Transiting from Crescendo to INTO-CPS

Kenneth Lausdahl\textsuperscript{1}, Kim Bjerger\textsuperscript{1}, Tom Bokhove\textsuperscript{2}, Frank Groen\textsuperscript{2}, and Peter Gorm Larsen\textsuperscript{1}

\textsuperscript{1} Aarhus University, Denmark
\textsuperscript{2} Controllab Products, Netherlands

Abstract. In the development of Cyber-Physical Systems, interoperability between tools supporting heterogeneous models is essential. In the Crescendo technology, a direct connection between simulation in Overture and solving in 20-sim was established. In the Integrated Tool Chain for Model-based Design of Cyber-Physical Systems (INTO-CPS) project, this connection was generalized to include any number of constituent components. The connections between the different simulations were established using the Functional Mock-up Interface. This paper explores the portability of a model of a trolley conveyor system from a Crescendo setting to the INTO-CPS technology.

1 Introduction

Whenever it is desirable to combine models described in different formalisms that do not share the same semantic foundations, it is worthwhile to consider technologies that enable a collaborative simulation of the different constituent models. Such simulations are called co-simulations [13]. In the Design Support and Tooling for Embedded Control Software (DESTECS) project [4], such a solution, called Crescendo [12], was enabled by coupling the Discrete Event (DE) formalism VDM, supported by the Overture tool [16], with the Continuous-Time (CT) formalism bond graphs, which is supported by the 20-sim tool [15]. This work was subsequently generalised in the INTO-CPS project to any number of DE and CT models using the Functional Mock-up Interface (FMI) to connect the constituent models, represented as Functional Mock-up Units (FMUs) [9,10,17,19].

The conversion from a collaborative model produced using Crescendo to a corresponding INTO-CPS solution has already been reported [22]. However, the case study we present in this paper is novel in the sense that it also uses event-driven control, which are not supported by the FMI standard. The event-driven control which takes place when something happens oppose to time-driven control which occurs as predefined time intervals. In addition, this work also attempts to turn the 3D animations made inside 20-sim into a special 3D FMU so that it is possible to demonstrate system model behaviour in a user-friendly manner.

The case study used is a trolley conveyor system in which the trolley can be tilted. This functionality is used, for example, to move parcels on the trolleys to their desired destination. The case study originates in a collaboration with BEUMER Group, who deliver such conveyor systems, for example, for baggage handling in airports all over the world. In this work we are interested in tilting the trolleys as they pass around a
bend in the conveyor in order to compensate for the centrifugal force generated at high travel speeds.

This paper is structured as follows. Section 2 introduces the different co-simulation technologies used in this work. Section 3 introduces the case study used here. This is followed in Section 4 by an explanation of the transition from the Crescendo technology to the INTO-CPS setting. Finally, Section 5 explains the transition of the 3D animation from 20-sim to Unity, while Section 6 provides a number of concluding remarks.

2 The Co-simulation Technologies used

The technology initially used to realise the case study was the Crescendo Tool. It combined two tools, Overture and 20-sim, allowing co-simulation of DE and CT models. The INTO-CPS technology – the target of this paper – allows for not only two, but $n$ models of any combination of DE and CT to be simulated. The project is focused on the FMI standard.

2.1 Crescendo

The Crescendo tool is the outcome of the European Framework 7 project DESTECs, (www.destecs.org) which ran from early 2010 until 2013 [4, 11]. Its aim was to develop methods and tools that combine continuous time models of systems (in tools such as 20-sim) with discrete event controller models in the Vienna Development Method (VDM) through co-simulation [25, 23]. The approach was intended to encourage collaborative multidisciplinary modelling, including modelling of faults and fault tolerance mechanisms. The Crescendo tool, in combination with Overture and 20-sim, provides a platform for co-simulation. The platform only supports two simulators using a variable step size algorithm to progress simulation time. The intention in the project was to have one discrete controller, which typically did not support roll-back, and allow the plant model to be in charge of providing future synchronization time intervals. In Crescendo, the controllers were modelled in VDM using the Real-Time dialect, which added timing information to the execution of each expression and statement [24]. The physical plant model was intended to be modelled using 20-sim in continuous time using block diagrams or bond graphs. The tool shown in Figure 1 provides a single user interface based on Eclipse for VDM modelling and co-simulation, and good integration with 20-sim. The co-simulation is defined based on a contract which consists of shared design parameters used for sharing constant parameters between models; monitored variables used for providing inputs to the controller model; controlled variables which are controller outputs and events which are used like interrupts in the controller.

2.2 The Functional Mock-up Interface

The tool-independent standard FMI was developed within the MODELISAR project [14]. It supports both model exchange and co-simulation and exists in two versions. Version 1.0, released in 2010 and Version 2.0, released in 2014. It was developed to improve
exchange of simulation models between suppliers and Original Equipment Manufacturers (OEM). The standard describes how simulation units are to be exchanged as ZIP archives called Functional Mock-up Unit (FMU)s and how the model interface is described in an XML file named `modelDescription.xml`. The functional interface of the model is described as a number of C functions that must be exported by the library that implements the model inside the FMU. Since the FMU only contains a binary implementation of the model, it offers some level of intellectual property protection. The standard lists a number of constraints on how the C functions in FMI can be called without an explicit co-simulation algorithm.

2.3 The INTO-CPS Technology

The vision of the INTO-CPS\(^3\) consortium is that Cyber-Physical Systems (CPS) engineers should be able to deploy a wide range of tools to support model-based design and analysis, rather than relying on a single factotum. The goal of the project is to develop an integrated “tool chain” that supports multidisciplinary, collaborative modelling of CPSs from requirements, through design, to realisation in hardware and software, enabling traceability through the development. The project integrates existing industry-strength baseline tools in their application domains, based centrally around FMI-compliant co-simulation [3]. The project focuses on the pragmatic integration of these tools, allowing for extensions in areas where a need is recognised. The tool chain is underpinned by well-founded semantic foundations that ensures the results of analysis can be trusted [26,7].

\(^3\) See [http://into-cps.au.dk/](http://into-cps.au.dk/).
The tool chain provides powerful analysis techniques for CPS models, including generation and static checking of FMI interfaces; model checking; Hardware-in-the-Loop (HiL) and Software-in-the-Loop (SiL) simulation, supported by code generation. This enables both Test Automation (TA) and Design Space Exploration (DSE) of CPSs.

The INTO-CPS project provides a lightweight application interface for managing the development of constituent system models for co-simulations and for configuring and running such co-simulations. The project offers the following FMI-enabled modelling tools: Overture, 20-sim, OpenModelica and RT Tester. However, it is an open tool chain, so in general any FMI 2.0 enabled tool can be used.

The project also provides high-level system design facilities using the Systems Modelling Language (SysML) through the Modelio tool [20]. A co-simulation configuration can be modelled in SysML using block diagrams and a custom INTO-CPS profile for FMI. Co-simulation in INTO-CPS conforms to FMI version 2.0 and the master algorithm supports both fixed and variable step sizes. The variable step size algorithm supports constraints to adjust the step size. There are four types of constraints that can be utilized:

**Zero Crossing**: A zero crossing constraint is a continuous constraint. A zero crossing occurs at the point where a function changes its sign. In simulation, it can be important to adjust the step size such that a zero crossing is revealed as accurately as possible. For instance, a ball should rebound from a wall exactly when the distance between the ball and the wall hits zero and not before or after. A solver in a tool such as Simulink can adjust the step size using iterative approaches, but in a co-simulation a roll-back of the participating models’ internal states would be required, but is in general not possible or efficient. Hence, the variable step size calculator bases its step size adjustments on the prediction of a future zero crossing. It uses extrapolation and derivative estimation to estimate changes and therefore reduce the need for roll-back.

**Bounded Difference**: A bounded difference constraint is also a continuous constraint. A bounded difference ensures that the minimal and maximal value of a set of values do not differ by more than a specified amount (the underlying assumption is that this difference becomes smaller when the step size is reduced). The bounded difference problem is distinct from the zero-crossing problem in that there is not a specific time instant (the zero crossing) to hit, but rather a specific time difference (the step size that keeps the difference bounded).

**Sampling Rate**: A sampling rate constraint is a discrete constraint. It constrains the step size such that repetitive, predefined time instants are exactly hit. This can be useful in co-simulation, for instance, when a modelled control unit reads a sensor value every x milliseconds.

**FMU Max Step Size**: This constraint implements the `getMaxStepSize` method from [6,8] providing a good prediction of the maximal step size that a given FMU instance can perform at a given point in time. It limits the need to roll back a simulation at a given point. It is enabled by default and it constrains the step size as follows:

$$size = \min\{\{(getMaxStepSize(i)) \mid \forall i \in \text{instances}\}\}$$
These constrains can be used to improve the overall co-simulation result by reducing the step size at important time points during a simulation. This can be illustrated by a small example of a water tank which has a constant inflow and a drain valve where the aim is to keep the water level between a minimal and maximal marker as shown in Figure 2.

![Water tank with min and max marker](image)

If the controller is implemented as a periodic check then the constraint *FMU Max Step Size* can be added to make sure that the co-simulation does not perform steps larger than the period of the controller. However, if the controller is implemented at event-driven control then the *Zero Crossing* constraint will result in a faster co-simulation while retaining the same simulation precession since the controller only deals with points in time where the level is about to cross the mim and max markers. Defining a zero crossing constraint between *(level and max)* and *(level and min)* will result in a reduction of the step size when approaching the zero crossing and making sure that a step is taken as close to the crossing as possible.

**The Overture FMU Extension**  The Overture tool is extended with an FMU exporter that conforms to FMI 2.0. The exporter relies on a VDM FMI library and modelling guideline. The library consists of a *Port* class and the following specialisations: *BoolPort, IntPort, RealPort, and StringPort*, each mapping to the corresponding scalar variable type in FMI. These all have *getValue* and *setValue* methods. Integration with FMI is modelled using a special *HardwareInterface* class where parameters, inputs and outputs are modelled using this port hierarchy. Through the explicit modelling of FMI scalar variables as ports, it is possible to generate FMUs without requiring additional input. A simple mapping can be made from *shared design variables: sdp* into parameters, *controlled* into input on the plant side and output on the controller.

The tool requires the user to follow a design pattern where an instance of *HardwareInterface* is declared in the *System* class, and a *World* class is responsible for blocking the initial thread to ensure infinite execution. This class must contain all
ports and custom annotations that set the type and name of the exported port. Listing 1.1 shows a small example of three ports exported as parameter, input and output, respectively.

```java
class HardwareInterface
values
  -- @ interface: type = parameter;
  public v : RealPort = new RealPort(1.0);
instance variables
  -- @ interface: type = input;
  public distanceTravelled : RealPort := new RealPort(0.0);
  -- @ interface: type = output;
  public setAngle : RealPort := new RealPort(0.0);
end HardwareInterface
```

Listing 1.1: The HardwareInterface class

While these all map with the same semantics as in Crescendo, this does not apply for events, which are not supported in INTO-CPS.

**20-sim FMU Extension** The 20-sim tool is extended with FMI support through a custom code generator template\(^4\) that is able to export a module in a 20-sim model as an FMU. This solution does not support event detection and external library usage (Dynamic-Link Libraries, or DLLs), which could be used, for example, for collision detection using a physics library. It has support for the following integration methods: Discrete Time, Euler, Runge Kutta 2, Runge Kutta 4, CVODE\(^5\), and MeBDF\(^6\). A tool wrapper is also available which does not have these limitations and which thus works much like in Crescendo, where the full potential of the simulator can be used during simulation.

### 3 Case Study

The trolley conveyor case study aims to investigate the behaviour of different tilt strategies when a parcel follows a specific track. The model describes the environment for the path of the conveyor system, which is composed of both straight and curved segments, as illustrated in Figure 3. On this conveyor, parcels are transported on a trolley at a constant speed. In the bends of the conveyor, the trolley is tilted to ensure that parcels are kept in the same position on the trolley [2]. Actuators on the trolley control the tilt to

---

\(^4\) The FMU generator template for 20-sim is available at [https://github.com/controllab/fmi-export-20sim/](https://github.com/controllab/fmi-export-20sim/).

\(^5\) [https://computation.llnl.gov/projects/sundials/cvode](https://computation.llnl.gov/projects/sundials/cvode)

\(^6\) [http://www.netlib.org/ode/mebdfi.f](http://www.netlib.org/ode/mebdfi.f)
the desired position such that the centrifugal force experienced by the parcel is neutral-ized. Only one trolley is modelled, with the focus being on transporting a single parcel. The CT model is composed of a number of blocks modelling the environment, which includes the conveyor path, trolley and parcel. These modelling blocks are described briefly in this section.

A CT model consists of components connected by power and signal ports. Power ports are bidirectional and represent energy flow, while signal ports are unidirectional, from one component to another, and carry information. In the SysML internal block diagram these signals are described with arrows on edges between the component blocks. The $\text{SetAngle}$ in Figure 4 is such a signal that simulates the current desired tilt angle. A power port is always characterized by two domain independent variables [1]. The product of these variables has the dimension of power, measured in Watts. Therefore, they are called power conjugated variables. In the mechanical domain the variables are force and velocity, for the electrical domain the variables are voltage and current. In 20-sim diagrams, like those we use in this CT model, a power port is a port where power can be exchanged between a component and its environment in terms of these.

The purpose of the DE model is to control the motion of the tilting trolley. Exploring different motion curves for the tilting trolley device has been the primary purpose of this part of the model. In the DESTECS project, the DE model was linked to the CT model in terms of the co-model, interface as described in the DESTECS methodological guidelines [5].

4 Transitioning to INTO-CPS

To perform a translation from a Crescendo model into an INTO-CPS co-simulation project, both simulation tools must be capable of exporting FMI 2 compliant FMUs as described in section 2.3. The original model of the Conveyor-Path (CP) uses events that are not supported in INTO-CPS. The Crescendo co-simulation contract for the CP
model is shown in Listing 1.2. It describes the interface between the DE controller and the CT plant model. The first section of the contract lists Shared Design Parameters (SDP), that is, shared constants that are made available to both simulators. Controlled variables set by the controller become monitored variables that are observed by the controller. Finally, events generated by the CT model and handled by the DE model.

```plaintext
sdp real v;
sdp real r2;
sdp real r4;
sdp real l1;
sdp real l3;
sdp real trayPitch;
sdp real p;
controlled real setAngle := 0.0;
monitored real distanceTravelled := 0.0;
monitored real distCTB1 := 0.0;
monitored real distCTB2 := 0.0;
monitored real distCTB3 := 0.0;
monitored real distCTB4 := 0.0;
```

Listing 1.2: Crescendo contract

The first step in a transition to INTO-CPS is to convert the contract to establish the interface for the co-models for the controller and plant. The controlled variables from Crescendo can be mapped directly to output ports, and any monitored variables can be mapped directly to input ports. The SDP can be mapped to properties of both the controller and plant block. FMI does not have a notion of SDP, but individual FMUs can have parameters. Once block creation is completed, the connections are made and this mapping preserves the same overall co-simulation semantics as in Crescendo. In Figure 4 the SysML connection diagram of the CP model is shown. Note that events are also converted to ports and connections.

The diagram is made in the Modelio tool using the INTO-CPS profile for SysML. This tool can export an initial model description for each FMU containing the required scalar variables (a.k.a. ports). It can also export the connection information in a format supported by the INTO-CPS Application, which is the interface used for performing co-simulations.

4.1 Updating the DE Controller model

The model description exported from Modelio is imported into Overture, resulting in the HardwareInterface class shown in Listing 1.3. It lists all the ports used by
Transitioing from Crescendo to INTO-CPS

the controller. The remainder of the VDM model must also be updated to use the ports from the new FMI library instead of the local instance variables used in Crescendo.

The events cannot be directly mapped because FMI does not have a corresponding notion. The events are not essential for the controller in the CP model, as they mainly serve a logging purpose. It is however still possible to achieve the same effect using a real port which is set to change its value at the same time as the events in the original model when used in combination with the variable step algorithm. The variable step algorithm can be configured with constraints if needed to make sure that a step is performed on every value change.

```java
class HardwareInterface

values
-- @ interface: type = parameter;
public v : RealPort = new RealPort(2.0); -- [m/s]
-- @ interface: type = parameter;
public r_max : RealPort = new RealPort(5.0); -- [m] maximum radius
-- @ interface: type = parameter;
public r1 : RealPort = new RealPort(3.0); -- [m] radius first curve
-- @ interface: type = parameter;
public r2 : RealPort = new RealPort(3.0); -- [m] radius second curve
-- @ interface: type = parameter;
public r3 : RealPort = new RealPort(4.0); -- [m] radius second curve
-- @ interface: type = parameter;
public r4 : RealPort = new RealPort(4.0); -- [m] radius second curve
-- @ interface: type = parameter;
public l1 : RealPort = new RealPort(2.0); -- [m] length of first straight line
-- @ interface: type = parameter;
public l2 : RealPort = new RealPort(0.5); -- [m] length of straight line
-- between curves
-- @ interface: type = parameter;
public l3 : RealPort = new RealPort(0.5); -- [m] length of straight line
-- between curves
-- @ interface: type = parameter;
public trayPitch : RealPort = new RealPort(0.9); -- [m]
-- @ interface: type = parameter;
public p : RealPort = new RealPort(0.5); -- Percentages accelerating

instance variables
-- @ interface: type = input;
public distanceTravelled : RealPort = new RealPort(0.0); -- Distance since start
-- @ interface: type = input;
public distCTB1 : RealPort = new RealPort(0.0); -- Position of CTB1
-- @ interface: type = input;
public distCTB2 : RealPort = new RealPort(0.0); -- Position of CTB2
-- @ interface: type = input;
public distCTB3 : RealPort = new RealPort(0.0); -- Position of CTB3
-- @ interface: type = input;
public distCTB4 : RealPort = new RealPort(0.0); -- Position of CTB4
-- @ interface: type = input;
public switchNumber : RealPort = new RealPort(0.0);
```

Fig. 4: SysML connection diagram for the CP model.
4.2 Updating the CT plant model

There were two options that could be used for updating the 20-sim model: 1) Use the standalone FMU exporter with an updated version of the model, 2) Use a tool wrapper FMU that uses 20-sim for simulation without requiring model changes, except for handling events.

The first option is already supported, since 20-sim has been updated to support FMI using code generation, similar to how 20-sim can generate code for deployment. The result is a stand-alone FMU that does not require 20-sim during co-simulation. The stand-alone FMU can be generated from any submodel. The case study was then changed by replacing the block that encoded the shared DESTECS variables with a block that was created by importing the interface description from Modelio into 20-sim to ensure that the interface matched the overall design.

The second option did not require changes to the model, since a tool wrapper FMU support feature would interact with the model in its current state. To enable this, 20-sim was in fact updated with support for tool wrapper FMUs. It uses the same import and export variables as the original Crescendo model, which means that this model can be used directly without modification for the INTO-CPS co-simulation. The generated tool wrapper FMU lists the import and export variables as input and output scalar variables. The original 20-sim model is also stored inside the FMU. This model is extracted when the co-simulation is initialized, and the FMU tool wrapper then starts 20-sim and requests it to open the 20-sim model from the tool wrapper FMU. If the model opens successfully, the tool wrapper communicates with the 20-sim instance using the same interface as the Crescendo tool, thus the tool wrapper FMU acts as a proxy between INTO-CPS and the 20-sim model.

The model has further been improved by using an external collision detection package to improve the detection of collisions between the parcel and the tray. Because the collision detection package uses an external Dynamic-Link Library (DLL), the first option of using a stand-alone FMU through code generation could not be used. The tool wrapper FMU export was the only option, then, for this model. Furthermore, events were replaced by a `switchNumber` variable that is set to the segment that is approaching at the same moment that the event was triggered in the Crescendo co-simulation. The event handlers in the DE model can be triggered by casing on the `switchNumber` variable. It should be noted that this event handling is not critical to the controller’s ability to perform correct control, but is mainly used for user feedback. If these events were critical they should be constrained by e.g. a zero crossing constraint.
4.3 The INTO-CPS Co-Simulation

The INTO-CPS co-simulation has been carried out using the controller FMU exported from Overture and the tool wrapper plant FMU exported from 20-sim. The simulation was configured in the INTO-CPS Application based on the SysML connection diagram of the connection definitions. The variable step size algorithm was used with the FMU Max Step Size constraint enabled. This algorithm configuration is nearly identical to the one used in Crescendo. In Figure 5a the desired bank angle from the controller is plotted from a simulation performed both in Crescendo and INTO-CPS. It shows that the behaviour of the controller is the same but with a small offset. The model itself behaves correctly in both simulations and the shape of the bank angle is also as expected.

The initial offset, illustrated in Figure 5b, seems to be caused by the plant model integration method not being able to precisely match the desired point in time (multiple of 0.02 seconds) during the first two steps of the simulation. It can be seen that the INTO-CPS simulation always keeps a span of 0.02 seconds between control changes, while the first two steps in Crescendo do not match the period of the controller because of this offset. It has not been possible to determine why the Crescendo simulation has these deviations. It could be that the newer version of the tools have improved the ability to hit the precise point in time, or that it is related to model enhancements that come with a new collision detection library. The Crescendo simulation was performed with Crescendo version 2.0.0 (build GF1B8817) and 20sim version 4.4.1 (build 4356). The INTO-CPS simulation was performed with COE version 0.2.6, Overture FMI Exporter version 0.2.8 and 20sim 4.6.4 (build 8007).

5 Moving the Animation into Unity

As opposed to most FMUs, an animation FMU does not add any simulation data to a co-simulation. However, its purpose lies in giving another view of the data generated by the other components of the co-simulation. It becomes possible to see the behaviour of a co-simulation in a different way than simply observing signal graphs. This is useful for physical systems like robots or the trolley conveyor system, because it becomes
possible to have direct visual feedback of the actual physical system simulated in the co-simulation.

5.1 Providing 3D animation for INTO-CPS simulations

During the DESTECS project, 20-sim was used for 3D visualization of the models during simulation. To bring the same level of visualization to the INTO-CPS project, a custom FMU was developed based on the 3D animation feature of 20-sim. This enabled a generic solution for 3D animations in any co-simulation in INTO-CPS. This solution provides the same feature as in DESTECS. Although Unity is a cross-platform animation framework, the link to 20-sim currently makes it impossible to provide, in a 3D animation FMU, the platform independence that FMI has the ability to support, since 20-sim only supports Microsoft Windows platforms.

5.2 Converting the Animation into Unity

To improve upon the render quality of the animation used in the DESTECS project, a game engine called Unity [21] was employed in the INTO-CPS setting. A conversion tool has also been developed during the project that converts a 3D scene produced in 20-sim into a 3D scene usable in Unity. The 3D scenery of the CP case study from section 3 was converted from 20-sim to Unity. The original 3D animation and the result of the conversion to Unity is shown in Figure 6.

The conversion is not yet completely covered. Some of the objects, lighting and cameras, among others, are implemented differently in Unity with respect to 20-sim, making it difficult to convert them. This can also be observed from the illustrations in Figure 6, as the camera angle is different between both animations. The difference in lighting causes colours to also look different. The colours, however, have been verified to be identical between 20-sim and Unity.

Unity has additional features to make an animation look more polished. Examples include support of virtual and augmented reality, the production of complex particle effects and the possibility to add assets like skyboxes and floor textures. As an example, a skybox and floor texture were added to the original Unity animation in Figure 6, resulting in Figure 7.
5.3 FMI Support for Unity Animations

A new tool has been created for Unity that enables arbitrary Unity models to be exported as tool wrapper FMUs that can be used in co-simulation on the platforms supported by Unity. It includes a thin FMI implementation that provides communication to the Unity animation stored inside the resource folder of the FMU. Upon instantiation it will launch the Unity application and establish a communication link between it and the FMI functions. During simulation, the new values are sent to the animation, which in turn renders the animation in relation to these new values.

The tool is implemented as a Unity plug-in that enables the FMU to be exported from within Unity at the press of a button. It has an editor extension that provides an interface to select the axes of objects that should be included in the FMU. It also enables the user to name the axes that become the inputs of the FMU. Among others, the supported axes are position, rotation and scaling of a 3D object, but also Unity-specific features like texture sliding, blend shapes and showing or hiding 3D objects.

Finally, to use the animation from section 5.2 during co-simulation, it has been exported as an FMU and the Plant FMU has been updated to export the signals required to drive the animation. The following signals have been added as outputs from the Plant and connected to the animation:

```plaintext
- TrayPositionCtrl_z.sphere2 = TrayModel.Tray.x
- TrayPositionCtrl_y.sphere2 = TrayModel.Tray.y
- TrayPositionCtrl_k.sphere2 = TrayModel.TrayAngle
- TrayPositionCtrl_z.sphere3 = TrayModel.Parcel.x
- TrayPositionCtrl_y.sphere3 = TrayModel.Parcel.y
- TrayPositionCtrl_k.sphere3 = TrayModel.ParcelAngle
- TrayPositionCtrl_z.sphere1 = TrayModel.Slider.x
- TrayPositionCtrl_y.sphere1 = TrayModel.Slider.y
- TrayPositionCtrl_k.sphere1 = TrayModel.SliderAngle
- TrayModel.Tray.y = Tray.Loader.Angle.y
```
6 Concluding Remarks

This paper has illustrated that it is possible to efficiently transfer collaborative models developed using the Crescendo technology to the INTO-CPS FMI-based setting. From a debugging perspective, the Crescendo technology still has some advantages over the new INTO-CPS tool chain. However, there are many limitations of Crescendo that are eliminated with the INTO-CPS tool chain [18]. Most importantly, INTO-CPS represents an open tool chain that enables users to include any tools that are able to provide FMI version 2.0 compliant co-simulatable FMUs. It is also worth noting that it supports the full development life cycle, from initial requirements towards the final realisations of all constituents included. We have also illustrated that the case study can be simulated in INTO-CPS and determined that events were not critical for the control of the model, while showing how constraints can be added to handle event-controlled models.

Acknowledgments. The work presented here is partially supported by the INTO-CPS project funded by the European Commission’s Horizon 2020 programme under grant agreement number 664047. We would like to thank all the participants of this project for their efforts making this a reality. We would also like to thank BEUMER Group for the ability to analyse this case study.

References