' the same as reality. In order to make the mechatronic device design more compatible with reality, it is useful to model non-ideal cases during the early design phase (modelling phase). In this way, the whole co-design process can be more robust: the controller takes non-ideal situations (faults) into consideration.

In this section, we introduce how to perform fault modelling and how to make controller software compatible with different abstraction levels (with faults and without faults) of the same plant model.

The procedure of fault modelling and designing controller software that handles these faults is as follows:

1. Identify the faults and make a model which can represent the faulty behaviour.
2. Determine the priority of the faults by analysing the consequence of the faults and design the corresponding fault-tolerant software.

2.3.1. Fault modelling

We use the terms *ideal*, *realistic* and *fault* models. The *ideal* model is a model of a component's essential functionality ignoring all parasitic effects, like physical implementation limitations, e.g. spring without mass, and non-idealnesses due to manufacturing tolerances. A realistic model of a component is more faithful to that of a real object and includes the previously ignored parasitic effects. A fault model includes faulty behaviour of the component being modelled. This model mimics the behaviour of the component exhibiting the envisaged fault.

Note that such a modelled fault must be triggered in order to see the effects of that fault model in simulation results as shown in Fig. 4. One can use a scripting language [22] to activate faults, or one can physically inject faults, e.g. physical switches to turn off/on certain components. Here *faulty behaviour* is considered as a deviation from its specified behaviour [23], whilst the presence of this error is the *fault*. *Fault tolerance* means to avoid functionality failures in the presences of faults.

2.3.2. Controller structuring to accommodate fault-tolerance

In this paper, we propose a controller structure (shown in Fig. 5) to elegantly accommodate fault-tolerant add-on code. The functionality of the actual control-law implementation is separated from fault-handling issues. This allows for separate testing, and makes the implementation First-time right. Existing controller architectures with fault-tolerance facilities, in general, are either application specific, or are general structures with ad hoc add-ons for fault tolerant functionality. Examples of the latter are subsumption architectures [25] and middleware software layers [26].

In Fig. 5, the *Measurement and Actuation* block of the embedded control software denotes filtering and scaling to adapt the value ranges. The *Safety layer* block checks all signals going to and coming from the hardware. Safety issues are on all controller levels, shown by its U-shape in the figure. The *Safety layer* deals with the fault handling issues. More detail of the *Safety layer* is given in Fig. 6. The *Loop control* block contains the control algorithms controlling the actuators. The *Sequence control* block is a kind of task level controller: it commands the loop controllers, by computing the setpoints of the control algorithms, and if applicable, enabling them and adapting parameters. The *Supervisory control* block is a strategy controller: calculations, often taking considerable amount of computing time (relative to the sampling period), that instruct the Sequence controller to determine its next task.

Safety-layer parts, shown in Fig. 6, are *Error Detection*, *Safety Controller* and *Decision Maker*. Error detection: use the sensor values and reasoning algorithms to detect what fault has occurred and give a status of the sensors. Different safety controllers are available to take over control depending on the exact situation of the detected fault. Examples are: immediately stop the whole

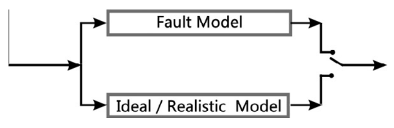


Fig. 4. Fault modelling.

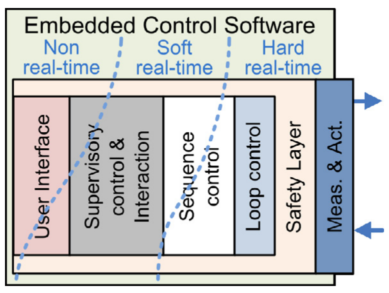


Fig. 5. Controller layered structure [24].

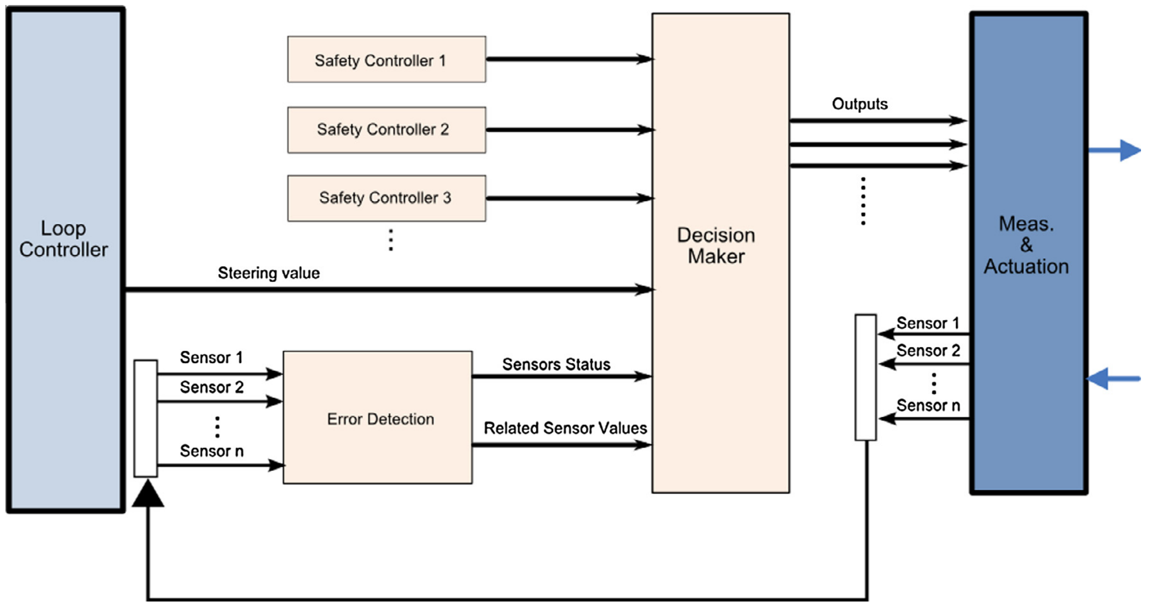
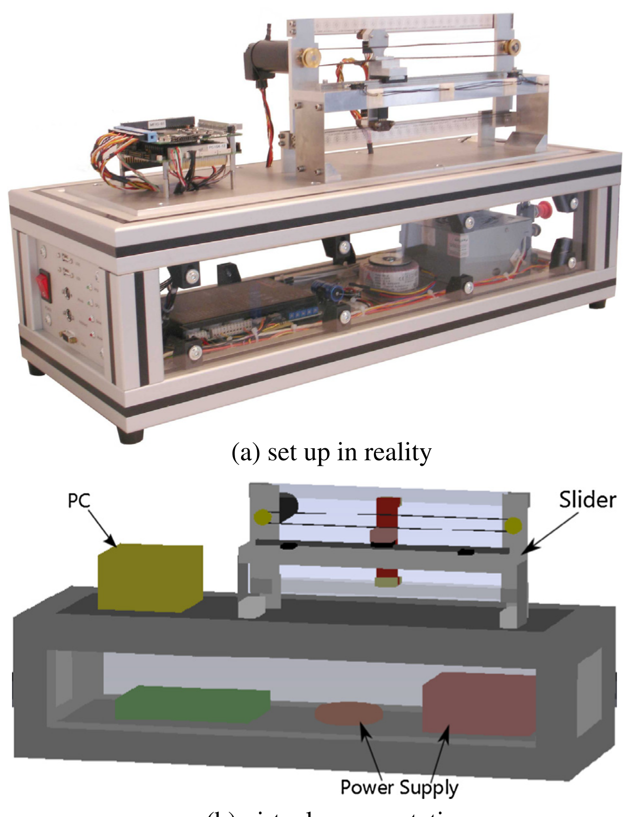


Fig. 6. Safety layer components.

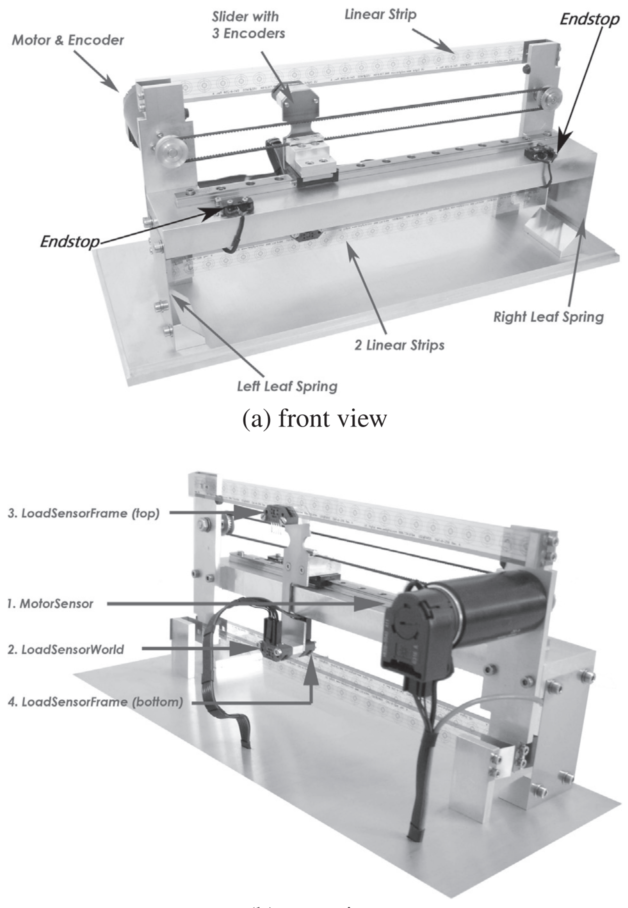
device; limit the outputs; bring the device to a safe state, etc. These strategies are device dependent. Decision maker: based on the severity of the fault, it decides which controller output is passed to the device, so either the steering value from the loop controller in the normal case or the value calculated by the selected appropriate safety controller. All blocks can be computed at any time and decision maker only passes the values computed by the selected controller to the *Meas.& Actuation*. For performance reasons, the safety controller may only be calculated when needed. However, when such a safety controller needs the sensor values, it might be necessary to always calculate this safety controller to ensure a



(a) set up in reality

(b) virtual representation

Fig. 7. The whole set up.



(a) front view

(b) rear view

Fig. 8. The dynamic plant part of the slider.

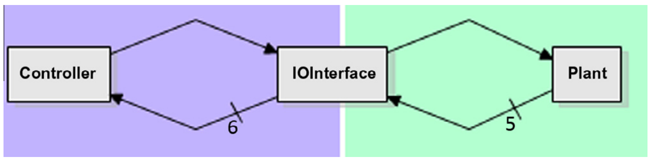


Fig. 9. Top-level of the slider system model.

bumpless transfer of the steering value from the loop controller to the safety controller at the moment of selection.

2.4. Practical use

The proposed co-design flow of a mechatronic device is as follows:

- First, make a co-model which consists of different components from different specific domains. In this paper, it means that designers make the controller software model using VDM. At the same time, the plant model is produced in 20-sim.
- Second, a co-simulation is performed by using the DESTECS tool to assess whether the design of this particular mechatronic device is good or not.
- To validate this design, experiments on final products are needed. For this, it is most elegant that the controller code is automatically generated from the controller software models. Unfortunately, at this moment, code *cannot* be generated from the DESTECS tool. Therefore, we have modelled the controller structure in 20-sim, according to the structure given in Figs. 5 and 6.

3. Case study

In this section, we have chosen a mechatronic device, called *slider*, to demonstrate the approach introduced in the paper. However the approach can also be applied on other setups.

Components	Parameter	Value
Motor (b)	Motor mass	$1e^{-5}kg$
Frame (a)	Mass of frame	$0.8kg$
	Spring constant	$4.4KN/m$
	Damping	$4.4Ns/m$
Slider (d)	Mass	$0.3kg$
Belt (c)	Spring constant	$800N/m$
	Damping	$1Ns/m$
Rail (e)	Viscous friction	$3Ns/m$
	Coulomb friction	$0.5N$

Fig. 11. Mechanical parameter specification for the set up.

3.1. Set-up description

The slider demo set up was designed to demonstrate typical mechatronic systems in practical situations, see Fig. 7. This set up originates from the principle of a printing device: a slider moving back and forth over a rail guide which is controlled by an embedded computer (PC block in Fig. 7). The frame of the system is flexible, which introduces vibrations in the set up when the slider accelerates, see Fig. 7a. The device has six sensors in total: 1 motor position encoder, 1 position sensor with respect to the fixed world and 2 position sensor with respect to the frame (upper and lower), 2 endstops (left and right), as shown in Fig. 8.

3.2. Modelling and implementation

3.2.1. Top-level modelling

By applying the methods mentioned in Section 2.1.1, we produced the top-level model as shown in Fig. 9. The co-model of the slider consists of `Controller` block, a `Plant` block and a `IOInterface` in between. The plant block contains the slider dynamics model. The `Controller` block reads data from sensors to determine the position of the slider to control the motor servo. This suggests 7 variables being exchanged between the plant and

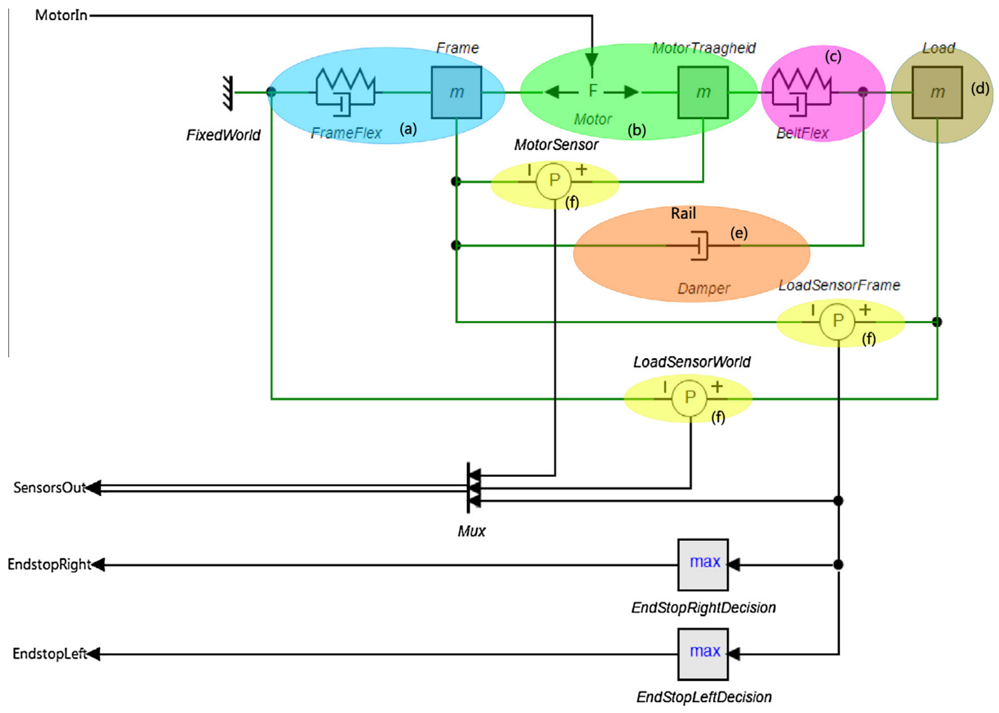


Fig. 10. Plant model.

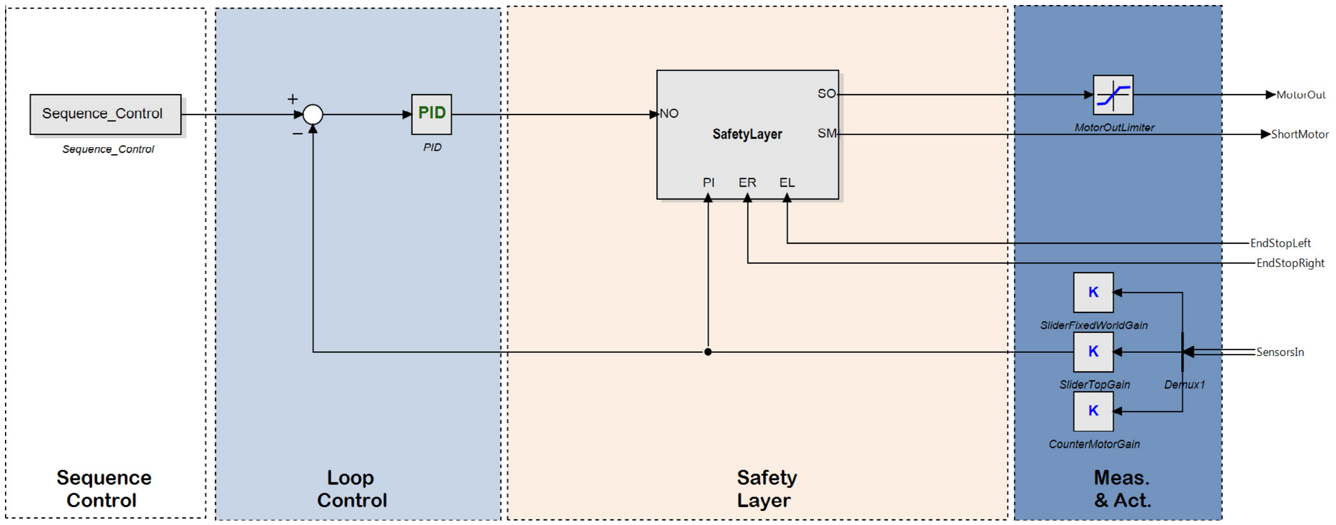


Fig. 12. Controller implementation (layered structure).

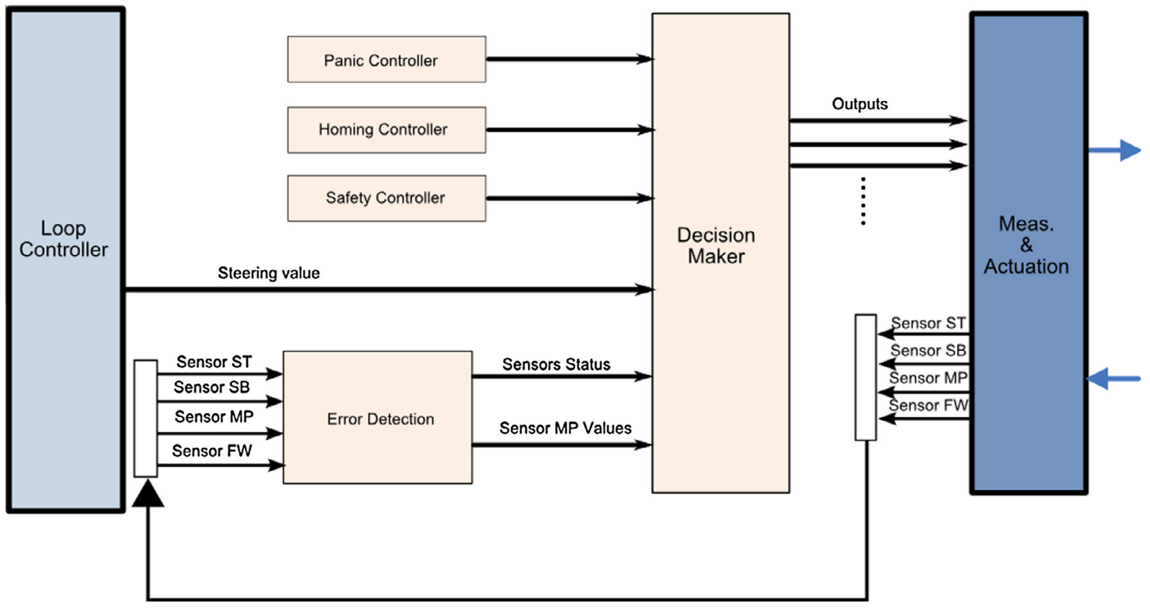


Fig. 13. Safety layer components on the slider set up.

the controller via the `IOInterface`, the shared variables: one motor steering variable and the six sensor values.

3.2.2. Plant modelling

The details of the `Plant` block from Fig. 9 are shown in Fig. 10, with parameters values in Fig. 11. This is a dynamic model which is based on the basic fourth-order mass-spring-mass system. As both the flexibility of the frame and belt is taken into account, this model is a sixth-order model. Including Coulomb friction of the damping phenomena makes this model non-linear. Fig. 10 shows a IPM (Ideal Physical Model) which can be seen as a domain-specific bond graph. In this model, the top and bottom position sensors are considered one, because the slider stays upright during movement, thus these sensors read the same value all the time. Hence in Fig. 9, the signal connection from `Plant` to `IOInterface` has a multiplication of 5. The signal connection from `IOInterface` to `Controller` has a multiplication of 6, since here, all four encoders are considered (next to the two end switches). The

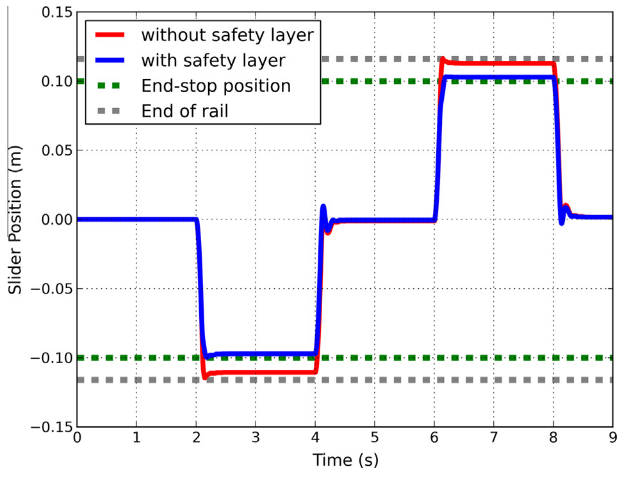


Fig. 14. Slider position comparison with safety layer and without safety layer (measurements from the set up).

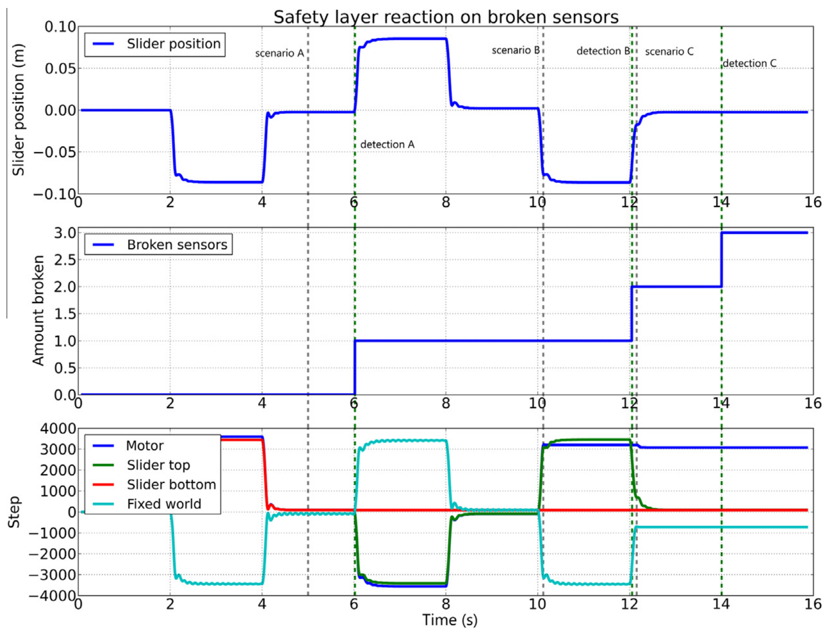


Fig. 15. Slider position together with broken sensors fault scenarios. scenario A: sensor SB broken; detection A: detection of fault A; scenario B: sensor ST broken; detection B: detection of fault B; scenario C: sensor FW broken; detection C: detection of fault C.

coloured ellipses labelled with letters indicate the components as listed in Fig. 11.

3.3. Fault modelling and fault tolerance mechanisms

3.3.1. Fault modelling and detection

Scenarios of complete failure of sensor were investigated for this set up. There are four sensors considered: 1 motor position encoder, 1 position sensor with respect to the fixed world and 2 position sensors with respect to the frame (upper/lower sensor). So using the matrix as given below:

$$\text{ComparisonMatrix} = \begin{bmatrix} |FX-FX| & |FX-SB| & |FX-ST| & |FX-MP| \\ |SB-FX| & |SB-SB| & |SB-ST| & |SB-MP| \\ |ST-FX| & |ST-SB| & |ST-ST| & |ST-MP| \\ |MP-FX| & |MP-SB| & |MP-ST| & |MP-MP| \end{bmatrix} \quad (2)$$

where FX, SB, ST and MP denote the sensor values from FixedWorld, SliderBottom, SliderTop and MotorPosition accordingly. So if the FixedWorld sensor was broken, the difference comparing to the other sensors would be out of the normal range: e.g. $|SB-FX|$, $|FX-ST|$ and $|FX-MP|$ are all out of range. So, we can deduce that the FixedWorld sensor is broken. In this particular set up, a switch, disabling the sensor's power, was made for each of the sensors to mimic the sensor failure: when a sensor loses power, then that sensor stops functioning, so applying the switch means injecting a fault into the system.

3.3.2. Fault tolerance mechanisms

Based on the method introduced earlier in Section 2.3, we can structure the controller software of the slider as in Figs. 12 and 13. In detail, Fig. 12 shows a layered structure of the overall controller software. Fig. 13 shows the components of the slider safety layer: *Error Detection*, *Panic Controller*, *Homing Controller*, *Safety Controller* and *Decision Maker*. Error detection: identifying how many sensors are broken, by using the algorithm mentioned above. Different safety controllers: panic controller (the slider stops moving immediately); homing controller (move the slider to the safe area, the middle of the frame, in this case); safety controller (limit the motor output, reduce the motor speed). Decision maker: based on the sensors status, the decision maker decides which controller to enable.

3.4. Analysis

In Figs. 14 and 15, it is shown that with the aid of the used safety layer in the controller software, the system can react more robust when faults are injected into the system. As indicated in Fig. 14, in between the green dashed lines (±10 cm) (the origin point is the middle of the rail) are the safe range that the setup can operate. The slider can move in between the endstops with the aid of the safety layer. In Fig. 15, the grey dotted vertical line indicates when a sensor error is injected while the green dotted vertical line indicates when the software safety layer detects the broken sensor. It can be seen that even when the system has faults injected, it can still operate normally: in this case, the slider still moves within the safety zone (in between the endstops). In this case, the *safety controller* and *homing controller* (activated when two sensors are broken) from Fig. 13 were activated sequentially.

Note that due to the fact that the DESTECS tool has not yet facilities to generate code from the VDM controller models, we have specified the controller in 20-sim, according to the structure give in Fig. 12. Using the automatic code generation feature of 20-sim and the deployment and experiment tool 20-sim 4C,[5] we have conducted experiments on the real system, of which the key result is shown in Figs. 14 and 15.

4. Conclusions and future work

In this paper, we have first listed the current challenges (incompatibilities) during the co-design process of mechatronic devices. Later, our co-modelling approach was introduced both from methodology and tool aspect: domain-specific modelling (top-level system structure) and co-simulation tool (co-simulation

[5] http://www.20sim.com/products/20sim4c.html.

synchronisation scheme). Furthermore, the fault modelling and fault tolerance mechanism of the controller software were introduced. This allows for designing and testing controller software that can handle faults, i.e. more realistic cases.

In this way, the whole system design is more compatible with the real situation: taking the non-ideal cases into account. Hence the incompatibilities during the whole co-design process were addressed by deploying our methods and tooling.

Looking to future work, constructing the code generation facility of the co-simulation tool is useful for the mechatronic devices designers. More fault patterns and thorough testing is of interest as well.

Acknowledgements

The research leading to these results has received funding from the European Community's Seventh Framework Programme (FP7/2007–2013) under Grant Agreement No. 248134 (project DESTECS). Thanks to Bob van de Vijver for his effort on conducting the experiments on the setup.

References

[1] K. Pierce, C. Gamble, Y. Ni, J.F. Broenink, Collaborative modelling and co-simulation with destecs: a pilot study, in: WETICE, 2012, pp. 280–285.
[2] L. Gheoghe, Continuous/Discrete Co-simulation Interfaces from Formalization to Implementation, Ph.D. thesis, August 2009.
[3] P. Fritzson, V. Engelson, Modelica – a unified object-oriented language for system modelling and simulation, in: ECCOP '98: Proceedings of the 12th European Conference on Object-Oriented Programming, Springer-Verlag, 1998, pp. 67–90.
[4] Modelisar. Functional Mock-up Interface for Model Exchange 1.0., Tech. Rep., 2010.
[5] W. Chen, M. Huhn, P. Fritzson, A generic FMU interface for modelica, in: F.E. Cellier, D. Broman, P. Fritzson, E.A. Lee (Eds.), EOOLT, Linkping Electronic Conference Proceedings, vol. 56, Linkping University Electronic Press, 2011, pp. 19–24.
[6] J. Davis, R. Galicia, M. Goel, C. Hylands, E. Lee, J. Liu, X. Liu, L. Muliadi, S. Neuendorffer, J. Reekie, N. Smyth, J. Tsay, Y. Xiong, Ptolemy-II: Heterogeneous Concurrent Modeling and Design in Java, Technical Memorandum UCB/ERL No. M99/40, University of California at Berkeley, July 1999.
[7] C.Verhaar, An Integrated Embedded Control Software Design Case Study using Ptolemy II, Msc thesis, Control Laboratory, University of Twente, May 2008.
[8] S. Wolff, K. Pierce, P. Derler, Multi-domain Modelling in Destecs and Ptolemy – A Tool Comparison, Technical Report ece-tr-15, Aarhus University, Department of Engineering, 2012.
[9] D. Jovanović, Designing Dependable Process-oriented Software, A CSP Approach, PhD thesis, University of Twente, Enschede, The Netherlands, 2006.
[10] A. Roscoe, C. Hoare, R. Bird, The Theory and Practice of Concurrency, Prentice Hall PTR, Upper Saddle River, NJ, USA, 1997.
[11] M. ten Berge, B. Orlic, J. Broenink, Co-simulation of networked embedded control systems, a CSP-like process-oriented approach, in: Proc. IEEE International Symposium on Computer Aided Control Systems Conference, Munich, Germany, 2006, pp. 434–439.
[12] M. Groothuis, A. Damstra, J. Broenink, Virtual prototyping through co-simulation of a cartesian plotter, in: IEEE International Conference on Emerging Technologies and Factory Automation, 2008. ETFA 2008. No. 08HT8968C, IEEE Industrial Electronics Society, 2008, pp. 697–700.
[13] J.F. Broenink, P.G. Larsen, M. Verhoef, C. Kleijn, D.S. Jovanović, K.G. Pierce, F. Wouters, Design support and tooling for dependable embedded control systems, in: Proceedings of SERENE '10, ACM, ACM Sigsoft, 2010, pp. 77–82.
[14] E. Sontag, Mathematical Control Theory: Deterministic Finite Dimensional Systems, Texts in Applied Mathematics, Springer-Verlag, 1990.
[15] B. Schutter, W. Heemels, J. Lunze, C. Prieur, Survey of Modeling, Analysis, and Control of Hybrid Systems, Handbook of Hybrid Systems Control – Theory, Tools, Applications, Cambridge University Press, 2009. Chapter 2, pp. 33–55.
[16] J.F. Broenink, Y. Ni, M.A. Groothuis, On model-driven design of robot software using co-simulation, in: E. Menegatti (Ed.), SIMPAR, Workshop on Simulation Technologies in the Robot Development Process, 2010.
[17] F. Zhang, M. Yeddanapudi, P. Mosterman, Zero-crossing location and detection algorithms for hybrid system simulation, in: Proceedings of the 17th IFAC World Congress, vol. 17, 2008.
[18] J. Fitzgerald, P.G. Larsen, P. Mukherjee, N. Plat, M. Verhoef, Validated Designs for Object–oriented Systems, Springer, New York, 2005.
[19] H.M. Paynter, Analysis and Design of Engineering Systems, MIT Press, Cambridge, MA, 1961.
[20] P.C. Breedveld, Multibond-graph elements in physical systems theory, J. Franklin Inst. 319 (1/2) (1985) 1–36. http://dx.doi.org/10.1016/0016-0032(85)90062-6.
[21] M. Verhoef, B. Bos, P. van Eijk, J. Remijnse, E. Visser, M. dePaepe, Y. deWitte, K.Rombaut, R.van Lembergen, D4.3—Industrial Case Studies, Final Report (Public), The DESTECS Project (CNECT-ICT-248134), 2012. <http://www.destecs.org/>.
[22] Y. Ni, J.F. Broenink, Hybrid systems modelling and simulation in destecs: a co-simulation approach, in: M. Klumpp (Ed.), 26th European Simulation and Modelling Conference, ESM 2012, Essen, Germany, EUROSIS-ETI, Ghent, Belgium, 2012, pp. 32–36.
[23] A. Avižienis, J.-C. Laprie, B. Randell, C. Landwehr, Basic concepts and taxonomy of dependable and secure computing, IEEE Trans. Dependable Sec. Comput. 1 (2004) 11–33. http://dx.doi.org/http://doi.ieeecomputersociety.org/10.1109/TDSC.2004.2, doi:http://doi.ieeecomputersociety.org/10.1109/TDSC.2004.2..
[24] J.F. Broenink, Y. Ni, Model-driven robot-software design using integrated models and co-simulation, in: J. McAllister, S. Bhattacharyya (Eds.), International Conference on Embedded Computer Systems, SAMOS 2012, Samos, Greece, IEEE Computer Society, USA, 2012, pp. 339–344.
[25] R. Brooks, A robust layered control system for a mobile robot, IEEE J. Robot. Autom. 2 (1) (1986) 14–23. http://dx.doi.org/10.1109/JRA.1986.1087032.
[26] H. Bruyninckx, Open robot control software: the orocos project, in: IEEE International Conference on Robotics and Automation, 2001. Proceedings 2001 ICRA, vol. 3, 2001, pp. 2523–2528. http://dx.doi.org/10.1109/ROBOT.2001.933002.