Communicating Process Architectures 2014
P.H. Welch et al. (Eds.)
Open Channel Publishing Ltd., 2014

Hardware Ports – Getting Rid of Sandboxed Modelled Software

Maarten M. BEZEMER and Jan F. BROENINK

Robotics and Mechatronics, CTIT Institute, Faculty EEMCS,
University of Twente, The Netherlands

{M.M.Bezemer , J.F.Broenink} @utwente.nl

Abstract. Model-Driven Design (MDD) techniques can be used to design control software for cyber-physical systems, reducing the complexity of designing such software. Most MDD tools do not provide proper support to interact with the environment of the models, i.e. the actuators and sensors of a cyber-physical system, resulting in the models being sandboxed. So-called "Hardware Ports" are proposed in this paper to get rid of this sandboxed model issue. These hardware ports are designed in a modular way, preventing a completely separated implementation for each type of piece of hardware. Even though the concept of hardware ports is generally applicable, in this paper the TERRA MDD tool and the LUNA concurrent runtime framework are used as examples to clarify the design and implementation. The paper concludes with a reflection on the usability of the hardware ports and on the planned future work to further improve the interaction between modelled software and the hardware it controls.

Keywords. code generation, hardware channel, hardware port, LUNA, meta-model, model-driven design, TERRA

Introduction

Systems consisting of mechanical, electrical and software components, called cyber-physical systems, are complex to control. The scope of their mechanical capabilities, along with the range of actuators and sensors that are available to them, are steadily increasing over time. As a result of these expanding capabilities, the complexity of tasks and environments that these systems are able to handle increases. Therefore, the control algorithms and software of cyber-physical systems become more complex.

The parallel nature of the world also needs to be taken into account by the control software, so a lot of these described tasks and components need to be updated at the same time, requiring the software to be concurrent. This also adds up to the complexity of the control software of cyber-physical systems. Designing control software for such systems becomes too complex to be done without the aid of Model-Driven Design (MDD) techniques. MDD is a technique that reduces the gap between a problem and the software implementation through the use of models and automated support for transforming and analyzing models [1].

Models are used to specify the architecture and/or the implementation of the control software. The structure of the models, consisting of their elements and properties, is defined by meta-models [2]. Since the models are compliant with their meta-model, all MDD tools that are compliant with the meta-model are able to interpret the models. MDD tools are used to construct the models, to transform a model into something else, or to obtain information from the models for all kinds of purposes. For example, a code generation tool converts a set of models into software that is able to control the cyber-physical system, and a simulation tool use the same models to mimic the software as if it is executed on the actual system.

When designing control software, the control algorithms need to be embedded in the software application. It is good practice to keep the algorithm components separated from the rest of the software [3], so the algorithms are not polluted with details that are not essential for them. This separation also increases the reusability and decreases the complexity of the algorithms. Keeping the software details from the algorithms can be done using software architecture models. These models define the components that are required for the complete application, like the algorithms, schedulers, hardware interaction, user interfaces, and so on.

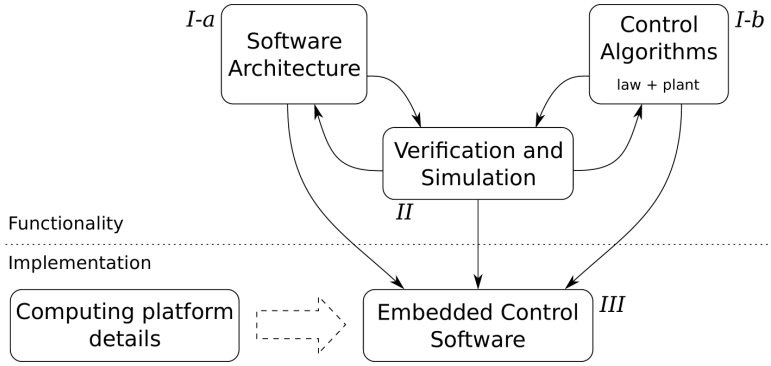


Figure 1. Design steps for embedded control software development of cyber-physical systems, based on [4].

The design steps that are required to obtain the control software are shown in Figure 1. The development starts with modelling the software architecture and the control algorithms, shown by step *I*. This is typically done using multiple (graphical) editors; each editor is able to handle its own meta-model. For example, the software architecture models require a different meta-model than the control algorithm models.

These models can be verified, formally checked and/or simulated, step *II*. Depending on the result of this step, the models are further refined until the results are as required.

Model transformations are used to convert the models into the control software, step *III*. Combined with the details of the computing platform, where the software is executed, the software is able to control the cyber-physical system. The focus in this paper is on this code generation step, and especially on the generation of code interacting with the computing platform.

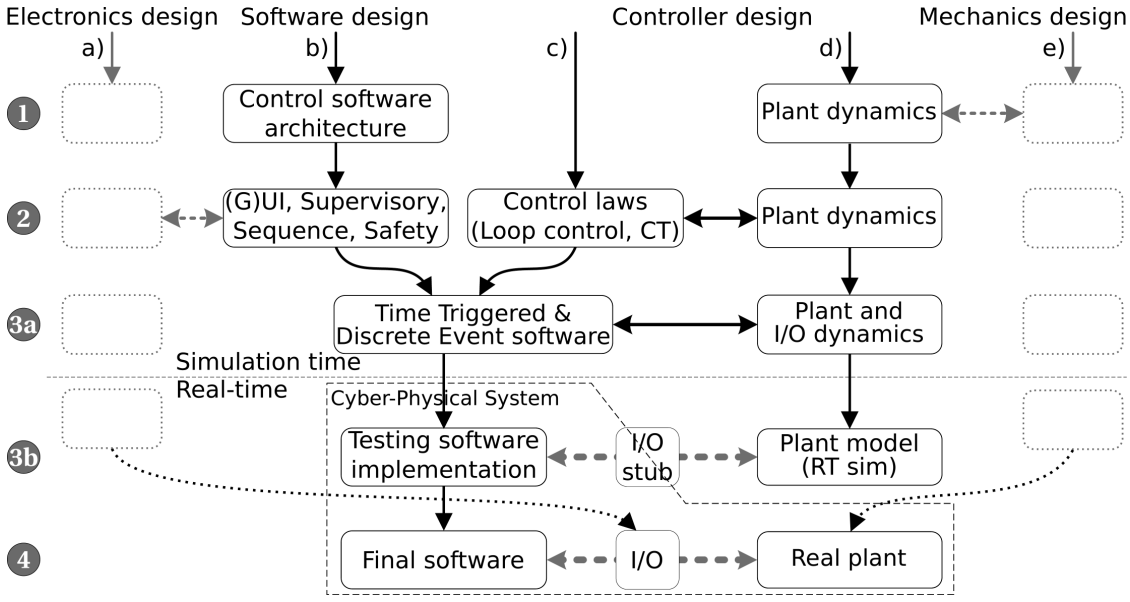


Figure 2. Steps of the process for designing control software for cyber-physical systems [4].

A more detailed overview of the software development steps is shown in Figure 2. Different starting points, or branches, are shown at the top of the figure and the detailed steps are shown on the left. For this paper no distinction is made between branches (b) and (c); both result in the control software. The same is assumed for branches (d) and (e): both result in the plant, either simulated or real. The full explanation of the figure and the design steps is available in [4].

There are many MDD tools available, but they either lack meta-model support to structure their models, proper modelling support to interact with the hardware of cyber-physical systems, or do not provide architecture software modelling capabilities.

EMF [5] is an example of tooling that supports meta-modelling extensively. It provides means to design custom meta-models based on the Ecore meta-model, which is even a meta-model of itself. EMF is suitable to design a software architecture meta-model and to generate accompanying (Java) code that supports creation, manipulation and storage of models based on the designed meta-model. It is suitable as a framework to support tools that require meta-modelling support, but it does not provide out-of-the-box features to interact with anything besides the (meta-)models.

MATLAB/Simulink [6], LabView [7] and 20-sim [8] are examples of tools that do support interaction with their computing platform or other types of hardware. All of these tools support MDD, but lack proper software architecture modelling support, i.e. they do not have meta-model-based architecture support.

The MDD tool-suite TERRA [4,9] makes use of EMF, but currently lacks the functionality to model the interaction with the computing platform. Basically, the resulting generated software is sandboxed within its execution environment, meaning that the software is unable to interact with the "outside world". Until now, breaking out of this sandboxed environment was done by manually modifying the generated code. TERRA is therefore used as a practical example for the implementation details to provide model-based hardware interaction.

The interactions between the software branch and plant branch, marked with the I/O (stub) blocks in steps 3b and 4, is the part that is lacking in the TERRA tool-suite. The actual implementation of these I/O blocks is different in each of these steps, as 3b requires interaction with the simulated plant model, while 4 requires interaction with the real plant.

The main goal of this paper is to provide a way to make the models more usable by getting rid of the sandbox situation. In order to fully obtain this goal, the provided way should support interaction with both the plant model and the real plant, without the need of building two different models. This additional requirement reduces errors due to manually keeping multiple models in par with each other, one for simulation and one code generation.

Besides usability, the goal and requirements also result in a reduction in design time, due to reusing the model for multiple activities. This is also less error-prone, as the designer does *not* need to recreate the model, with the likelihood of adding errors, for each activity.

In the first section of this paper, the way of working when modelling interaction with the environment outside of the modelled component is discussed, and is followed by a discussion on related work of hardware interaction from a modelling perspective. The next section goes into more detail on modifying the meta-models in such a way that modular support is provided for the interaction, by looking at the current implementation of the external tool/-model support in TERRA and reusing this for hardware ports. This is followed by a section describing some implementation details of hardware channels in LUNA and the conversion of hardware ports into them while generating code. Next, a simple use-case is presented and some details on the generated code is discussed. The paper finishes with conclusions and a future work discussion reflecting on the presented work.

1. Hardware Ports

As stated in the introduction, a model needs to be usable for interacting with both a simulated plant model and the real plant. Additionally, the user should not be forced into a specific way of working, i.e. the user should be free to choose a starting point to model the control software. It makes sense to either start with modelling the architecture of the cyber-physical system, usually consisting of the input, controller and plant blocks, or to start with the controller block and later adding the plant and input models when required.

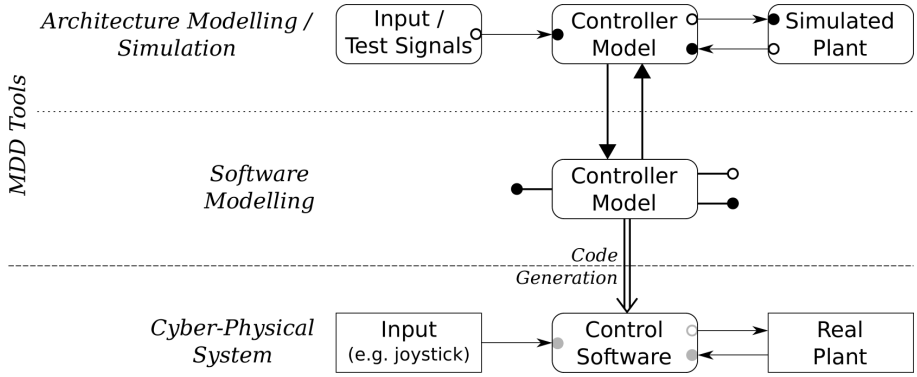


Figure 3. Different phases in modelling the cyber-physical systems software.

Both modelling phases are shown in Figure 3. At the top, the architecture modelling phase is shown and the software modelling phase in the middle. The architecture model is a typical use-case for control software with feedback loops, hence it is a reasonable starting point. In a situation where the designer is not (yet) interested in the complete system, it makes sense to focus on the controller model. Both phases result in a number of signals that become available while modelling, indicated with the horizontal arrows in the figure. The small circles describe the required signals of each component, i.e. the component ports.

These ports indicate the interaction of the controller with its environment, either the simulated plant, or the real plant. In the simulation phase, the component is used in an architectural overview and the component ports are connected to the inputs and plant components. After code generation the component ports need to be replaced, hence they are drawn in gray, by pieces of code that are able to interact with the actual hardware. The description above shows that the ports are required to be used in two different situations.

There are existing solutions to interact with hardware in other modelling tools, each is used in its own way with its own implementation ideas behind it. gCSP [10] uses derivatives of CSP reader and writer objects for the interaction with the hardware, called “Link Drivers”. They need to be connected, via channels, to regular readers and writers. The designer needs to add custom C++ code to these objects, which is used to construct the actual Link Driver in the generated software. From a modelling point of view this is fine, as the user is able to fully design models. But, from a code generation point of view this is not user-friendly, as the user needs to know the details of the cyber-physical system and needs to provide the actual C++ code for the link driver implementation to interact with the hardware.

20-sim on the other hand does not provide direct support for hardware interaction, but lets the user select a component from the model that needs to be executed on the computing platform. This selected component is exported to the 20-sim 4C tool that lets the user connect the “loose signals”, basically the ports as described earlier, to the hardware. Next it compiles and deploys the component to the target and provides commands to start and log the application. These steps are taken outside the “modelling environment” and are not part of the modelling process.

MATLAB/Simulink and LabVIEW are examples of modelling tools that do not provide explicit means of connecting the model to the hardware. They have focus on the signal paths between the inputs and outputs of the model. Hardware information needs to be put in the same model, which results in an undesired mix of control algorithm and deployment information.

In order to comply to an MDD way of working, the hardware interaction needs to be part of the model, so the gCSP and 20-sim approaches do not fit. Furthermore, the models (backed by their meta-models) need to be used for their specific domains. For example, mixing (control) algorithms with hardware interaction details should not happen. This keeps the models clear and understandable, resulting in less errors and lower (tool) complexity.

It also makes more sense to provide ports than readers/writers to interact with hardware. Initially, it seems to make sense to read from (or write to) hardware, but all other communication outside of the model is done using ports. Ports are in this sense effectively pass-through objects that let a channel go outside the enclosed environment of the model. Hardware interaction is a similar situation: so-called "Hardware Ports" should let a channel go outside of the model in order to "reach" the hardware.

The hardware ports need to be reusable in order to support both described phases and to prevent two copies of the controller model, one for simulation and one for code generation purposes. In the architecture model the ports need to behave as regular ports, as they are connected to other modelled (simulation) components. The code generation requires additional information to replace the port with a piece of code that is able to interact with the hardware. So only this additional hardware information needs to be stored in the the port model elements. This new "Hardware Port" element needs to be added to the meta-model to provide a structure to enrich models with this information.

2. Modular Meta-Model Design

The previous section introduced hardware ports to provide a means for the control software to interact with its environment, either the simulated or the real plant. In this section the addition of such a new element into the meta-models that are part of an MDD tool-suite is discussed.

As the meta-model explicitly specifies the structure of the model elements, it also needs to explicitly specify the required hardware information. Because "hardware" comes in numerous guises, each with their own interaction details, all of the required hardware configuration details need to be provided by the meta-model structure. This either results in a complex Hardware Port element with lots of unused fields, or in numerous slightly different Hardware Port elements. This first case makes it extremely hard for the MDD tools to properly provide support for the Hardware Ports, as they need to manage complex data sets, of which it will become unclear what piece of data is required in a specific situation. The second case requires the MDD tools to support all of the available Hardware Port elements separately, which makes the MDD tooling bloated and complex. Both cases are likely to result in tools that are difficult to maintain.

A better solution is to provide generic, flexible support for hardware ports. Specific hardware port implementation can then use the generic support to hook into the main tool. By using this scheme, neither the hardware port element requires lots of (unused) fields nor are there lots of hardware port elements required.

The hardware port support is similar to the support of external tools in TERRA. Therefore, this support is described first, followed by the description of the hardware port design.

2.1. Modular External Model Support in TERRA

As mentioned, during earlier TERRA development a similar problem was encountered with the "external tool/model" support in TERRA, providing support to use non-native models as component implementations in native TERRA models. A design philosophy of TERRA is to avoid reinventing models and tool support where there is existing tooling available. TERRA currently has support for external 20-sim [8] and SCXML [11] models. This first type of model adds support for control algorithm design and the latter for state chart design.

The tooling for such models is often specialised for their domains and it makes no sense to try and recreate such specialised tools. For this reason TERRA has the external tool/model support: it provides means to make use of the specialised tooling and models without reinventing and reimplementing these specialised tools.

There are numerous types of external models possible and adding meta-model elements for each of them would have resulted in the same disadvantages as described earlier for hardware ports. These disadvantages are tackled in TERRA by introducing so called "configuration elements". A configuration element provides the specific, additional information that is required to use the external model in TERRA. Each configuration element has its own tooling support, that hooks into the generic TERRA support for external models. An advantage is that this results in a modular design of the tools, as support for each configuration element is provided by a small separate tool, without the need to modify the existing tools.

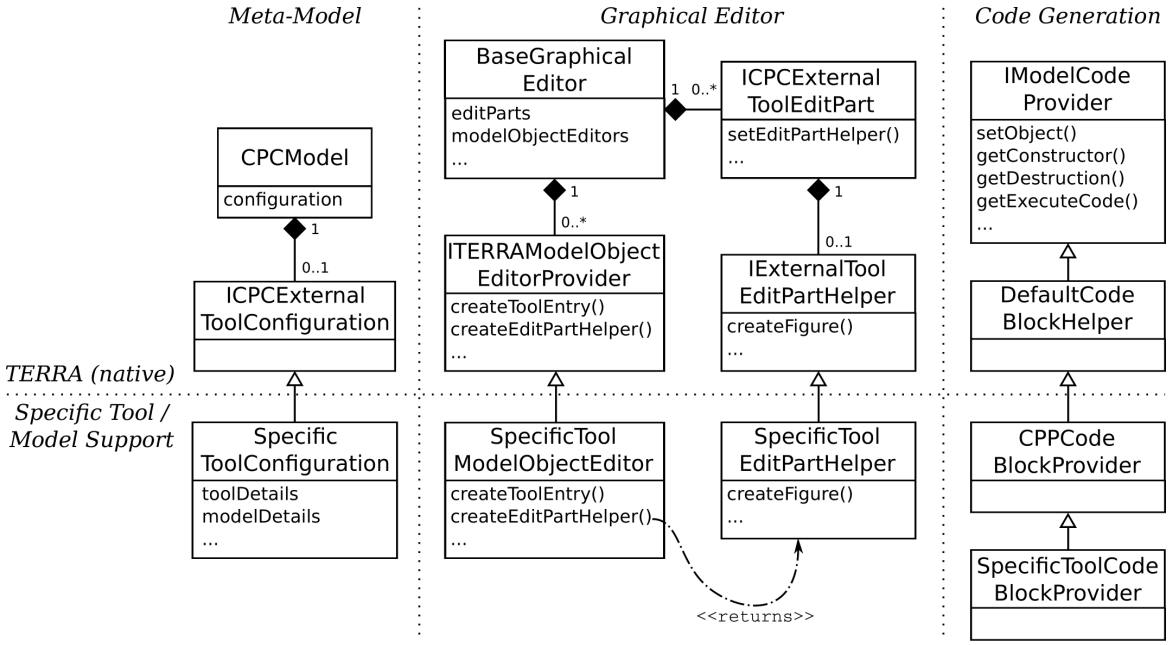


Figure 4. Partial overview of the modular external tool support in TERRA.

Some parts of the external tool support in TERRA are depicted in Figure 4. At the left the meta-model extension is shown that is required to support the external models. `CPCModel` is an element of the Component Port Connection (CPC) [3] meta-model, which is used as the base of all other TERRA meta-models. CPC defines a component as an element that provides the designated, functional parts of the system. A port is used to indicate that a component requires data from another component and connections are used to connect ports to each other to determine the data flow. With respect to the CPC paradigm, `CPCModel` is a component that has another model as its implementation[1], either a native TERRA model or an external tool model.

[1] A more detailed overview of the CPC meta-model and the `CPCModel` element is provided in Chapter 6, Section 6.3.1 of [4].

Each CPCModel has a placeholder that might be used to contain a element that is based on ICPCExternalToolConfiguration to store details of the external tool/model. If it does contain such a configuration element, the CPCModel element is handled by the "Specific Tool/Model Support". BaseGraphicalEditor has a list of ITERRAModelObjectEditor-Provider objects, these providers hook into the TERRA (base) editors by providing for example a tool entry that can be used by the model designer to add a CPCModel element to the design.

Each model element is handled in the editors by a so-called "EditPart", which is used in a modular way to provide interaction between the model and the editor. A CPCModel element containing a configuration element, needs a custom EditPart implementation, of which the interface is described by ICPCExternalToolEditPart. The actual tool-specific implementation is obtained via the createEditPartHelper() method of the SpecificTool-ModelObjectEditor.

Code Generation works in a similar way, the CPCModel configuration is used to generate the (C++) glue code for the external model. The actual code for the external model is usually provided by the external tool itself, as the required (meta-)model specification and the required code generation knowledge are contained within the external tool.

The description above shows how the specific support for an external model is provided in a modular way within TERRA and that the actual implementation is strictly separated from the generic support for external models. The plug-in support of Eclipse is used to provide the tools that handle the external models. Each plug-in registers itself with the native TERRA components, using the "extension point" support of Eclipse. The registration informs TERRA about the information on the type of external model they support and how this support is provided. This mechanism is used for example to fill the list of ITERRAModelObjectEditor-Provider objects mentioned earlier.

2.2. Hardware Port Design

For hardware ports, the approach of configuration elements and separation of the hardware specific tooling is chosen, similar to the described situation of the external models of the previous section. The ease of adding support for new types of hardware is especially important, as each cyber-physical system often comes with its own hardware components that need to be controlled by the software.

Though the description seems complex, the approach of configuration elements works well. Besides, the hardware ports require fewer hooks and are therefore less complex than the external model support, which is due to the nature of the model elements[2].

The design of the hardware port support is depicted in Figure 5, showing the similarities with the external model support. As the CPCPort (and thus ArchPort as well) already has a complex nature, we chose to add a new meta-model element (ArchHWPort) instead of extending the existing port element with the hardware configuration details. This prevents making the CPCPort even more complex to be managed by the TERRA tool-suite. However, for another implementation scenario another decision might be better. This results in each ArchHWPort requiring a placeholder to contain a configuration element that is based on IArchHWPortConfiguration specifying the details of the hardware.

The support for the graphical editor is straightforward: when an ArchHWPort element is encountered in a model, the "EditPart factory" uses the implementation of IArchHW-

[2]Besides the CPCModel element, the CPCExternalModel element also implements ICPCExternalTool-Configuration. This model element is used to refer to a model (implementation) that exists outside of the current model. Therefore it is not possible to directly extend the EditPart of the meta-model elements without having to provide the same code in all of the EditPart implementations (Java limitation); the shared code is provided in the EditPartHandlers.

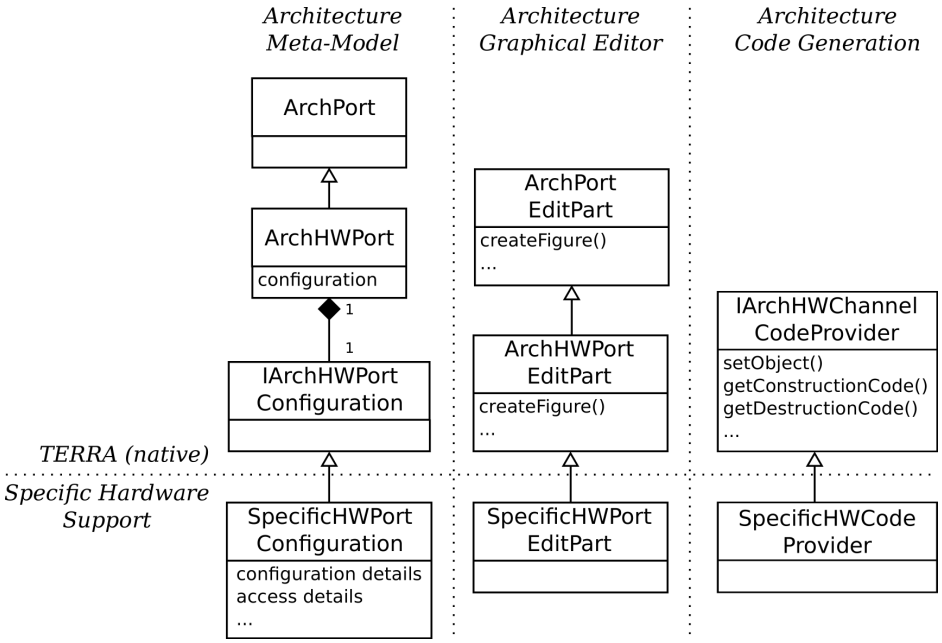


Figure 5. Modular hardware port support in TERRA.

`PortConfiguration` to provide the corresponding `SpecificHWPortEditPart`. Similarly, `SpecificHWCodeProvider` is used to provide the actual code for the `ArchHWPort` elements.

3. Implementation

As a proof of concept, the initial implementation provides support to directly control hardware pins only. The hardware pins are provided by a Mesa Anything I/O FPGA board [12]. The FPGA on this board is accessible by "Memory Mapped" registers. These registers are programmed to set or read the state of the pins.

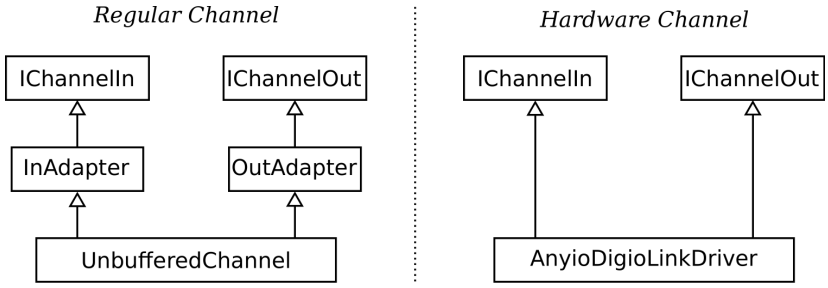


Figure 6. Composition of regular and hardware channels.

The main part of the implemented code support is provided by LUNA [4,13] in the form of "Hardware Channels", depicted in Figure 6. Regular (CSP) channels provide means to communicate data from a writer to a reader, and hardware channels replace either side with hardware. For example when writing to a hardware channel, the channel sends the written data to the actual hardware. The same holds for reading from a hardware channel.

Regular channels are slightly more complex, as they use adapters to support different amounts (one or multiple) of readers and writers that are connected to the channel. Hardware channels do not support multiple readers or writers: it makes more sense to control the hardware from one point in the application, so the adapters are left out.

The `IChannelIn` and `IChannelOut` interfaces are used by readers and writers to communicate over the channel. The actual channel implementation is hidden from readers and

writers, resulting in the difference between regular and hardware channels being indistinguishable. As a result, TERRA code generation is able to generate the standard readers and writers, even when hardware communication is required.

The only thing that is required by the code generator, is to generate the proper hardware channel for each hardware port that is encountered in the model. Only the code to construct (and to break down) a hardware channel needs to be generated; see `IArchHWChannelCodeProvider` of Figure 5. The actual implementation is provided by LUNA. The hardware configuration, provided by `SpecificHWPortConfiguration`, is used to properly construct the channel.

```
function constructElement(modelElement)
   if modelElement.isTypeOf(Channel) then
      if modelElement.isConntectedToHardware() then
         # Generate HW channel using the specific code provider
         hwPort = modelElement.findHardwarePort()
         configuration = hwPort.getConfiguration()
         codeProvider = getCodeProviderFor(configuration)
         codeProvider.getConstructionCode(configuration)
      else
         # Generate regular channel
         constructChannel(modelElement)
      end
   else if modelElement.isTypeOf(...) then
      # construction code for other elements...
   end
end
```

Listing 1. Pseudo code showing the steps to generate construction code for hardware ports.

The steps to generate the construction code for hardware channels are shown in Listing 1. When a channel is encountered, a check is done to detect whether it is connected to a hardware port or not. If this is the case the hardware port is used to obtain its configuration element. This configuration element is then used to find the specific code adapter that is provided by the tool that provides the support for that particular piece of hardware. The code provider is used to finally obtain (generate) the construction code that is required to construct and configure the hardware channel.

The code to destroy the hardware port, i.e. clean up the hardware port and free its resources, is obtained using the same steps.

The first hardware port implementation supports the interaction with generic I/O pins of the Anything I/O FPGA board. It is able to either read or write a single (`bool`) bit or a series of 16 bits (`uint16_t`) without any manual modifications to the generated C++ code.

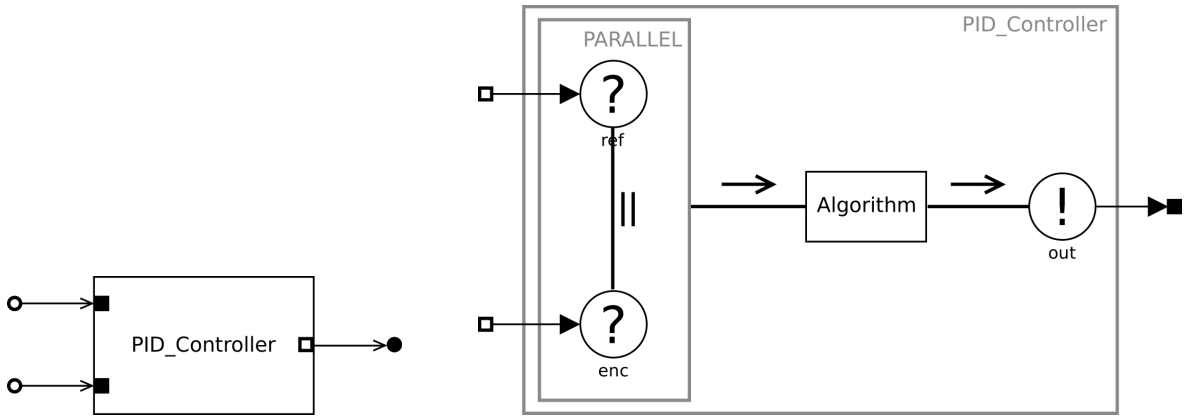
After the initial implementation was finished and working properly with the editors and code generation, two other hardware port implementations were added to TERRA: one to read encoder values and one to write pulse width modulation (PWM) values. Due to the flexibility and simplicity of the generic hardware-port support we have already described, it took less than half an hour to fully implement the meta-model, editor support and code generation for each of the new hardware ports. Note that LUNA already contained the Anyio link drivers, which is the C++ code that actually performs the interaction with the hardware.

The only part of TERRA that required modification was the set of Anyio hardware support plugins. The hooks provided by the generic hardware port support were sufficient: no generic TERRA code needed to be modified to add these two new hardware ports.

4. Use-Case

The use-case example, shown in Figure 7, is a modelled PID controller. This is a basic control algorithm that uses two input signals, a reference and a measured signal, to calculate the required value of the output signal. It is typically used to control a motor to let it follow a desired path using an encoder to measure the actual angle of the motor axis.

Figure 7a contains the architecture model, consisting of 3 hardware ports connected to the PID controller. The hardware ports are depicted as circles, instead of the squares which are used for regular ports. Shown at the left are the reference input (top) and the encoder (bottom) input hardware ports, and the PWM output hardware port is depicted at the right. Figure 7b contains the CSP implementation of the PID controller. Its readers and writer are actually interacting with the hardware ports via the channels.



(a) Architecture model (b) CSP implementation model

Figure 7. Use-case example showing a control algorithm with 3 hardware ports.

The generated code to construct the reference link driver, as a result of the reference hardware port, is shown in Listing 2. A settings structure is created and filled using the modelled details. The settings structure is finally provided to the `AnyioDigioLinkDriver16Bit` class via its constructor. The class, being part of the LUNA framework, uses these settings to actually interact with the hardware providing the reference values. The generated C++ code is provided by `getConstructionCode()` function of the code provider belonging to the Anyio tool. This function is called by the hardware port support of the generic code generator.

```
AnyIO::AnyioDigio16BitSettings settings;
settings.deviceNr = 0;
settings.address = 0x68;
settings.controlAddress = 0x6a;
settings.bitCount = 16; // 16 bits signal
settings.direction = 0; // input
myReferenceSignal_to_PID_ControllerRefChannel =
            new AnyIO::AnyioDigioLinkDriver16Bit(settings);
```

Listing 2. Generated C++ code to construct the reference hardware port/link driver.

The generated C++ code of the other hardware ports of the use-case example is similar. These other ports have their own code providers, using different settings structures and LUNA link driver classes to implement the interaction with their type of hardware.

5. Conclusions and Future Work

The main goal of this paper is to provide a means to get rid of the sandbox environment that is present when designing control software for cyber-physical systems using modelling techniques. The software architecture models are enriched with hardware port elements that allow the control software to interact with the hardware of the cyber-physical system. These proposed hardware ports thereby solve the issue of the modelled software interacting with the outside world while keeping an MDD way of working in mind.

The reusability of the software architecture models is improved, as these models use the hardware ports to specify the capability of the hardware (I/O) and how to connect to it. This leaves the actual controller components free of such specific non-related details, making them less complex and more generically applicable. Removing the manual task of connecting the software to the hardware also reduces the likelihood of introducing errors and reduces the design time of the control software.

Initial tests showed that the improved reusability and less manual coding tasks are promising. Further experimentation is required to find out whether the overall decrease in design time, the lower amount of errors, and better quality of code are indeed the case. We plan to involve undergraduate and graduate students studying mechatronics and embedded systems in these kind of end-user tests.

The architecture models are also reused for simulation purposes. The hardware ports are used as indicators for connections to the outside world, which need to be fed with (simulated) signals. Simulation software then is able to connect plant models and other input/test signals to the hardware ports in an automated way. This allows the developer to easily simulate the complete cyber-physical system, in order to test the controller models.

The flexibility of the generic hardware port support is discussed at the end of section 3. The TERRA implementation of the proposed hardware port support is organised in a modular way. The modular hardware port design dictates which meta-model, editor and code generation elements need to be provided in order to support new hardware. Adding two new hardware port types took less than half an hour to fully implement, without the need to modify any part of the generic TERRA code.

In our research group, we have a (modified) Gumstix board [14] that is intended to become a standard platform for our setups. TERRA needs to support this standard platform as well, so a new plug-in is planned to add several hardware ports to support the variety of I/O that comes with the board. It is expected that this also can be done without modifying the generic TERRA hardware port support.

TERRA should also be extended with support for complete hardware platforms. This additional support provides means to add the details of a complete hardware platform, like the Gumstix board. TERRA users then can use this platform support to add all available hardware ports to a (new) model at once, completely configured and ready to use. In this situation, TERRA users are able to easily attach the controller components to their required hardware ports, without the need to find out how to exactly access the hardware and to manually fill the configuration elements of the hardware ports with these details.

Besides the improved ease of using the hardware ports, this additional support also could provide knowledge of the actual computing platforms that are used. This platform knowledge can be used to generate distributed software, that spreads the computation load over the available platforms, while making sure that a specific piece of software is placed on the platform that provides the required I/O. Sunter [15] provides more details on this subject. Furthermore, automated deployment strategies could make use of the available platform knowledge to update the software of the multiple computing platforms in an automated way when the corresponding models are updated.

As described above, the hardware ports are suitable for interaction of the software with

its environment as well as for simulation purposes. Simulation support still needs to be designed and implemented into TERRA, in order to make full use of the hardware ports and the described reusability advantages.

References

[1] R. France and B. Rumpe. Model-driven Development of Complex Software: A Research Roadmap. In *2007 Future of Software Engineering*, FOSE '07, pages 37–54, Washington, DC, USA, 2007. IEEE Computer Society.

[2] C. Atkinson and T. Kühne. Model-Driven Development: A Metamodeling Foundation. *Software, IEEE*, 20(5):36–41, September 2003.

[3] M. Klotzbücher, N. Hochgeschwender, L. Gherardi, H. Bruyninckx, G. K. Kraetzschmar, D. Brugali, A. Shakhimardanov, J. Paulus, M. Reckhaus, H. Garcia, D. Faconti, and P. Soetens. The BRICS Component Model: a Model-Based Development Paradigm For Complex Robotics Software Systems. In *Symposium On Applied Computing*, number 28. ACM, 2013.

[4] M. M. Bezemer. *Cyber-Physical Systems Software Development - way of working and tool suite*. PhD thesis, University of Twente, November 2013.

[5] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks. *EMF: Eclipse Modeling Framework 2.0*. The Eclipse Series. Addison-Wesley Professional, 2nd edition, 2009.

[6] MathWorks. Simulink - Website, May 2014. `http://www.mathworks.nl/products/simulink/`.

[7] National Instruments. NI LabVIEW - Website, May 2014. `http://www.ni.com/labview/`.

[8] Controllab Products. 20-sim - Website, May 2014. `http://www.20-sim.com/`.

[9] M. M. Bezemer, R. J. W. Wilterdink, and J. F. Broenink. Design and Use of CSP Meta-Model for Embedded Control Software Development. In P.H. Welch, F. R. M. Barnes, K. Chalmers, J. B. Pedersen, and A. T. Sampson, editors, *Communicating Process Architectures 2012*, volume 69 of *Concurrent System Engineering Series*, pages 185–199. Open Channel Publishing, August 2012.

[10] D. S. Jovanović, B. Orlic, G. K. Liet, and J. F. Broenink. gCSP: a graphical tool for designing CSP systems. In I. East, J. Martin, P. H. Welch, D. Duce, and M. Green, editors, *Communicating Process Architectures 2004*, volume 62 of *Concurrent Systems Engineering Series*, pages 233–252, Amsterdam, September 2004. IOS press.

[11] Voice Browser Working Group. State Chart XML (SCXML): State Machine Notation for Control Abstraction. Technical report, World Wide Web Consortium, May 2014.

[12] Mesa Electronics. Mesa Anything I/O FPGA Cards - Website, May 2014. `http://www.mesanet.com/fpgacardinfo.html`.

[13] M. M. Bezemer, R. J. W. Wilterdink, and J. F. Broenink. LUNA: Hard Real-Time, Multi-Threaded, CSP-Capable Execution Framework. In P.H. Welch, A. T. Sampson, J. B. Pedersen, J. M. Kerridge, J. F. Broenink, and F. R. M. Barnes, editors, *Communicating Process Architectures 2011, Limmerick*, volume 68 of *Concurrent System Engineering Series*, pages 157–175, Amsterdam, November 2011. IOS Press BV.

[14] Gumstix Inc. Website, May 2014. `https://www.gumstix.com/`.

[15] J. P. E. Sunter. *Allocation, Scheduling & Interfacing in Real-Time Parallel Control Systems*. PhD thesis, Control Engineering, University of Twente, 1994.