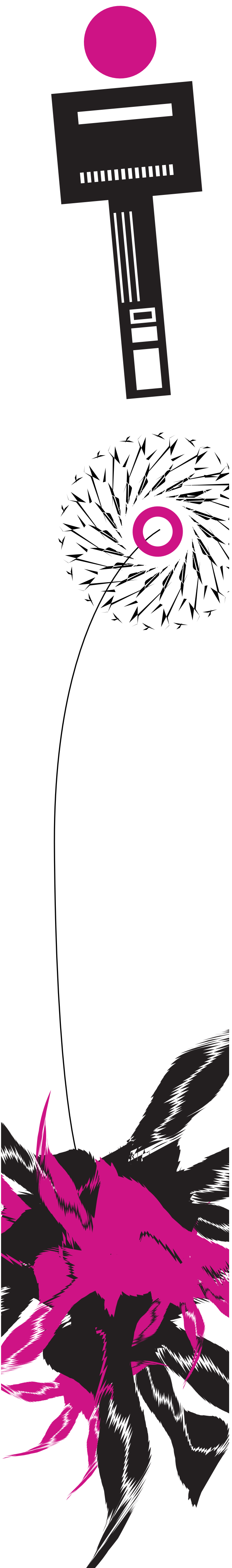

ROBOTICS
AND
MECHATRONICS

DATAFLOW-BASED MODEL-DRIVEN ENGINEERING OF CONTROL SYSTEMS

K.A.E. (Koen) Zandberg

MSC ASSIGNMENT

January, 2020

003RaM2020
Robotics and Mechatronics
EEMCS
University of Twente
P.O. Box 217
7500 AE Enschede
The Netherlands

UNIVERSITY OF TWENTE. | TECHMED CENTRE

UNIVERSITY OF TWENTE. | DIGITAL SOCIETY INSTITUTE

Contents

1 Introduction

With the need for more advanced control systems, tools and applications to aid the design of these control systems help to reduce the number of design iterations required. One of the often used design paradigms is model-driven development (MDD) (Selic, 2003). These paradigms allow the engineer to focus on the control system by offering abstraction layers for both the hardware platform and the software implementation details.

The increasing demand and increasing complexity of embedded control systems demands strong requirements on the design paradigms. The systems, often distributed, are limited in resources while still having to adhere to strict execution deadlines. With these limitations and added requirements to the interaction between the software and the hardware components, designing a full embedded control systems becomes a challenging exercise.

While it is clear from earlier work that a model driven structured approach with accompanying tools is a suitable approach, there is no single correct solution. Multiple approaches each with their advantages and disadvantages are available. Often these approaches focus on a hierarchical structural approach to the software based on object oriented design principles. From a control engineer working with an algorithmic approach, this is not an optimal approach.

1.1 Context of This Thesis

This thesis shows a stepwise-refinement approach based on dataflow representations of the models as an alternative to the previous work at the research group. The workflow is geared towards an incremental approach to control system design, each step increasing the level of detail available in the models.

The approach uses a dataflow representation internally for analysis and simulation of the models. The dataflow-based approach offers analysis into the latency of the models.

The main use of dataflow models is in the analysis of DSP applications, but is used here for modelling control systems. In contrast to control systems, for DSP systems throughput is often more important than latency. Within this paper, throughput analysis is going to be neglected in favor of latency analysis.

An analysis and simulation application is presented and used as proof of principle of the approach. This is used for verification of the ideas and for validation of the proposed workflow. At last it will be used to provide answers to the research questions.

1.2 Application Workflow

A top down approach to the design of these control systems is given here with an accompanying application to enhance the development. The workflow used for the system design flow is based on Broenink, Ni, and Groothuis (2010) and Broenink and Ni (2012). The multi-step refinement approach allows for a modular and incremental way to the challenges of the control system design.

The different steps of the workflow are shown in Figure 1.1. For an end user (control engineer) this application workflow consists of a multi-step approach where each step increases the level of refinement of the model of the control system descriptions until the result is the generated application code. This workflow allows the engineer to supply a high-level model based on the component interaction as a start. The application should allow for verifcation of functional properties, with emphasis on non-functional properties such as deadline analysis. After the final step, platform-specific application code is presented as implementation of the control system on the selected platform.

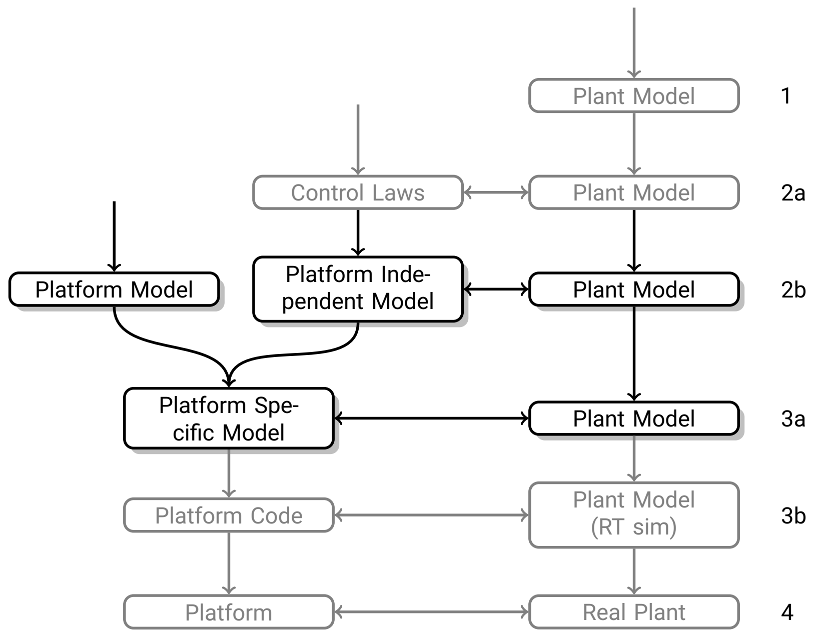

Figure 1.1: Stepwise refinement workflow for control systems, the bold steps are the main focus of this thesis.

The first step of the workflow is to have a set of control laws engineered for the plant model designed. The design of the control laws itself is out of scope for the application proposed here, but must be in a format suitable for conversion to a block diagram-like structure. This step is shown as stage 2a of the workflow described in Figure 1.1.

The next step as shown in stage 2b of the workflow is to convert the control laws into a platform-independent model (PIM), converting the control laws into a suitable format for the application. The PIM allows for early formal verification of the model using modelling analysis features, and for functional verification of the model by co-simulation with the plant model. The PIM allows for analysis and simulation with arbitrary accuracy, with infinite accuracy assumed by default. Function blocks are assumed to have no delay, but can be configured with a non-zero worst-case execution time.

Parallel to this stage models of the available hardware platforms are created. These models contain the specifications of the hardware components on the platform. The separate platform models can be used as a library and they are aimed to be reusable by multiple different control systems.

In the third step, the model is adapted to a chosen target platform system, an automated conversion from the PIM to a platform-specific model (PSM). The specification from the platform models is used to augment the PIM with details from the platform model to a PSM. The goal is to have a model that closely matches the worst-case behaviour on the target hardware platform, enabling analysis and verification of the behaviour of the system on the target hardware.

Step 3b from Figure 1.1 consists of a translation of the PSM to platform code. Based on the hardware description from the second step, platform- and application-specific code is generated. The resulting code can be deployed on the hardware and should function comparable to the earlier analysis. This step is out of scope for this thesis.

1.3 Research questions

The workflow described above offers a number of challenges when using dataflow-based models. This thesis attempts to provide answers to the challenges involved in step 2b and 3a of the workflow. The set of research questions presented here are used to explore these challenges and come to a set of requirements for the application.

1.3.1 Which dataflow models of computation are suitable as basis for model-driven development?

Within dataflow different models of computation (Stuijk et al., 2011) are available. Comparisons of these models often results in a trade-off between available formal model analysis techniques and modelling expressiveness. Not all models provide the analysis techniques or the expressiveness to model control systems. The focus of this question is to determine which models of computation are suitable for modelling control systems, resulting in a list of models of computation which offer enough expressiveness while offering enough analysis techniques to still allow for sufficient model verification.

- How is the system limited by the choice of dataflow model of computation? When a control system is modelled as dataflow model, sufficient modelling expressiveness is required to represent the control system as the dataflow model. Some models of computation are too restricted to properly model all possible aspects of a control system.
- Which analysis techniques from the dataflow models of computations can be applied for control-model analysis? Each model of computation provides a set of analysis methods. The analysis required for formal verification of the control system must be mapped to these analysis methods. The expressiveness of the Model of Computation (MoC) can be limited to satisfy requirements of required analysis techniques, placing restrictions on the expressiveness available for the control system models.

Together these subquestions limit the available models of computation by either a minimum amount of required expressiveness or the required analysis provided.

1.3.2 Which simulation features are required to verify the supplied models?

While dataflow based models allow for some analysis by themselves, it does not include functional aspects of the system in the models. To extend the models in such as way to allow functional simulation, a dataflow simulator must be extended to include these features.

- Which additional details are required for improving the design and analysis of models? Model design can benefit greatly from additional details and features added to the models. This includes features to clarify the model and limit the interpretation ambiguity of the models. A simulator must provide the level of detail required for a sufficient model verification required by the end user.

- How should external models be integrated in the simulator? Accurate model simulation is only possible with an accompanying plant model integrated in the simulator. Without this model, it is not possible to verify the model against the real-world usage of the system.

The goal is to end up with an extended feature set on top of the dataflow models to allow for functional simulation of models. Without these features the analysis is restricted to timing analysis, but without verification of the model functionality it is not possible to show how the model behaves under the timing restrictions. These questions attempt to seek for answers on which features must be implemented to add sufficient analysis options to the simulator to allow the end user to verify the functionality of their models.

1.3.3 Which platform specific information is required to transform the dataflow based model into a PSM?

Translating the dataflow based model to a platform specific model requires refinements to the initial model adding platform information. The platform meta model (PMM) provides the necessary platform abstraction for the control model to enable this transformation. This abstraction must strike a balance between providing enough detail for the control model to adapt without increasing the complexity necessarily

- Which properties of the PMM are strictly necessary for the model-to-model conversion? The PMM must properly propagate platform-specific limitations to the model. Based on these limitations, the model-to-model conversion can be adapted to adhere to the requirements of the platform.
- How should the PMM be structured to allow for reuse of different components? Multiple target platforms can share components between them, such as different platforms sharing an identical sensor or actuator. This reuse can exist at multiple levels, Different SoCs could share processor architectures or peripheral IP implementations. For efficient use of the PMM, enabling the reuse of information is a must have.
- Which extended analysis options are available with the additional information from the PMM for the dataflow model? Adding the extra information from the PMM results in a more detailed and more specific model. Information such as allowed parallel execution or peripheral access modes can add additional refinement to the model. This extra information can then be used to affirm whether the model is realisable on the specified platform.

The resulting PMM, when adhering to the results of these questions, offers the required features for the model-to-model conversion and allows the refinement of the PIM into the PSM. Furthermore it offers a basis for the code generation required for a final model-to-text conversion.

1.4 Thesis Structure

The rest of this thesis is structured as follows. First the previous work relevant for this thesis is described (Chapter 2), including an concise exploration of the different dataflow models of computation and a description of a number of different existing model-driven engineering solutions. Based on this, a dataflow model of computation is selected in the analysis (Chapter 3). The extensions required to the simulator are also discussed in the analysis. From this analysis and the research questions a set of requirements for the proof-of-principle application are formulated in the design decisions (Chapter 4). These design requirements are used as a basis for the application of which the design principles are described in the execution in Chapter 5. The resulting features are extensively tested and verified in Chapter 6: Testing. In Chapter 7, a further case study showing the design steps, expectations and results for a model is presented in the case study. Finally the results from the verification and the case study are discussed in Chapter 8 and finally the research questions are answered in Chapter 9.

2 Background

2.1 model-driven development Methodologies

Code generation based on model-driven development (MDD) as modelling system is not a new area. Already a number of solutions exist with multiple different approaches to MDD. These approaches distinguish themselves based on modelling language, component representation and offered features.

RobotML (Dhouib et al., 2012) is a DSL for design, simulation and deployment of robotic application. The model consists of architecture, communication, behaviour and deployment meta-models. Each meta-model has their own function defining different aspects of the robotic application. The architectural model defines the high level aspects of the robotic application. This includes the robotic system, describing the structural composition of the application using the CPC model. The communications aspect handles formalized data exchange using ports. Behaviour allows defining high level instructions using algorithms or finite state machines. Deployment specifies the target robotic platform, assisting in the code generation for the appliction A full eclipse based toolchain provided with the workflow. RobotML allows for defining non-functional properties such as timing information. This can be fed into schedulability analysis tools e.g. Cheddar (Singhoff et al., 2004).

The BRICS component model (Bruyninckx et al., 2013) describes two paradigms – Model-driven approach and separation of concerns. The separation of concerns paradigm is mapped to the 5Cs: Communication Computation, Configuration, Coordination and Composition. The model-driven approach follows a Component-Port-Connector (CPC) meta-model. The components represent computations and can be hierarchically composed to represent composite components. Ports represent communication where connectors combine two compatible ports, Configuration allows for influencing the system behaviour by configuration parameters Coordination determines the behaviour of the components and how the different components must function together. Composition brings the model together as a instance of a particular system, allowing decoupling and reuse where necessary. The workflow as defined by the BRICS model is roughly as follow: A structural architecture is defined using components, ports and connectors. Each complex component includes a coördinator based on state machines. A model to text transformation is applied to generate the code.

SmartSoft (Schlegel and Worz, 1999) has a model based approach based on a multilayered component approach. SmartSoft provides primitives as building blocks to create robotic systems. The internal component implementation is abstracted away by a skeleton representation. Modelling is based on UML based diagrams to facilitate reuse of components. The component itself define strict interfaces to provide reliable reuse of components. Models do not contain any timing information, although one of the target platforms is a real time operating system.

The V^3CMM components meta-model (Diego et al., 2010) uses three complementary views: Structural, coordination and algorithmic to achieve a component based model. The structureal view provides the static structure of the required components. The coordination view describes the event-driven behaviour of each component. Lastly the algorithmic view provides a way to express algorithms implemented by the components. Components are reusable in multiple ways. The modelling tools use UML to represent the models using both class diagrams and state transition diagrams.

The model-driven approach as described for TERRA (Bezemer, 2013) separates the design workflow into a multi stage approach; each stage refining the model. The TERRA models are based on the Communicating Sequential Processes (CSP) language. Hardware design and loop control components are separated into two branches. It allows for a strong separation of components and hardware. Furthermore the incremental approach of TERRA allows for increasing the complexity model in a stepwise manner.

While not containing a code generator, the *Ptolemy II software framework* (Liu et al., 2001) contains functionality to experiment with actor-oriented design. It supports multiple different models of computation, synchronous dataflow among others. It allows for hierarchically designed models and is able to model heterogeneous systems. The aim is to be able to study timing properties of different hybrid systems.

ThingML(Harrand et al., 2016) is a framework for code generation targeting heterogeneous platforms aimed at sensor network applications. Multiple different target languages and platforms are targeted and a lot of work went into tuning the approach for high customizability. It supports multiple different architectures and frameworks as targets. Each language has their own set of code generator targets. Furthermore, functionality is split into a number of categories. For each category, behaviour can be overridden by using inheritance. Support for so called channels and connectors is used to implement either inter-actor communication possibly spanning multiple physical systems. This also includes functionality to communicate directly to peripherals, for example I²C connected peripherals. Behavioural implementations are generated by generating state machine structures. It can generate either full code for a component or rely on middleware API's to support the required functionality.

Among the advantages of ThingML is the use of a single language with multiple supported platforms as target backend, resulting in a platform independent specification. To allow for a practical system, the system must allow for extending and integrating the generated code with existing code. This to allow for a gradual adoption of model-driven engineering (MDE) and allowing interfacing legacy code.

Summarizing, most of the approaches describe the models using an UML-like structural approach where the relation between different components is specified. This allows for detailed descriptions of the models, focussing on describing the functional aspects of the models. However they lack simulation and analysis options required for verification of the system models.

In contrast to most of the approaches listed here, the proposed workflow will focus on letting the end user create a model from a chain of computational functions. The approach used here allows for providing analysis and functional simulation during the modelling process.

2.2 Dataflow Models of Computation

Dataflow is a modelling technique to describe operations on a stream of data. Closely related to Kahn Process Networks (Kahn, 1974), dataflow diagrams can be used to describe concurrent real-time systems, and data processing applications. Different MoCs are available within dataflow, each with different limitations on expressiveness and analysis. Figure 2.1 shows a hasse diagram of different dataflow MoCs. Each MoC in the diagram is able to model all MoCs below itself.

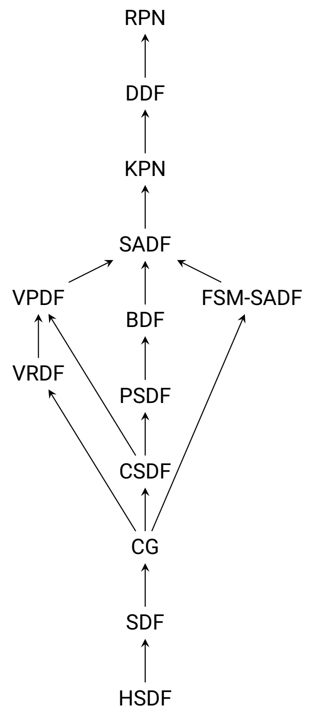

Figure 2.1: Hasse diagram of different dataflow models of computation

2.2.1 Dataflow Theory

A concise description of dataflow operation theory is required as background information. Dataflow diagrams are expressed as a set of actors. Each actor has an associated predefined firing duration. An actor fires as soon as it's firing condition is satisfied. For most dataflow MoCs the firing condition is satisfied as soon as each input edge contains at least the number of required tokens for that input. Tokens are used to represent data within the model, representing abstract data. An actor consumes the required number of tokens from each inbound edge when starting and releases the indicated number of tokens on each outbound edge when done.

Actors are connected using edges. An edge can be modelled as a buffered directional unbounded communication channel. Edges can contain an initial set of tokens.

An actor is only limited by the firing condition before it is executed. There is no limitation on concurrent execution of a single actor. However it can be limited by adding a self edge with only a single token

The MoC selected for this system should allow for enough expressiveness to describe control systems with, while allowing for enough analysis possibilities to validate the control system.

An example dataflow graph is shown in Figure 2.2. The dataflow graph contains two actors, named `t1` and `t2`, each with an execution time of 1 ms, indicated by the ρ symbol. `t1` Has a production of 4 tokens on the edge to `t2` and consumption of 1 token on the edge from `t2`. Actor `t2` has opposite but matching consumption rates. Furthermore 1 initial token are available on the edge from `t2` to `t1`, indicated with δ symbol. Each actor also has a self edge with one initial token and consumption and production equal to 1. This limits parallel execution of each actor by requiring a token to be available on the self edge. This token is consumed when the actor execution starts and produced when the execution of the actor finishes.

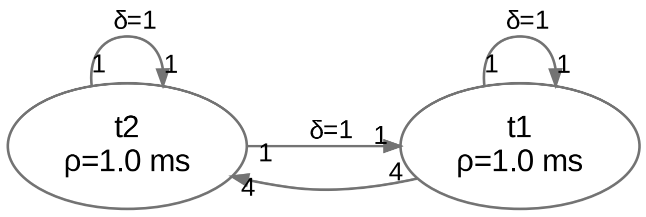

Figure 2.2: Example synchronous dataflow model with two actors.

A limited set of dataflow MoCs relevant for this work is presented here. Not all properties of the different MoCs are described. Features required for this work are emphasised.

2.2.2 Synchronous Dataflow

Synchronous dataflow (SDF) allow for modelling statically scheduled applications. Dynamic behaviour however is not possible. Within an SDF model, actors can have a arbitrary but fixed number of consumed or produced tokens for each edge, and firing times are constant. Analysis options contain schedule, deadlock and buffer size analysis. Due to the static nature of the SDF schedule, the firing pattern will form a repetitive pattern, or iteration, after each iteration, a deadlock-free SDF model is at the same state. With these properties, an SDF-based model allows for synthesis based on a static schedule. When data-dependent choice is not required in the model, SDF-based models are sufficient to model the applications.

2.2.3 Homogeneous Synchronous Dataflow

Homogeneous Synchronous Dataflow models are a subset of SDF models. Within an HSDF model, all token production and consumption rates are equal to one. While this simplifies the analysis, the expressiveness is severely limited with this.

2.2.4 CSDF

Cyclo static dataflow (Bilsen et al., 1996) allows for variable schedule based on a predefined repeating sequence. No data dependent pattern is possible, keeping the system statically schedulable. Some additional expressiveness is available due to this pattern based variations in the actors.

2.2.5 VRDF/VPDF

Variable Rate Dataflow (Wiggers et al., 2008) allows for actors with predefined variable production and consumption rates. This allows for data dependent data rates. VPDF extends this in a CSDF-like way with different phases for each actors. Multiple different predefined variable production and consumption rates can be defined for each actor, with the actor state being in one of these predefined rate settings.

2.2.6 BDF

Boolean Dataflow (Buck and Lee, 1993) allows for an additional actor type implementing an boolean choice between two output actors. Tokens are produced on one of the two edges. The output edge depends on the value of the control token consumed by the boolean-controlled actors. This provides the model with a limited way to influence the behaviour of the models.

2.2.7 SADF

Scenario-aware dataflow (Theelen et al., 2006) allows for a fully dynamic schedule, with switching based on special detector actors. The additional detector type actor allows for changing state based on a combination of a state machine and the received token. Special control edges, originating from the detector actors, allow for propagating a state change to different actors. The control tokens from these edges are consumed by the actors befor processing the regular tokens. Both token consumption and execution time is determined by the control token consumed.

While analysis is limited by the dynamic scheduling nature, it is possible to reduce the model to a set of different SDF-type models, based on the different states reachable. When the models are limited to strong consistency and strong dependency, similar analysis to SDF-based models is applicable.

An important aspect of scenario-aware dataflow (SADF) based graphs is the explicit sequence of scenarios possible represented by the state machines associated with the decoder actors. This is not available with other data dependent dataflow MoCs such as VPDF and DDF.

2.2.8 FSM-SADF

Finite State Machine Scenario Aware Dataflow (Stuijk et al., 2011) are less expressive compared to SADF in that data rates can be varied once per iteration. It is based on the SADF MoC with the limitation that the system is only allowed to switch states at the beginning of the SDF iteration. Each iteration, a state switch happens, influencing token rates and firing times. While technically data dependent choice is available, state transitions are modelled with a probabilitstic model.

2.2.9 Dataflow analysis

The dataflow analysis as used here focuses on consistency checks, deadlock checks and latency analysis.

- *Consistency*: A dataflow graph is consistent if on each edge, in an infinite firing, the same number of tokens are consumed as produced.
- *Repetition vector*: The vector describing the number of repetitions for each actor that will cause the number of tokens produced and consumed on an edge to be equal.
- *Deadlock Free*: An dataflow is deadlock free if sufficient tokens are available such that all actors are able to fire a number of times equal to their repetition vector entry.

Consistency checks are important to prevent either deadlocks or token accumulations in long-running models. Depending on the dataflow MoC, different techniques are available to verify whether a graph is consistent, based on a set of balance equations (Lee, 1991), (Buck and Lee, 1993), (Theelen et al., 2006) The null space of the balance equations has a nontrivial solution when the graph is consistent. This nontrivial solution is the repetition vector of the graph.

A model is deadlock free when sufficient tokens are available on every cycle of actors in the graph to let every actor execute a number of times equal to their repetition vector. A consistent dataflow graph is not necessarily deadlock free, a consistent graph does not necessarily have sufficient tokens available to prevent deadlocks. After each execution cycle, the token distribution is at the initial distribution. A deadlock-free graph is not necessarily consistent as a graph accumulating tokens can be deadlock free but is not consistent.

2.3 Platform Meta Model Designs

Platform models are already extensively used on computer systems where no enumeration of connected components is available, often the case with embedded systems.

Device Tree (Likely and Boyer, 2008) is used on the Linux operating system when the hardware does not support enumeration. While enumeration is available on platforms containing a BIOS, platforms such as PowerPC require a description of the hardware to provide the kernel with the hardware configuration. Device Tree offers a tree based system containing the layout of the hardware with all properties required to initialize the proper drivers. It offers a human readable format supporting a hierarchical and reusable format which is compiled into a binary representation when deployed.

Mbed OS (Arm Limited, 2019) follows an hierarchical structure separating the SoC from the platform config code. Configuration can be specified at multiple levels with the target board as the highest level. Boards can contain configuration for peripherals and supported features.

Both of these frameworks succesfully separate function and configuration from eachother. They also succesfully use a hierarchical description of reusable components. However, their application is strictly geared towards a single application and their definitions are not aimed at reusability by other projects.

3 Analysis

Identification of the required features for modelling control systems must be explored before application requirements can be formulated. First modelling and analysis requirements for control systems must be explored. This provides the groundwork for the requirements of the application. Based on the same requirements, the explored dataflow models of computation are compared and one is selected as model backend for the application.

3.1 Control System requirements

For the system to be of use to the control engineer, some requirements have to be explored first. These requirements are necessary to allow the control engineer to use the software for control system design. Requirements can be categorized:

- *Modelling*: The expressiveness required for modelling control systems.
- *Analysis*: The set of analysis features required for verification of the non-functional parts of the control system.
- *Simulation*: The features required in the simulator for verification of the functional aspects of the control system.

3.1.1 Modelling Requirements

Some degree of expressiveness is required in the modelling system to allow the control engineer enough expressive freedom to design the control system. Limiting this too much obstructs the control engineer making the software unusable. First the requirements from control system perspective are enumerated. These are then translated to requirements for the modelling language in Chapter 4.

Identified control system modelling expressiveness requirements are:

- *Function blocks*: It must be possible to model function blocks such as multipliers or integrators or more com. The components are the basic building blocks of the control system and provide the computational aspect of the models
- *Component interaction*: As a basic requirement it must be possible to model interaction between different components of the system. This allows for making complex networks of components interacting to create a control system and model the communication aspect of the control system.
- *Runtime reconfiguration*: Control systems could require multiple operation regions or configurations. Having a way to model behavioural change based on model values allows for expressing choice. The co-ordination can be used to model multi-agent systems, safety layers or startup and shutdown effects. Two types of runtime reconfiguration can be distinguished, influencing configuration parameters of a function block during execution and influencing the active processing path within multiple parallel processing paths. Both are required for modelling control systems.

3.1.2 Model Analysis Requirements

Verification of the supplied model is required on non-functional aspects to confirm that the control system will not stall during execution.

Latency analysis allows the end user to investigate the influence on the dynamic behaviour of the plant (Jensen et al., 2011).

- *Deadlock*: It must be possible to analyze the system for deadlock problems. No guarantees are possible with a system where a deadlock situation is possible in the control flow.
- *Latency*: As latency determines for a large part whether the system is stable, it must be possible to view and analyze the latency between different components of the control system. This requires that an actor execution schedule for the system must be constructed. Sufficient knowledge of the target platform by the software is required for the construction of the schedule and the following analysis.

3.1.3 Simulation Requirements

Part of the model verification requires simulations of the model. This allows for determining the response of the model to different situations and determine the functional performance of the control system (Jensen et al., 2011). A number of aspects are required to facilitate the verification of the models for the end user:

- *Plant integration*: Integration of continuous-time plant models via a co-simulation interface provides the end user with a way to support the design flow. Without this integration, the models can not be verified against scenarios representative for real-world usage and will hamper a first-time right realisation (Broenink and Ni, 2012).
- *Unit aware*: The simulator must allow the end user to take advantage of defining units for different signals (Broenink and Broenink, 2018). This provides additional details to the models to verify signal relations.

Simulation should be possible with multiple levels of detail, depending on the amount of information provided by the platform model. This allows for a trade-off between simulation speed and accuracy for the selected platform.

3.2 Dataflow Model of Computation

The selected dataflow MoC must provide the required expressiveness and analysis options suitable for modelling control systems. The requirements described earlier must be satisfied by the MoC. All dataflow models of computation model systems based on a collection of actors and edges between the actors, effectively satisfying the function block and interaction requirements. Runtime choice, deadlock analysis and latency analysis differ for the different models of computation.

Model of Computation	Deadlock analysis	Latency analysis	Runtime path selection	Runtime reconfiguration
SDF	✔	✔	✘	✘
CSDF	✔	✔	✘	✘
VRDF	✘	✘	✘	✔
VPDF	✘	✘	✘	✔
BDF	✔	✔	✔	✘
FSM-SADF	✔	✔	✘	✘
SADF	✔	✔	✔	✔

Table 3.1: Requirements versus the different dataflow MoC options

Shown in Table 3.1 are the different dataflow models of computation. Two dataflow models of computation can be selected from the table. The first case when no runtime choice is required, SDF as MoC satisfies the requirements. While CSDF appears to be suitable for this case, it allows for variable consumption and production rates, adding complexity where it is not necessarily required.

When runtime choice is required, SADF satisfies all requirements. To allow for deadlock and latency analysis, the SADF models however need to be limited in their expressiveness. Notably any detector actor must fire only once every cycle. Still with this limitation in place, SADF offers the flexibility required by the control system requirements while retaining the required analysis options.

When runtime choice is not required for the model, the SADF-based graphs can be simplified to SDF graphs without any effort. The main advantage of using SDF instead of SADF as MoC is that it allows for a static scheduler when realizing the model. Simplifying an SADF-based graph to SDF is possible when no detector actors are present in the model.

3.3 Platform Meta Model

The PMM gives a description complete enough to generate the PSM. It needs to describe the hardware platform in a code-independent way, and provide the information required for both the PSM and the generation of the final platform-specific code. The platform description domain-specific language (DSL) should allow for reuse of components, component timing, and performance specification, furthermore, it must specify a target execution architecture. The information can be categorized into code-generation assisting information and timing information. The code-generation related information allows for converting the PSM into the platform-specific code. Timing information is required for analysis and simulation of the PSM, while essential for analysis, it is not required for the code generation.

3.3.1 Platform Meta-Model Components

The PMM describes a contained and interconnected set of hardware providing sufficient details for a refinement of the PIM to a PSM. The PMM must allow for describing the required information to allow for sufficient timing analysis on the PSM. It provides a worst-case execution time and a value accuracy specification of the processing units. This allows for more detailed simulations using the additional hardware specification from the platform model on the system model.

A hierarchical modelling structure describing the components of a hardware platform is required to model the complexities of hardware platforms. Hardware platforms consist of multiple components, the physical components on a platform acting on the data, interconnected to complex structures sometimes containing another different platform as subcomponent. For example, the RaMstix board[1] contains an FPGA and a Gumstix platform.

[1] https://git.ram.ewi.utwente.nl/ramstix/ramstix

Different types of components can be identified to simplify the PMM into a set of component-type specific meta-models. Components can to be separated into execution models such as a SoC or FPGA and peripheral based components such as sensors, actuator and communication modules. A short description of each identified component type follows here:

Sensors: The sensor component type provides the properties of different available sensor types. The sensor acts as a boundary between the continuous-time plants and the event-driven function blocks. Main attributes of a sensor are measurement interval, accuracy, and delay. A sensor requires at least one event-driven input to trigger the sensor and one analog input to sample values from.

Actuators: The actuator component type provides the properties required for modelling different actuator types. It acts as a boundary between the event-driven function blocks to the plant models. Attributes used here consist of update interval, accuracy and delay. Actuators require at least one analog output and at least one event-driven value input.

Compute modules: These modules consist of components such as microcontrollers and FPGAs. They allow for allocating multiple function blocks on them to be scheduled within the same processing unit.

Plants: A component representing the dynamic system of which the behaviour is steered by the control system. The plant models can include additional components to aid the plant connectors such as built-in actuators or sensors.

From the components described above two structures can be derived:

Boards: Boards represent components containing multiple interfaces and one or more compute modules. A board can inherit (contain) different other boards, modules and components. The structure of a board allows for a hierarchical description of the components.

Platform: A platform is the final structure consisting of at least a single board. The platform defines the full configuration for a single model and is to be used as a target for a model. A platform can include plants.

Interfaces provide the meta-model framework for describing ports on different modules. The interface provides the additional information required to model the properties of the interconnection between the components. Different types of interfaces provide different properties here and the type of interface is dictated by the physical implementation of the component.

An example platform model is shown in Figure 3.1. It consists of a platform containing connecting a board component with a plant component. The board contains a compute module, a sensor and an actuator. In the example, the different components are named (shown between parentheses) after their specific component types. Interfaces provided by the components are used to connect the components with each other.

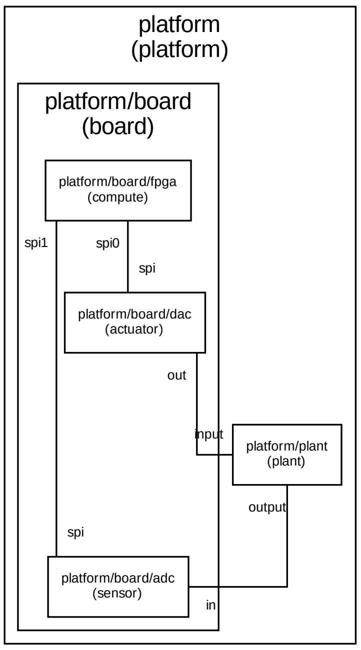

Figure 3.1: Platform model containing a plant and a board. The board contains a compute module, an actuator and a sensor connected to each other via their interfaces.

4 Design Requirements

A large part of the work consist of a tool design as a proof of principle for the proposed workflow and design paradigms. Both the application itself and the input files required for the application must adhere to a set of requirements. The requirements formulated here are based on the research questions and analysis results.

4.1 Application design

The application requirements are split into a number of different categories.

- The model backend specifies the requirements for the dataflow based model processing.
- The model analysis builds upon this and specifies which analysis is required for the models for verification of the models.
- The simulation requirements specify the requirements on the simulation of the models and which results are expected from the simulations by the engineer.

4.1.1 Model backend

Requirement 1: *The application must use scenario-aware dataflow as dataflow MoC*

The choice in model backend dictates the restrictions on the modelling freedom and which analysis is available for the models. Based on the exploration of the different dataflow MoCs, SADF fits the current approach. As discussed in Chapter 3, It allows a limited notion of choice within the model. With the limitations preventing Turing-completeness, it allows for sufficient analysis of the models. Having this in the application allows for supervisory control by modifying the state of actors during run-time. The consistency and deadlock analysis from SADF can be used on the models for formal verification of these aspects. While it does not put any guarantees on the functionality of the model, it allows for proving that the model will not deadlock at some point in time.

Requirement 2: *The application must be able to operate on physical quantities*

It must be aware of the relation between different quantities and units of measure. The advantage is that it allows for expressing models with the physical quantities included, increasing the ease of use for the end user by removing the need for converting measurements and setpoints to opaque values. The addition of physical quantities to the model adds another layer of verification by restricting operations to adhere to the rules of dimensional analysis.

4.1.2 Model analysis

The goal of the model analysis is to verify a number of essential properties of the model. This to ensure that the modelled system is able to execute indefinitely without running into memory constraints.

Requirement 3: *The application must be able to formally verify whether the dataflow equivalent of the model is consistent*

Without this verification, a cyclic dependency will either deadlock or use unbounded FIFO space. If this is the case, the model can not be run indefinitely and will crash the execution platform either by deadlocking or by consuming too much memory.

Requirement 4: *The application must be able to prove that the supplied model is deadlock free.*

The second requirement is that every cycle must be proven to have enough tokens available to be deadlock free. Together with the proof that the model is consistent, this ensures that the model can run indefinitely without deadlock. This is essential for long running applications.

Requirement 5: *The application should be able to prove whether clock elements are the limiting factor in throughput for a model*

This requirement allows the application to formally verify that token throughput of the system is limited by tasks representing periodic clocks in the models, in other words, all components in the system have at most an execution time equal to a clock element. When the token throughput of the dataflow model is not limited at the clock, it is limited at a component too slow for the configured clock frequency.

4.1.3 Model Simulation

Requirement 6: *The simulation must allow for verification of the functional aspects of the model*

One of the goals of the application is to allow the engineer not only analysis of the models, but also verification of the functionality of the models. This is essential for a performance-based comparison between the PIM and the PSM. To allow for this, the simulator must be able to produce results based on an execution.

Requirement 6.1: *A computational model must be supported by the simulator for function blocks*

For the model to simulate signals and events, a computational aspect is required to calculate values of the produced tokens. Detector function blocks depend on the token values for the selection of the emitted state token. This adds a hard requirement for including computational models in the dataflow-based actors.

Requirement 6.2: *A time-series output of one or more signals from a simulation must be available as output.*

Visualization of time-series allows for a convenient way to show the engineer signal traces from the simulation run. This helps with comparisons between different configurations of the model and allows for verification of the model functionality.

Requirement 6.3: *A graph of execution times of the different model components must be available as output.*

Within a complex model, it might not be directly clear how component dependencies are structured. An overview of actor execution times allow for visualization of component dependencies and timing schedules. It allows for identifying components with a significant impact on the performance.

Requirement 6.4: *It should be possible to include computational models for plants in the system*

Including support for plant-representing models in the tool adds significant value. It allows for more in depth confirmation of the functionality of the model by including the plant in the simulation. However, plant models do not convert properly to SADF semantics. Foremost as continuous-time models are not suitable for conversion due to the event-based nature of Dataflow. To allow for simulation including these continuous-time models, either the edges to and from plants require adaptation to facilitate the conversion, or the plant models themselves require to be adapted to event-based models.

4.1.4 Model-to-model conversion

The goal of the model-to-model conversions is to refine the supplied PIM to PSM.

Requirement 7: *It must be possible to specify a component for each system-model function block as target for conversion*

Converting platform-independent function blocks to a model representing the physical component on a system requires information on this conversion. One of the steps is to have a specification of function blocks to component mappings, otherwise the engineer has no control over the distribution of functionality over the physical system. Depending on the type of the function block, the specified component must be restricted to a compute component, a sensor, or an actuator.

Requirement 8: *The conversion must be able to adapt the execution times and the resolution of the function blocks to a value representative for the matching physical components.*

By adapting the execution times the function blocks match the worst-case execution time on the specified hardware. This can be an estimated worst-case execution time, in the case the component is only able to guarantee a maximum execution time, or a precise duration when the platform is able to guarantee an exact execution time.

Requirement 9: *The conversion must be able to adapt the resolution and bounds of output values of a function block.*

Depending on the hardware specification, output values of the function must be limited in resolution and/or bounded between minimum and maximum values. This to represent conversion hardware with limited resolution such as D/A- and A/D-converters

Requirement 10: *The modelled connection between the mapped hardware components must be included in the model.*

Depending on the specified hardware communication interface, the communication latency between two components is significant enough to influence the system behaviour and must be included in the PSM. The non-zero delays caused by communication systems can significantly influence the performance of the system.

The generated PSM must allow for identical analysis as the PIM, to allow for performance comparison between the PIM and different PSMs This includes a generated dataflow representation of the PSM to show the generated data dependencies between different components and identify performance limiting components.

4.2 Language design

The input of the tool consists of descriptions of the system models and the platform models. These model files contain detailed specifications of the models, with specifications such as component specification and interconnections. This can be implemented by a structured domain-specific language for describing the models.

Requirement 11: *Both the model language and the hardware description should be both human and machine readable*

This allows for both writing models manually as well as generating or converting them from an external application. Extending the tool becomes an exercise in generating the model files. While it is shown that it is possible to have a human readable format compiled to a machine readable format, it is chosen to keep a single format for simplicity instead of requiring an additional conversion application. A middle ground between machine readable and human readable is thus required.

Requirements for the model description and the platform descriptions are similar in their requirements, both must specify blocks with identifiers, properties and connections.

4.2.1 Model description

These requirements specify the functionality of the system modelling language required.

Requirement 12: *The model description must allow for describing a collection of function blocks including configuration.*

To satisfy the re-usability of components, the language must be able to specify a function block type. The model description must be able to specify configuration parameters of a function block, as function blocks must be reusable with only requiring modification of the configuration of the function block. Some of these configuration parameters can be optional, others mandatory, depending on the function block.

Requirement 12.1: *The model description language must be able to specify connections between function blocks including properties of these connections.*

Not only must it be possible to specify the relation between the function blocks, it must be possible to specify a set of initial values for an function block input, otherwise it is impossible to create a loop without deadlocks.

4.2.2 Hardware description design

The hardware description is on a high level similar to the model description.

Requirement 13: *The description language must allow for describing physical components with specifications and interfaces.*

The hardware description must contain a specification of the components, describing the parameters of the components. The goal is to be able to directly copy them from the component vendor datasheets.

Requirement 13.1: *Component interconnections must also be part of the specification.*

Without this, the relation between different components of a platform can not be specified. To transfor a model, information about the relation between the different components and which communications interface is used must be part of the specification.

Requirement 13.2: *it must be possible to describe a component as a collection of components.*

This requirement facilitates the reuse of component descriptions. For example, a board with an FPGA and a number of sensors and actuators must be specified as a component containing include-like statements for the separate FPGA, sensor and actuator specification.

5 Implementation

The tool created, Deimos, consists of a collection of functions to model, convert, analyze and simulate complex dataflow based systems. Deimos is named after the second moon of the planet Mars.

First the approach of how systems are modelled with Deimos is described, describing the function blocks and describing how the model files are designed and processed around these function blocks. Later the simulator itself is thoroughly explained. At last the conversion method from a PIM to a PSM is described.

5.1 Function blocks

Function blocks are the components of which the system is build upon based on the system model. Each function block, when instantiated, represent some functionality of the system. For example, this can be computational operations executed by the controller or some interface providing conversion between data formats. The building blocks of the models contain computational functionality to provide not only a timing analysis but also functional simulation of the modelled system.

When instantiating function blocks, Deimos can use multiple locations to look for the specified function block type. This ensures an extensible library where function blocks specific for a single model can be added separate from the main library. A function block can be represented by either a single dataflow actor, or as a composite block using multiple dataflow actors. Every dataflow actor representing function blocks is automatically created with a self edge to prevent unwanted concurrent execution of the actor.

A single function block requires the information contained in the supplied model and an implementation to function. A function block requires a number of properties:

- Inputs and outputs
- Data type
- Function block instantiation
- Computational functions
- Parameters

These property requirements can be split into communication, configuration and computational properties.

5.1.1 Communication

The input and output ports of a function block dictate which connections the function block can have with other function blocks within a model. All input ports must be connected to an output port, but not all output ports within a function block have to be connected. Output ports that do not have a connected input port discard their produced tokens. Output ports that have multiple connected input ports duplicate their produced tokens to all connected input ports. It is valid for an input to be connected to the output of the same function block, however care must be taken to ensure that enough initial tokens are available on the connection as not to cause a deadlock. As an exception to this, the input for state tokens is only added and required when states are used with the function block. As to adhere to the dataflow model, function blocks are executed by the simulator as soon as all inputs have the required number of tokens.

5.1.2 Configuration

Parameters provide the configurational aspect of the function block and provide the flexibility in the models. The parameters directly determine the required set of parameters in the system model, from which the values can be requested at run-time. Function blocks can parse a parameters both as plain value or with interpreted physical dimensions.

Parameters can be supplied with dimensionality included such as lengths, time, speed, current and voltage. Calculations using these parameters will propagate the units and will detect errors when the operands of an operations have incompatible units. For example, it is possible to sum different lengths such as meters, centimeters and inches. In this case conversion will happen automatically. However, adding units with different dimensionality will result in an error condition.

5.1.3 Computational functionality

The computational functionality is added to a function block by implementing at least two functions of the python model class. This allows for the model to add computational aspects to the system model, allowing verification of the functional aspect of the system.

First is the initialization of the function block. While not strictly necessary, it allows for functional verification of the parameters and the instantiation of the initial values.

Second is the `step` function, handling the computational aspect of the model. It calculates the output of the function block based on the inputs and the state of the function block. The `step` function is supplied with the current start time and the input tokens. Based on these arguments, a finish time and a new set of output tokens is produced and returned to the simulator.

For example, the multiply-accumulate function gathers the input tokens from each input, multiplies them with the respective factors and sums the results. The finish time of the function is calculated by adding the worst-case execution time to the start time. This result is returned to the simulator. For this to work, the input tokens multiplied with their factors must all have the same dimensionality, otherwise summing them is not possible.

5.2 System meta-model

The system meta-model (SMM) provides a structure to describe system models as a system of interconnected function blocks. It must provide a framework to desribe the function blocks as described above with all required options. The meta-model designed for the systems is based on the both machine and human readable YAML Ain't Markup Language (YAML) specification (Ben-Kiki et al., 2005). This allows for models written directly by the engineer or generated by external tools. The YAML specification satisfies requirement 11.

A full model is based on a description of a set of interconnected function blocks. These function blocks define the different components of the system. Based on the input model description, the tool instantiates the function blocks configured and connected as specified by the model.

The SMM requires a number of values to instantiate a function block from:

- Name: An unique name to identify the function with. Used as an identifier to distinguish functions within the model.
- Type: The type of the block, used to identify and select the computational model for the function.
- Parameters: A dictionary of parameters to configure the function. The required parameters depend on the computational model of the function.
- Inputs: A dictionary of inputs, used to specify connections between the blocks. The inputs specify to which output port a specific input port of a function is connected. It also allows for specifying a set of initial tokens for the input port.

An example function specification of an proportional–integral–derivative (PID) controller is shown in listing 5.1. It shows a function block named `PID` using the computational model of the type `pid`. The parameters of the PID controller are specified, including the worst case execution time (the `wcet` parameter defined as `0ms`).

```
PID:                       # Name of the function block
  type: pid                # Model identifier
  params:                  # Configuration parameters
    kp: 0.2
    ti: 0.5
    td: 0.1
    beta: 0.2
    wcet: 0ms
  inputs:                  # Input port connections
    in:
      source: Error.out
  states:                  # Control states
    safe:                  # `safe' state
      params:              # Config parameters to
        ti: 0              #  override with this
        td: 0.0001         #  state
```

Listing 5.1: Example function block specification of an PID controller

The behaviour of a function can be influenced at run-time by signalling states to the function. When used, every execution requires a token specifying the execution state of the function. This state is used to override the parameters of the function to the state-specific parameters. Special detector based function blocks emit these state tokens to influence the run-time behaviour of the model.

5.3 Plant integration

Plant models can be added to the system model as a functional block representing the plant. Similar to regular function blocks they have inputs and outputs, but opposed to the regular function blocks, plants can not be modelled with an SADF model.

Two types of plant models are available for use, a continuous-time transfer function based plant and an external plant importer. The transfer function based plant allows for adding simple continuous-time functions. It does not have advanced capabilities, it allows however for rapid development to the plant during iteration cycles as it does not require an externally created plant model. The external import based plant model is using the Functional Mock-up Interface (FMI) specification (Blochwitz et al., 2012). Functional Mock-up Unit (FMU) based files can be specified and integrated into the model, their inputs and outputs can be connected to other function blocks.

The plant configuration requires specifying for each input and output the physical dimensions required by the port.

The main challenge with plant representing function blocks is that a plant does not adhere to SADF semantics. Analysis and simulation are adapted to take this into account. When modelling the pure SADF representation of the model, plants are omitted from the model.

5.3.1 Model rendering

A full system model as described by the model files can be displayed as a collection of function blocks with connections. Deimos includes options for rendering a block representation of the model. This allows for initial testing of the system model description, by allowing the engineer compare the generated model properties and connections with the expected system. Figure 5.1 shows the output of such a render for a simple PID based control loop with a plant included.

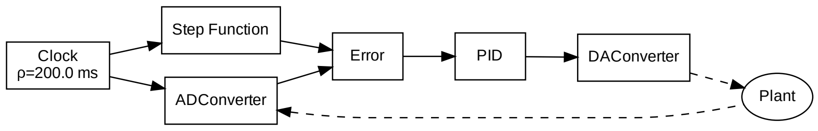

Figure 5.1: Block diagram of a PID based control loop

5.4 Model analysis

The SADF model as constructed internally by Deimos allows for verification of a number of properties. Before any simulation is done on the model, it is first checked whether the model is consistent and deadlock-free.

The analysis is done using an implementation of the definitions and theorems applicable to SADF graphs (Theelen et al., 2006). Required for this analysis is that the dataflow representation of the model is both strongly dependent and strongly consistent. This requirement allows for formal verification of the absence of deadlocks within the model.

The implementation first determines the non-trivial repetition vector of the SADF graph. For this, the full matrix of token consumption and production for every tuple of actors and the states of that actor is build. With this matrix, the repetition vector of the graph can be solved from the matrix.

For this to hold it is required that the production and consumption rates of tokens only depends on the parameters of a function block. The rates must not be time-dependent or dependent on an internal state of the function block.

Within this analysis, plants are excluded, this as they do not follow dataflow-based simulation rules, but also do not influence the activation condition of the dataflow-based actors. They can be excluded from the model analysis, a plant will never block the execution of a actor connected to a plant. Within the analysis, the model from figure 5.2 is converted to the model from figure 5.3.

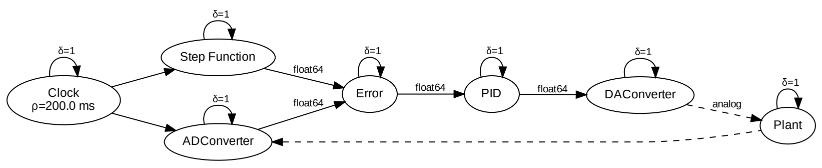

Figure 5.2: SDF model of a PID based control loop extended with analog components

5.4.1 SDF based models

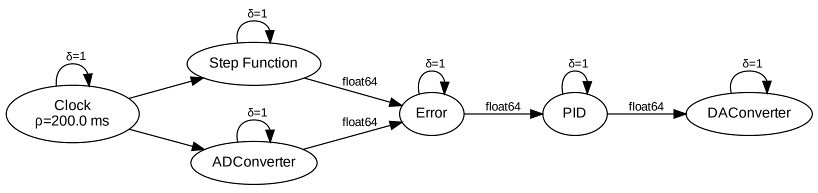

Figure 5.3: SDF model of a PID based control loop

5.5 Model simulation

Simulation of the system is based on SADF token interaction between the different function blocks. The simulator works as a discrete-event based system, executing function blocks from an execution queue. The main advantage is that this results in a fast simulator design with a flexible time step based on the granularity of the actors. Function blocks are always executed in sequence and causal in time.

The simulator maintains an internal time stamp which contains the simulator time stamp of the last started function block. To determine the next function block to simulate, the execution queue is sorted and the function block that has the lowest start time stamp among function blocks in the queue is executed.

After a function block is executed, every connected function block is checked for readyness. Every function block that is determined to be ready is added to the execution queue.

The start time of a function block is based on the production time of the token that caused the function block to become ready. At the simulation start, all function blocks are checked for readyness to set the initial execution queue.

The simulator works with a number of rules on which the simulation algorithm is based

1. A function block is considered ready when all inputs contain at least the minimum amount of tokens required for the function block to execute.
2. For every function block p, the simulator maintains a boolean indicating if the function block is ready to execute.
3. A function block that is executed has an associated execution start time t_{start} and an execution stop time t_{end}
4. A function block p has a possible empty set of function blocks R such that there is an output from p to at least one input of every function block in R.
5. After a function block p has fired, for every function block p' in R it is checked if the function block is ready. Every function block

that became ready receives a start time t_{start} equal to t_{end} of p and is added to the execution queue.

6. The function block in the execution queue with the lowest value for t_{end} is executed next.

While it is not required to simulate actor execution in order of their lowest starting time, it is required for the synchronization with an external simulation.

5.5.1 Co-simulation synchronization scheme

To allow for a simulation with a continuous time based plant, a co-simulation interface is available within Deimos. The interface, based on the FMI specification, requires synchronisation with the event-based simulation for proper interaction between the models.

Synchronization between the event-based system model and the plant is based on a lock-step simulation using token interaction to exchange state between the models(Nicolescu et al., 2007). The value of the token is always set to the output value of the plant at the time the function block consumes the token. This ensures that a function block consuming tokens from the plant always receives the output of the plant at the time the value is used, removing the FIFO-based nature of dataflow edges.

The token interaction between the system and the plant is not based on SADF, the non-blocking nature of a continuous time model does not fit the SADF modelling restrictions. A modified behaviour is used where the plant is always able to consume an infinite amount of tokens and always has tokens available for the component consuming from the plant. This way, a functional block representing a digital-to-analog converter is always allowed to specify input values to the plant. Furthermore, an analog-to-digital converter can always sample a value from the plant by consuming a token and reading the value from the token.

Tokens supplied to the plant behave as a zero order hold between the arrival time of the token and the arrival time of the next token. This is used to simulate the behaviour of a simple digital-to-analog converter. When a token value originating from the plant, the plant is simulated up to the consume time stamp of the token. By ensuring that function blocks are always processed in the order of activation, all tokens used as input of the plant are available. This removes the requirement for a roll-back scheme in the simulator and allows for the relative simple lock-step based scheme.

One of the limitations originating from the SADF MoC approach is the inability to model asynchronous external events. This is extended to the co-simulation interface, the approach does not allow for receiving an asynchronous event (interrupt) token from the plant.

5.6 Simulation Results

The simulator records results in a machine readable file format for post processing. Function-block execution times and token values are stored in the file. This allows for postprocessing by both Deimos and external tools.

Built-in are options for plotting the output values of function blocks over time. Multiple time series can be plotted and are split out over different graphs based on the dimension of the values.

Another option is to plot the start and finish time of function block executions, visualizing the delays caused by different function blocks.

5.7 Platform Meta Model

The PMM offers a structure to describe the hardware of a full physical system. The meta model includes options to describe components, their specifications and the interconnections between different components. The implementation allows for describing the models as a collection of components distributed over a number of different files. With the properties described in a platform model, a model can be converted from a platform-independent model to a platform-specific model.

First the PMM is described, later the way the specification interacts with the model-to-model conversions.

The main meta model only specifies a schema for the top level properties. Modules must contain a number of properties describing the component

- The name: a human readable descriptor of the component.
- The type: describing the component type, for example a sensor.
- The variant: indicating a specific component variation.
- The specification: the properties allowing for modelling the component.
- Interfaces: describing the interfaces used to build interconnections with the component.

The PMM defines a number of component types: boards, actuators, sensors and compute modules. Each type allows for describing a different set of properties related to the function of the type. The exact specification required to model the full component depends on the component type and variant. For example, two different digital-to-analog converters require both a sample rate and bit resolution, but can differ in the types of delays required, depending on the interfaces provided.

Each non-board non-platform component has a variant. The variant indicates which specific implementation of the function block must be used to represent the component. This allows to select a different model representation of the function block and thus requiring a possible different configuration of delays and data dependencies. With this the components with a similar function but with different architecture can be distinguished.

Specifications describe the components in more detail. It represents the configuration aspect of a combination of component type and variant. Depending on the component type and variant, different specifications are required to model the component. As an example, an ADC connected via the SPI bus is using the `spiadc` variant, therefor uses the ADC model of the `spiadc` variant, requiring properties such as the maximum SPI bus frequency.

Interfaces allow for specifying the physical input and output interfaces of a component. Each interface has a name, a type, a variant The name is a unique identifier for the specific port on the component. The type and the variant identify how the interface must be modelled. Further specifications can be required for an interface. This is used to model interfaces such as general purpose I/O and communication buses.

5.7.1 Component types

Different platform component types are defined to allow modelling the different properties of each component while maintaining a common set of properties over a single component type.

Compute: The compute model allows for defining a component with a computational function. It represents physical components such as microprocessors, microcontroller and FPGAs. It allows for accommodating function blocks with a purely computational aspect. Multiple components can be scheduled on a single compute module at a single time. The scheduling algorithm is left up to the implementation specifics of the compute module. Specification of the compute module is defined by an indication of the computational speed of the platform. With an FPGA, the speed can be expressed as a frequency, with microcontroller based platform not only the clock frequency but also the architecture of the system is important.

The compute module allows for specifying the speed of the platform and a set of interfaces. Depending on the variant, the speed is either used as an indication of the platform performance or used as a precise delay multiplier. For example on an FPGA-based variant the clock speed can be used as an accurate indication of the speed while on a processor based component the effective duration of a function is also dependent on the architecture and load of the processor.

Sensors: Sensors represent a physical measurement device on the platform. The specifications of the sensor are geared towards defining the accuracy and sample rate of the device. This includes the latency between triggering a sample and sampling a value, and the propagation delay from the sampling to the output of the sensor. The limited sample rate of the sensor can be included in the model by limiting the rate at which tokens triggering a conversion can be consumed. A sensor requires at least two interfaces of which one must be an analog interface and the other a regular (digital) interface.

Actuators: Actuators represent the devices on the system responsible for the controlling mechanism of the system. The conversion of signals from event-based to continuous time is an essential part of the actuators. They accommodate the function blocks that convert data from the ‘regular’ edges to an analog edge.

Boards: Boards provide the meta model with a way to group collections of components. The board does not allow for function blocks to be assigned to it, instead it allows for defining a set of sub-components with their interconnections. By allowing a platform or board type model to include different component files, including other boards, both hierarchical descriptions and reuse of component model files made is possible.

As a board itself does not define a component, but only includes other components, any interface from the board is an interface from a component on the board. A set of interfaces from components on the board can be exposed for use outside the board. This represents a physical connector available on the board itself. The following listing shows an example connector on the board, indicating the `expose` type to indicate that it is a subcomponent from which an interface is made available.

```
interfaces:
  dac0_analog:
    type: expose
    name: dac0_analog
    specification:
      passthrough: dac0/out
```

Listing 5.2: Exposed 'analog' interface of the component named 'dac0'

A board is also able to connect interfaces from different subcomponents. Separate specification indicated with `connections` indicate which component interfaces are connected together. For example, to connect the SPI type interface from two different components the following specification is required:

```
connections:
  - from: fpga/spi0
    to: dac0/spi
```

Listing 5.3: Specification of an interconnection between the first SPI interface of 'fpga and 'dac0'

The meta-model allows for defining a component hierarchy of which a platform is always the top level entity.

Plants: Plants are included in the PMM to allow providing details on the connected plants of the model. The main reason is that mapping a simulated plant on the PMM allows for verification of the connections between the actuators, sensors and plants. It removes ambiguity in how a plant is connected to the setup.

Platform: A platform describes a target for a model-to-model conversion. The minimal form of a platform is a plant and a component combined with the interconnections between the two. It is comparable to a board in that the platform does not allow for function blocks to be mapped on it. However, the platform does not allow for 'exposing' connectors as it is the most top level model component.

5.8 Model-to-model conversion

The model-to-model conversions translate the PIM to a PSM using the platform model and a mapping of function blocks to components.

Each function block is assigned to a platform component. The conversion looks up the component and replaces the function block with the variant of the function block as indicated by the component. A communications bus, while in the PIM represented by an ideal FIFO, can be replaced with a limited capacity bus with a non-zero latency.

The configuration of the function block from the PIM can be overridden. Based on the platform component specification, parameters such as worst case execution time and resolution are replaced to match the platform. The exact details of this conversion, how the configuration is replaced, is left up to the function block variant. This allows the function block to match a possible code generator for each supported variant type.

The result of a model-to-model conversion is a new model, compatible with analysis and simulation options described above. This model can be simulated and the results can be compared by the end user with the behavior of the PIM.

Limitations of the conversion however are present. SADF limits the model in the sense that it is not possible to model mutual exclusive access to a resource. This presents a limitation in that shared access to a communications bus cannot be modelled. Furthermore, the conversion does not take a limited concurrency scheduler in account. Actors scheduled together on a compute unit are executed in parallel when possible.

6 Testing

The implementation as described in Chapter 5 must be verified on a number of aspects. Without this verification, the results produced by the application can not be used in any form whatsoever. This verification check if the implementation satisfies the requirements as described in Chapter 4.

6.1 Model handling

6.2 Formal Model Verification

Before simulation, a model is formally verified for consistency and deadlocks. Deimos always verifies these properties before starting a simulation. The process of formal model verification is shown and verified here to shown that the application satisfies requirement 3 and 4.

The first test is a simple inconsistent model shown in Figure 6.1. Two actors, `t1` with a production of 4 tokens and actor `t2` with a consumption on the same edge of 5 tokens is constructed. The edge in the reverse direction has production and consumption equal to 1. This model is inconsistent as it consumes more tokens than produced. When the verification from Deimos is used, it refuses to simulate the model due to the consistency property missing as shown in Listing 6.1.

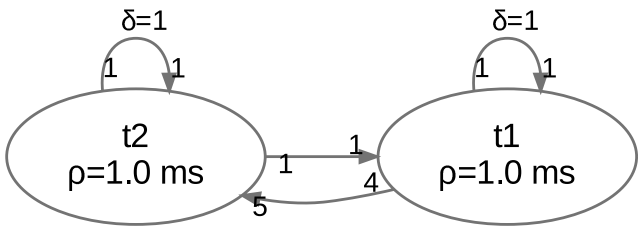

Figure 6.1: Small inconsistent dataflow model. The actor production and consumption rates are not balanced.

```
ERROR:root:Model is not consistent
CRITICAL:root:Error during model verification "model is not
    consistent"
```

Listing 6.1: Deimos output when supplying an inconsistent model. Shown is Deimos aborting the application due to an inconsistent model

An example of a consistent model is shown in Figure 6.2. This model has two actors, one with consumption of 1 token and production of 4 tokens, the other actor with consumption of 4 tokens and production of 1 token. Only 3 tokens are available where 4 are required to start actor `t2`. The repetition vector of the model is shown in Table 6.1. While the model is consistent and thus will not accumulate or exhaust tokens, the model is not deadlock free due to a missing initial token. Deimos is able to detect this for arbitrarily sized SDF cycles. Analysis of this model is shown in Listing 6.2

Table 6.1: Repetition vector of the model from Figure 6.2 and Figure 6.3.

actor	repetition
t1	1
t2	1

Deimos correctly detects a deadlock in the loop between actor `t1` and actor `t2`.

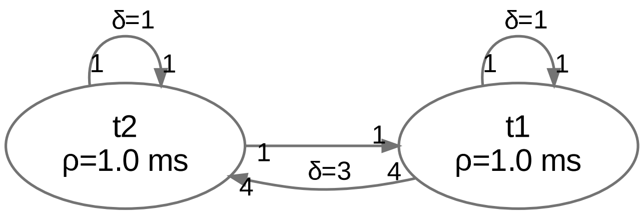

Figure 6.2: Consistent dataflow model without sufficient initial tokens to be deadlock free

```
DEBUG:root:Model has repetition vector {'t2': 1, 't1': 1}
INFO:root:Model is consistent
DEBUG:root:Cycle: ('t1',)
DEBUG:root:Building t1 to t1 with cons 1 and prod 1
DEBUG:root:Loop has tokens
DEBUG:root:No deadlock detected at cycle ['t1']
DEBUG:root:Cycle: ('t1', 't2')
DEBUG:root:Building t1 to t2 with cons 1 and prod 4
DEBUG:root:Building t2 to t1 with cons 4 and prod 1
DEBUG:root:Loop has tokens
WARNING:root:Deadlock detected at cycle ['t1', 't2']
ERROR:root:Deadlock in cycle <synth.analysis.cycle.Cycle
    object at 0x7f0c8bfee5f8>
ERROR:root:Model is not deadlock free
CRITICAL:root:Error during model verification "model is not
     deadlock free"
```

Listing 6.2: Deimos output when supplying a consistent model with a deadlock. Shown is the analysis stage being aborted due to possible deadlock in the model

When setting only a single initial token on the edge with a consumption of 1, the model is consistent and deadlock free. This model is shown in Figure 6.3 and the Deimos model source is shown in Listing A.1. The repetition vector of the model is identical to the previous model and is shown in Table 6.1. This model is identical to the previous example, but only has a single initial token on the other edge. As shown by Deimos Listing 6.3, this model is consistent and deadlock free. Only after the analysis showing a positive result, Deimos starts the simulation. The timing diagram of the simulation is shown in Figure 6.4. Visible is that the pattern repeats after each actor has executed a number of times equal to the corresponding repetition vector entry from Table 6.1.

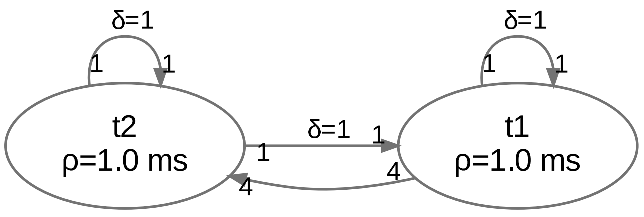

Figure 6.3: Deadlock free dataflow model

```
DEBUG:root:Model has repetition vector {'t2': 1, 't1': 1}
INFO:root:Model is consistent
DEBUG:root:Cycle: ('t1',)
DEBUG:root:Building t1 to t1 with cons 1 and prod 1
DEBUG:root:Loop has tokens
DEBUG:root:No deadlock detected at cycle ['t1']
DEBUG:root:Cycle: ('t2',)
DEBUG:root:Building t2 to t2 with cons 1 and prod 1
DEBUG:root:Loop has tokens
DEBUG:root:No deadlock detected at cycle ['t2']
DEBUG:root:Cycle: ('t1', 't2')
DEBUG:root:Building t1 to t2 with cons 1 and prod 4
DEBUG:root:Building t2 to t1 with cons 4 and prod 1
DEBUG:root:Loop has tokens
DEBUG:root:No deadlock detected at cycle ['t1', 't2']
INFO:root:Model is consistent and deadlock free
INFO:root:Simulating for 1.0 second
INFO:root:Starting simulation
INFO:root:Simulation done, exiting
```

Listing 6.3: Deimos output when supplying a consistent and deadlock free model. Shown is a positive analysis result as the model is proven consistent and deadlock free

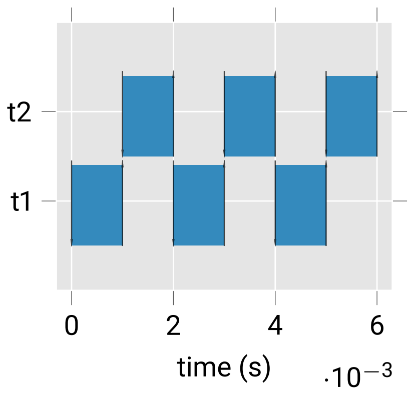

Figure 6.4: Timing diagram of the model from Figure 6.3

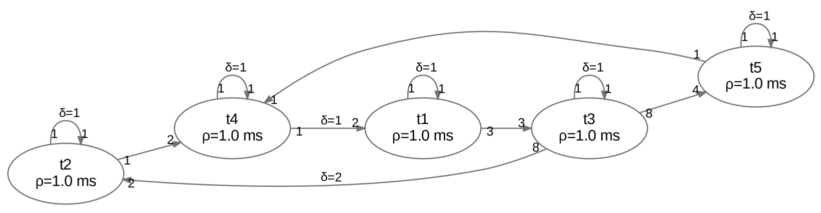

Figure 6.5: Larger consistent but not deadlock free dataflow model, there are not enough tokens on the cycle of `t1` - `t3` - `t2` - `t4`.

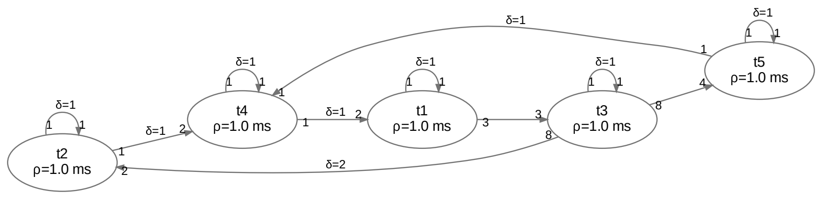

Figure 6.6: Larger deadlock free dataflow model

The analysis scales to larger models. Two similar larger models are shown in Figure 6.5 and Figure 6.6 with the source of the second one shown in Listing A.2. The repetition vector of the model is shown in Table 6.2. While both models are consistent, The repetition vector of the model is shown in Table 6.2 the first is not deadlock free due to a missing token between `t5` and `t4`. The second model does not lack this token and should thus be deadlock free.

Validating the models with Deimos shows for the first model an issue with the deadlock detection as shown in Listing 6.4. The second model is deadlock free as shown with Deimos in Listing 6.5 The timing diagram from simulation is shown in Figure 6.7. Visible is that here the execution pattern also repeats after each actor executed a number of times equal to the repetition pattern. For the timing diagram, the support for requirement 6.2 is used.

```
DEBUG:root:Model has repetition vector {'t2': 4, 't1': 1, '
    t3': 1, 't4': 2, 't5': 2}
INFO:root:Model is consistent
DEBUG:root:Cycle: ('t3',)
DEBUG:root:Building t3 to t3 with cons 1 and prod 1
DEBUG:root:Loop has tokens
DEBUG:root:No deadlock detected at cycle ['t3']
DEBUG:root:Cycle: ('t1',)
DEBUG:root:Building t1 to t1 with cons 1 and prod 1
DEBUG:root:Loop has tokens
DEBUG:root:No deadlock detected at cycle ['t1']
DEBUG:root:Cycle: ('t4',)
DEBUG:root:Building t4 to t4 with cons 1 and prod 1
DEBUG:root:Loop has tokens
DEBUG:root:No deadlock detected at cycle ['t4']
DEBUG:root:Cycle: ('t1', 't3', 't2', 't4')
DEBUG:root:t1: Production is 3
DEBUG:root:Building t1 to t3 with cons 2 and prod 3
DEBUG:root:t3: Production is 8
DEBUG:root:Building t3 to t2 with cons 3 and prod 8
DEBUG:root:t2: Production is 1
DEBUG:root:Building t2 to t4 with cons 2 and prod 1
DEBUG:root:Building t4 to t1 with cons 2 and prod 1
DEBUG:root:Loop has tokens
WARNING:root:Deadlock detected at cycle ['t1', 't3', 't2',
    't4']
ERROR:root:Deadlock in cycle <synth.analysis.cycle.Cycle
    object at 0x7fe54e18f208>
ERROR:root:Model is not deadlock free
Result: model is not deadlock free
ERROR:root:Stopping analysis because of problems with the
```

Table 6.2: Repetition vector of the model from Figure 6.5 and Figure 6.6.

actor	repetition
t1	1
t2	4
t3	1
t4	2
t5	2

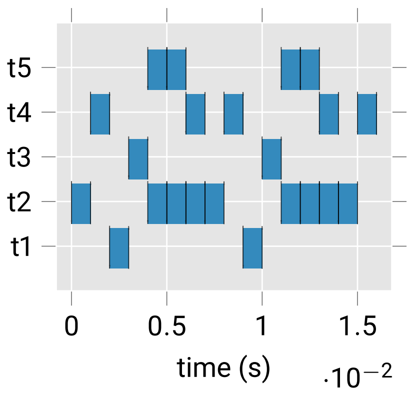

Figure 6.7: Timing diagram of the model from Figure 6.6

```
    model
```

Listing 6.4: Deimos output from the large consistent model of Figure 6.5. Shown is the analysis stage being aborted due to possible deadlock in the model

```
DEBUG:root:Model has repetition vector {'t2': 4, 't1': 1, '
    t3': 1, 't4': 2, 't5': 2}
INFO:root:Model is consistent
DEBUG:root:Cycle: ('t3',)
DEBUG:root:Building t3 to t3 with cons 1 and prod 1
DEBUG:root:Loop has tokens
DEBUG:root:No deadlock detected at cycle ['t3']
DEBUG:root:Cycle: ('t1',)
DEBUG:root:Building t1 to t1 with cons 1 and prod 1
DEBUG:root:Loop has tokens
DEBUG:root:No deadlock detected at cycle ['t1']
DEBUG:root:Cycle: ('t4',)
DEBUG:root:Building t4 to t4 with cons 1 and prod 1
DEBUG:root:Loop has tokens
DEBUG:root:No deadlock detected at cycle ['t4']
DEBUG:root:Cycle: ('t1', 't3', 't2', 't4')
DEBUG:root:t1: Production is 3
DEBUG:root:Building t1 to t3 with cons 2 and prod 3
DEBUG:root:t3: Production is 8
DEBUG:root:Building t3 to t2 with cons 3 and prod 8
DEBUG:root:t2: Production is 1
DEBUG:root:Building t2 to t4 with cons 2 and prod 1
DEBUG:root:Building t4 to t1 with cons 2 and prod 1
DEBUG:root:Loop has tokens
DEBUG:root:No deadlock detected at cycle ['t1', 't3', 't2',
     't4']
DEBUG:root:Cycle: ('t2',)
DEBUG:root:Building t2 to t2 with cons 1 and prod 1
DEBUG:root:Loop has tokens
DEBUG:root:No deadlock detected at cycle ['t2']
DEBUG:root:Cycle: ('t5',)
DEBUG:root:Building t5 to t5 with cons 1 and prod 1
DEBUG:root:Loop has tokens
DEBUG:root:No deadlock detected at cycle ['t5']
DEBUG:root:Cycle: ('t1', 't3', 't5', 't4')
DEBUG:root:t1: Production is 3
DEBUG:root:Building t1 to t3 with cons 2 and prod 3
DEBUG:root:t3: Production is 8
DEBUG:root:Building t3 to t5 with cons 3 and prod 8
DEBUG:root:t5: Production is 1
DEBUG:root:Building t5 to t4 with cons 4 and prod 1
DEBUG:root:Building t4 to t1 with cons 1 and prod 1
DEBUG:root:Loop has tokens
DEBUG:root:No deadlock detected at cycle ['t1', 't3', 't5',
     't4']
INFO:root:Model is consistent and deadlock free
Result: model is valid
```

Listing 6.5: Deimos output from the large consistent and deadlock free model from Figure 6.6. Shown is a positive analysis result as the model is proven consistent and deadlock free

6.2.1 SADF extensions for models

SADF models extend the SDF based functionality with run-time configuration through state-based actors called detectors. A detector is able to influence consumption and production rates of a system through state tokens. An example model is shown in Figure 6.8 with the Deimos model source shown in Listing A.3. The detector `d1` modifies the execution time, production and consumption rates of the other two actors.

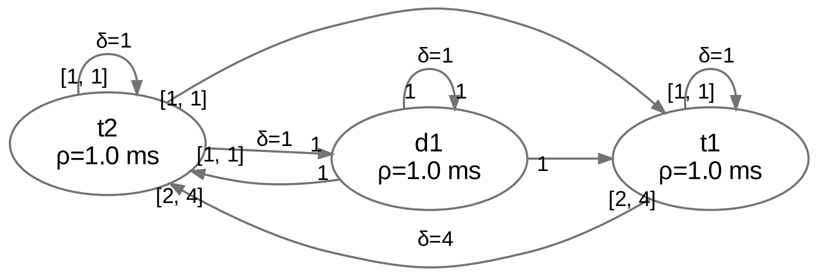

Figure 6.8: SADF-based model, Where shown, the production and consumption states are per state

The execution timing diagram is shown in Figure 6.9. The detector switches between states depending on whether the token consumed by `d1` has a positive or negative value. Visible is a variable execution time of `t2`, which is influenced by the detector state. The detector input from the output of `t2` is shown in Figure 6.10. The initial token supplied to `d1` is negative, supplying the `n` state to `t2`. The first execution thus is a short one. The output of `t2` after the first execution is positive as shown in the output plot. This changes the outputted state by `d1` to `p`, thus causing a long duration execution of `t2`. The new output of `t2` is negative, switching the state outputtted by `d1` again. As visible in Figure 6.10, the duration between a positive and negative value is longer than the duration between a negative and then a positive value, which is caused by the longer execution time of `t2` while it is the `p` state.

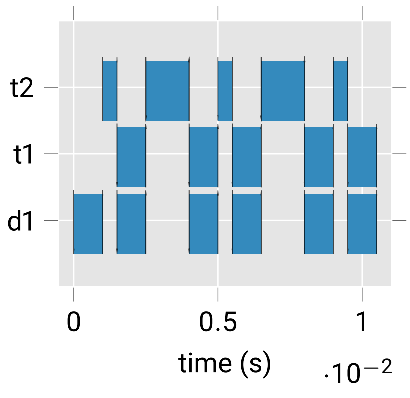

Figure 6.9: Timing diagram of the model shown in Figure 6.8

This sufficiently verifies the dataflow aspect of Deimos, showing how the models are verified and processed during simulation.

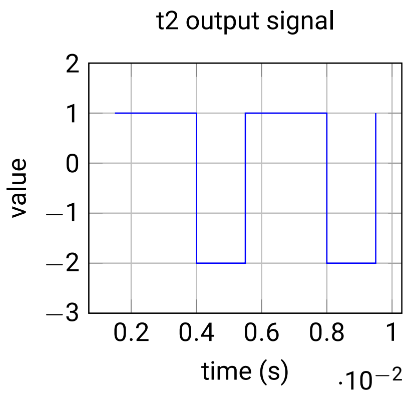

Figure 6.10: Simulated output value of `t2`.

With the verification output shown here on the available model verification and the interaction between the actors based on detector states emitted, the design is shown to implemententent an SADF based modelling sufficiently to satisfy requirement 1.

6.3 Model functionality verification

6.4 Simulation

To verify the functional aspects of the simulator itself, a comparison between Deimos and an existing and verified application, 20-sim, is done. This comparison is used as a verification of the basic simulation capabilities of the application.

Two models which should have identical behaviour in the two applications are simulated and compared for this verification. The two models each implement a simple PID control loop with a connected plant. A matching response, with small differences due to floating point rounding, is expected while comparing the results from these two simulators.

The model file used with Deimos is shown in Appendix A.2 and is shown in Figure 6.11.

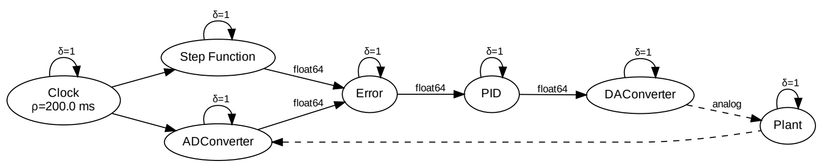

Figure 6.11: Representation of the Deimos model used for the 20-sim comparison

As shown in Figure 6.12, both simulators have a comparable signal trace. The maximum difference in time between 20-sim and Deimos in a matching data point is 1.07×10^{-14} s, the maximum difference in value between 20-sim and Deimos in matching data points is 1.22×10^{-15}. The value error here is well within the quantization error of the simulated 12-bit A/D-converter component.

The trace signal shown in Figure 6.12 from Deimos can be used to verify the functional aspects of the model. The full simulation and the output here shows support for design requirment 6, including all subrequirements except for requirement 6.2 for which support is shown. Furthermore, support for requirement 12 is shown as the model is configured to match the model from 20-sim.

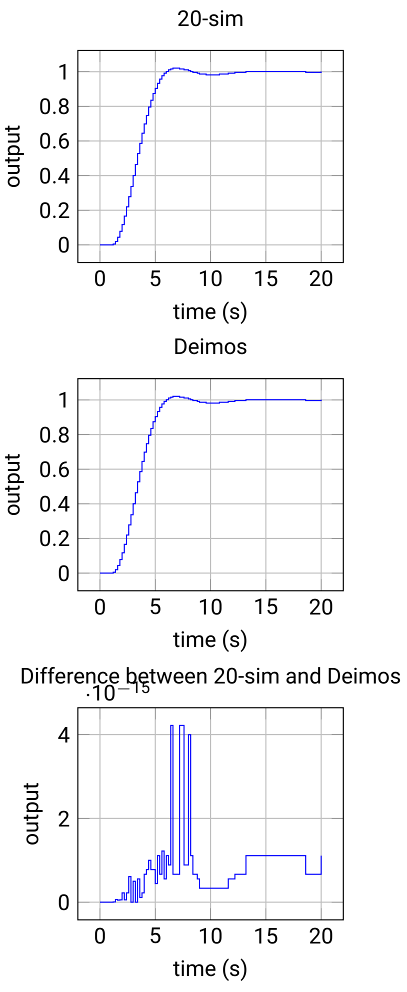

Figure 6.12: Simulation comparison between 20-sim and Deimos. Shown is the sampled A/D converter signal of both models

6.5 Physical Quantity support

The model from Appendix A.3 is used, this model has an integrator connected to a motion profile. The motion profile generates a profile between 0 m/s and 2 m/s with cosine-shaped ramps. It can be reasoned that with the motion profile ramping up and down over 1 s duration each and holding 2 m/s for 3 s, it must integrate to $2\,\mathrm{m\,s^{-1}} \times 3\,\mathrm{s} + 2 \times \frac{1\,\mathrm{s} \times 2\,\mathrm{m\,s^{-1}}}{2} = 8\,\mathrm{m}$ The first graph of Figure 6.13 shows the motion profile applied to the integrator. The second graph shows the output of the integrator, resulting in 8 m after the motion profile returned to $0\,\mathrm{m\,s^{-1}}$ output.

This example shows a multiplication between $\mathrm{m\,s^{-1}}$ and s. Attempting to add or subtract units with different dimensionality, an exception is produced and handled in the application aborting simulation as is shown in Listing 6.6.

```
INFO:root:Simulating for 8.0 second
INFO:root:Starting simulation
CRITICAL:root:Error simulating step for Integration at 0
    second: "Cannot convert from 'kilogram' ([mass]) to '
    meter * millisecond / second' ([length])"
CRITICAL:root:Task Integration received token with invalid
    unit [mass] received, expected [length]
```

Listing 6.6: Deimos output when simulating a model that attempts to add tokens with different dimensionality.

The support within Deimos for physical quantities can be shown and verified with a simple integrator design model. The verification done here is shown the support for design requirement 2.

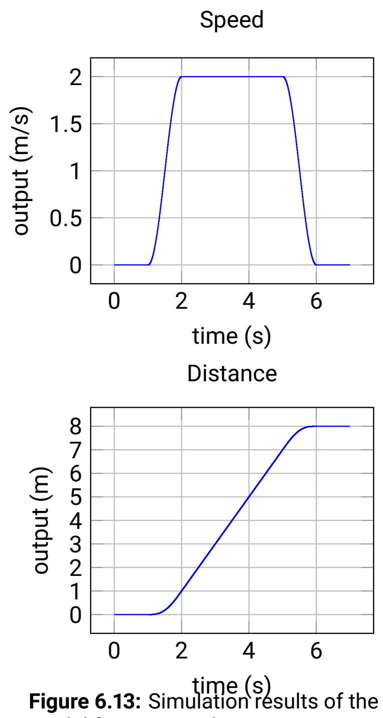

Figure 6.13: Simulation results of the model from Appendix A.3

6.6 Platform Metamodel

The platform metamodel offers a hierarchical way of describing component interconnections. It allows for descriptions of physical components with sufficient detail to allow for the model-to-model conversions.

An example board, a hypothetical board containing a microcontroller and the same A/D and D/A-converter as the RaMstix board, is shown in Figure 6.14. It contains an Atmel Atmega328 microcontroller as compute component. The sensor, an ADS8321, and the actuator, an AD5541, are connected to the microcontroller via two different SPI interfaces.

This model reuses the definition files for the A/D and D/A-converter from the RaMstix design and thus only contains a new board combin-

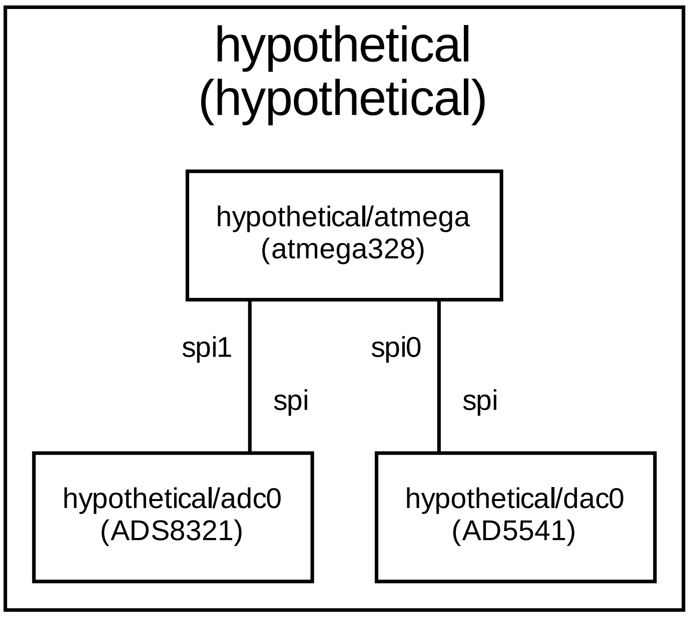

Figure 6.14: Graphical representation of the hypothetical platform model as described in Listing A.19. Each component has a name enclosed in parentheses. The model describes an Atmega328 compute component and two peripherals.

ing the atmega based microcontroller and the two other components with the necessary definitions for the interconnects. In both models it is visible how each part is connected to the other parts and which interfaces are used for the interconnects. This allows the model-to-model conversion to select the appropriate model representations of these interconnects. This fullfills requirement 13 including the sub-requirements.

The hierarchical nature of the platform metamodel allows for reusing the components by including them in different designs.

The plant is intentionally not included in the platform descriptions shown above, to allow reuse of the different boards with different plants. Adding plants in the platform models is possible by including them in a platform descriptor together with the board and the interconnections between the components.

6.7 Model-to-model Conversions

The platform models as shown earlier are used to convert the PIM to PSM. Three aspects of the model-to-model conversions are verified here. Whether the conversion results in a model accurately reprenting the target hardware requires additional measurements comparing the models with an implementation on the platform and is considered out of scope here.

First the expansion of function blocks to multiple actors is verified. This modification is used when a single actor does not offer the required level of detail for representing the platform components.

The second part is consists of modifications to parameters of the function block based on the specification of the platform models. This includes the calculation of approximate execution times based on the supplied platform model specification.

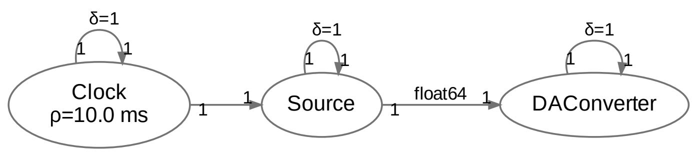

Figure 6.15: Simple model containing a source and a D/A-converter.

The main verification step required is to confirm that a model-to-model conversion selects the variant for the models and adapts the parameters. Whether the conversion results in a correct representation of the hardware platform then only depends on the model itself and the implementation of the function block variant which is out of scope here.

The system model used as example conversion is shown in Figure 6.15. The platform model used as target for the conversion is shown in Figure 6.16. It contains a clock, a source and a D/A-converter. The source acts as a sawtooth-wave generator.

The platform model used for this test describes a generic FPGA connected to the D/A-converter via a serial bus as shown in Listing A.14.

The platform is used for the verification tests and models a set of generic components without physical counterpart. As specified by the platform model, the D/A converter has the `SPIDAC` variant.

The description of the platform model indicates an SPI-based connection between the D/A-converter and the FPGA. The SPI bus, as a synchronous serial bus, needs to be modelled by two actors for sufficient detail. The first actor is representing the peripheral interface between the FPGA and the SPI bus modelling the latency in the interface. The second actor models the delay caused by the serial bus transfer.

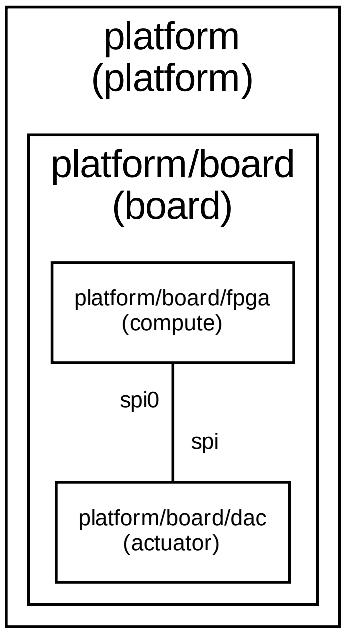

Figure 6.16: Board model containing a generic FPGA and a generic D/A-converter connected via an SPI-based interface. The platform model description is shown in Listing A.15

6.7.1 Actor Expansion

Conversion of the PIM to the PSM results in the model as shown in Figure 6.17. The actor representing the D/A-converter in the PIM is expanded into three actors to represent the whole SPI connected D/A-converter. The `DAConverter/periph/tx` actor represents the delay caused in the SPI peripheral of the FPGA. The `DAConverter/channel/tx` actor models the delay caused by the communication channel. This delay, and with that the execution time of the actor, depends on the throughput of the channel and the bits transfered per sample. The final component in the graph is the `DAConverter/dac/DAC`, representing the delay in the D/A-converter peripheral itself and adding the computational model for the D/A-converter.

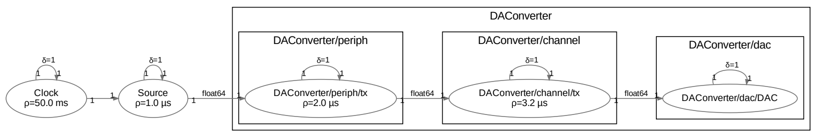

Figure 6.17: PSM of the model shown in Figure 6.15 using the platform model shown in Figure 6.16.

6.7.2 Timing Adaptation

The PSM as shown in Figure 6.17 is influenced by the specification of the platform model. The original PIM does not contain timing information besides the sampling rate dictated by the `Clock` function block. The PSM is adapted with the extra information and the function blocks now contain execution time configured by the function block variants based on the specification from the platform model.

The `Source` block is specified by the variant specific model to have an execution time equal to a single clock tick period of the specified FPGA clock rate. The default clock rate of the FPGA used here is configured at 1 MHz, resulting in an execution time of $\frac{1}{1\,\text{MHz}} = 1\,\mu\text{s}$. When this clock rate is modified to 100 kHz, the new execution time generated is equal to 10 μs.

The execution times of the function blocks in the D/A-converter block are derived from both the D/A-converter model and the FPGA model. The `DAConverter/periph/tx` actor timing is purely dependent on the FPGA specification and the execution time is lengthened by a factor 10 with the clock rate modification to 100 kHz. The `DAConverter/channel/tx` actor however depends on the SPI interface specification from both the FPGA and the D/A-converter platform models and is configured as an interface-specific property. The delay of the serial channel results from the maximum clock rate of the channel and the number of bits contained in a single transfer. The maximum clock rate of the channel is the lowest clock rate between the two components using the bus. The execution time of the channel is set to $wcet = \frac{bits}{clockrate}$. Between the D/A-converter and the FPGA, having a clock rate of 50 MHz and 5 MHz respectively, the 5 MHz from the FPGA SPI interface specification is used as a clock rate for the bus. The D/A-converter specifies 16 bits for a single transfer. This results in an execution time of $\frac{16}{5\,\text{MHz}} = 3.2\,\mu\text{s}$ as shown in Figure 6.19.

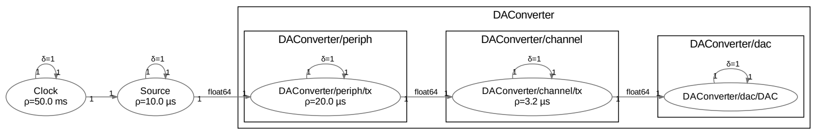

Figure 6.19: PSM of the model shown in Figure 6.15 using the platform model shown in Figure 6.16 with an FPGA clock rate of 100 kHz.

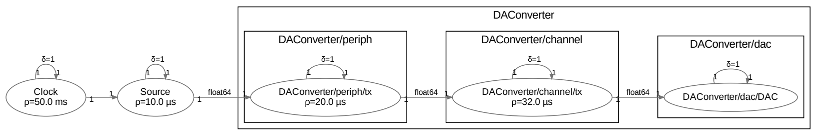

Figure 6.20: PSM of the model shown in Figure 6.15 using the platform model shown in Figure 6.16 with an FPGA clock rate of 100 kHz and an SPI interface clock rate of 500 kHz.

Decreasing the maximum clock rate of the SPI interface on the FPGA to 500 kHz results in an PSM as shown in Figure 6.20. The maximum clock rate of the D/A-converter is still specified as 50 MHz, causing the bus to be limited by the clock rate of the FPGA. The resulting execution time of actor representing the bus delay is $\frac{16}{500\,\text{kHz}} = 32\,\mu\text{s}$

6.7.3 PSM Parameter override

The specifications contained in the platform model override the parameters in the PIM when generating the PSM. In the example used here as shown in Figure 6.17, one of the parameters modified is the resolution of the D/A-converter is adapted to the resolution specified in the platform model.

Modifications to the specification of the D/A-converter component are reflected in the PSM. Reducing the resolution specified in the platform model from 16 bit to 8, 6 and 4 bit causes a reduction in the number of quantization levels in the D/A-converter output. The results of this parameter modification are shown in Figure 6.18. The input of the

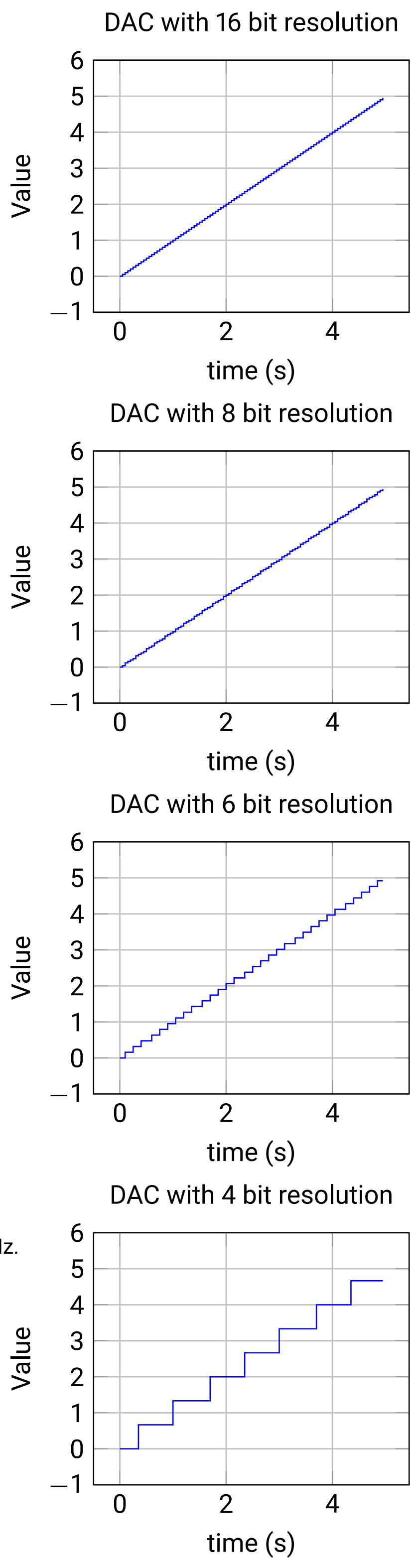

Figure 6.18: Simulation results of the DAC PSM with different resolution values specified in the platform model. Shown is the output of the DAC.

D/A-converter is a ramp function from 0 to 5 over 5 s, and the D/A-converter is used to quantize the values and convert them to an analog signal. Clearly visible in the series of graphs is the reduction in resolution and the reduced number of quantization levels produced by the D/A-converter. Where with the 16 bit resolution, a smooth ramp is produced, the 6 and 4 bit D/A-converter output the same values for multiple different closely related input values.

7 Case Study

A full workflow example using Deimos is used to validate the design flow of the application. The end goal is a functional PSM of the model to simulate the model as if it is running on the platform.

The in-depth verification options of Deimos are used to verify the design and the functionality of the different function blocks used.

The incremental refinement workflow as shown in Figure 1.1 is used here. The control laws in a format suitable for Deimos are provided as a start. Furthermore, an FMU for the plant model is available for cosimulation with the plant. The PIM is constructed from the control loop configuration and is evaluated using Deimos. Two platform model are available for the model-to-model conversion. Based on these platform models, two PSMs are generated and simulated. The results are compared with simulation results from 20-sim where applicable.

7.1 Model

The example used for the case study is the Linix system (Vries et al., 1997) including a controller. The Linix plant model consists of two axes, each with an inertia attached and connected with eachother via a belt running along pulleys on each axis. A sensor and motor are connected on the same axis with the secondary axis acting as load. A graphical representation of the Linix setup is shown in Figure 7.1.

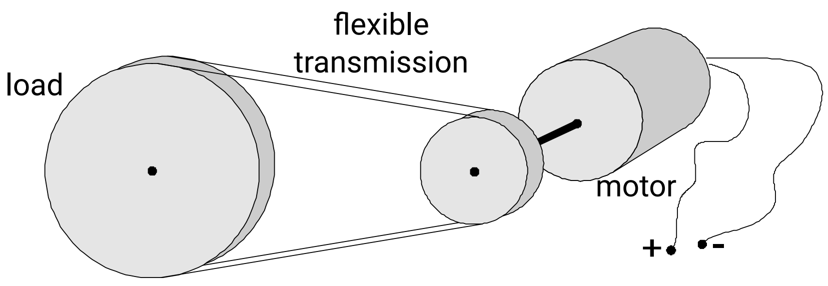

Figure 7.1: Graphical representation of the Linix setup.

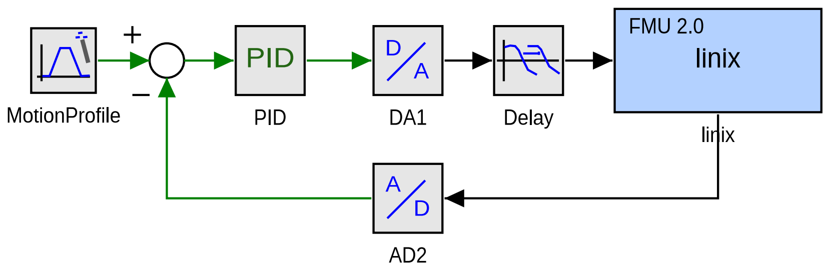

Figure 7.2: 20-sim model of the Linix system with additional configurable delay block.

Design and tuning of the controller is out of scope, the provided values are used. The controller is based on an PID controller, and AD/DA-converters for interfacing with the plant. The controller is configured with a loop frequency of 50 ms. The AD and the DA converters are configured with a resolution of 12 bits and with a range between `+10` and `-10`. The configuration parameters used for the PID controller are shown in Table 7.1. As input signal, a motion profile is used with with a stroke of 8 rad and using a sine wave shaped transition.

A model of the system with both the controller and the plant is available in 20-sim for a comparison of the simulation results. The model used in 20-sim is shown in Figure 7.2 The Deimos model is expected to have a similar block diagram topology as the 20-sim diagram. Deimos and 20-sim use the same FMU for simulation of the plant behaviour.

Table 7.1: Configuration of the PID controller used in the Linix setup.

Parameter	Value
β	0.001
k_p	0.29
τ_D	130 ms
τ_I	11.6 s

7.2 Platform Independent Model

A description of the PIM suitable for Deimos is available in Listing A.7. The resulting block diagram of the model is shown in Figure 7.3 and the SDF graph of the PIM is shown in Figure 7.4. To exclude effects from the loop delay, the PIM is intentionally designed without delays configured in the function blocks.

The block diagram shows the expected structure of the PID controller with motion profile function and A/D-converter as inputs to the error block. This setup is also reflected in the SDF graph.

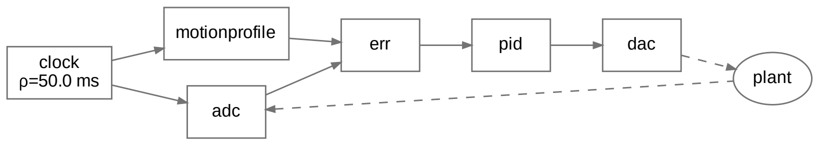

Figure 7.3: Block diagram of the Linix system model.

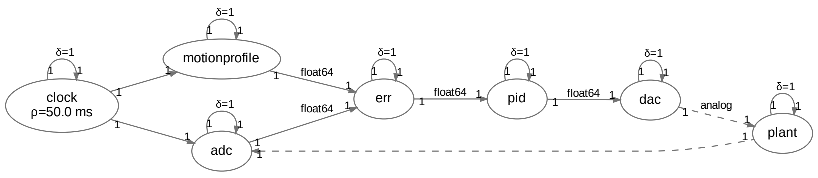

Figure 7.4: SDF graph of the Linix system model.

7.3 PIM analysis

The PIM as shown in Listing A.7 is verified in both non-functional and functional aspects. The non-functional verification consists of consistency and deadlock analysis performed by Deimos. After this verification, the model is simulated as to show the dynamic behaviour of the plant and the controller. This behaviour is compared to the 20-sim simulation results.

Analysis of the PIM by deimos shows a consistent and deadlock free model (Listing 7.1).

```
INFO:root:Model is consistent
INFO:root:Model is consistent and deadlock free
Result: model is valid
```

Listing 7.1: Deimos output when checks on the model as visible in Figure 7.4 is executed. Shown is a positive analysis result as the model is proven consistent and deadlock free

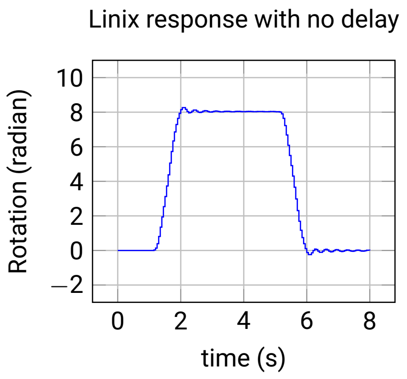

Figure 7.5: Simulation results of the Linix model. Shown is the response of the plant as sampled by the A/D-converter of the model.

Simulation results of the model are shown in Figure 7.5. A comparable simulation is performed using 20-sim. The difference in simulation results is visible in Figure 7.6. While both simulators have a comparable response, a difference of at most 6 least significant bit values is measured over the simulation time. This is a quantization difference of $\frac{6}{4096} \times 100\,\% = 0.15\,\%$ over the full range of the A/D-converter.

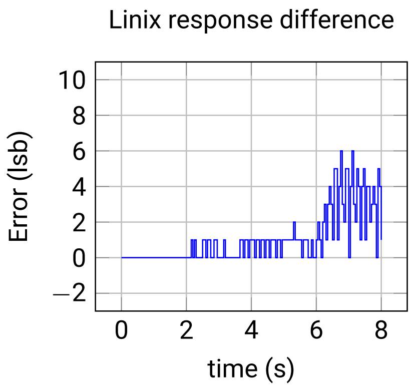

Figure 7.6: Simulation result difference between 20-sim and Deimos.

7.3.1 Delay effects

To show the effects of a delay within the loop, the Linix system model with plant is reconfigured with an additional delay. A delay is added to the Deimos system model using a FIFO function with a modified `wcet` to influence the delay. The D/A-converter will produce an analog value as soon as it received a new token, the delay causing a delay between the PID function block and the new analog value produced by the D/A-converter function block. The 20-sim model is also modified with the same delay for comparison between the two simulators.

The models differ in that Deimos has the delay before the D/A-converter function block and the 20-sim model has the delay after the D/A-converter block. This difference is caused by the limitations of both simulator implementations. For 20-sim, it is not possible to add a continuous-time delay in the discrete-time parts of the model, within a discrete section, it is limited to delays of a multiple of the sample period. By adding it in the continuous-time section of the system, it can be configured to match the delay configured in Deimos. Within Deimos it is not possible to add an event-based FIFO after the D/A-converter, this is a limitation of the simulated connection between the D/A-converter and the plant, it does not adhere to the dataflow semantics expected by the other function blocks.

The simulation results for the Linix model with 20 ms, 30 ms, and 40 ms delays in the loop using Deimos as simulator are shown in Figure 7.7. These results are compared to an identical simulation run using 20-sim for verification of the results. The difference between the two simulations is shown in Figure 7.8. The diffence is shown in as multiple of A/D-converter least significant bit values.

Between the two simulators, a difference in the resulting response increases up to 10 least significant bit values over the 8 s of simulation time. This is a quantization difference of $\frac{10}{4096} \times 100\,\% = 0.25\,\%$ over the full range of the A/D-converter.

The simulation performed here using Deimos will act as a baseline for the dynamic behaviour of the control system when the PSM behaviour is evaluated.

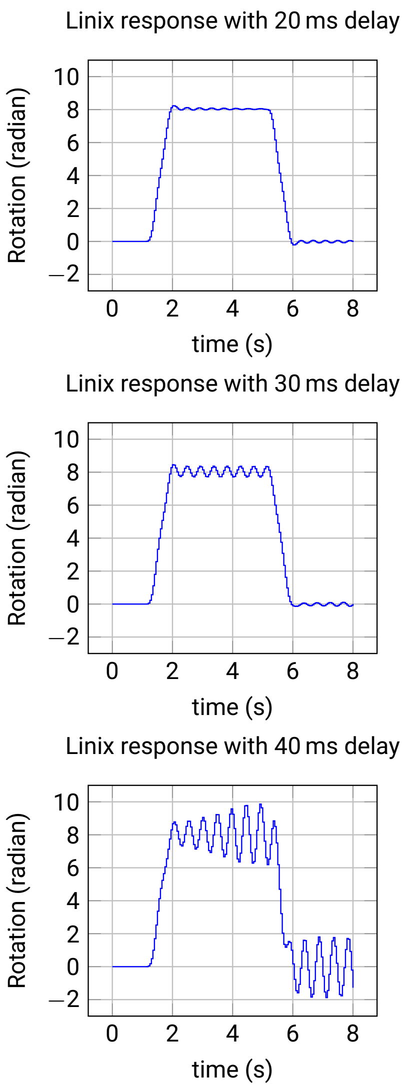

Figure 7.7: Simulation results of the Linix model with different delays. Shown is the response of the plant as sampled by the A/D-converter of the model.

7.4 Platform Design

Two platform models are available as target for the conversion to a PSM. First is the RaMstix board which includes an FPGA, D/A and A/D-converters to run the control system on. The second option is an Arduino based board with integrated D/A and A/D-converters.

7.4.1 RaMstix board

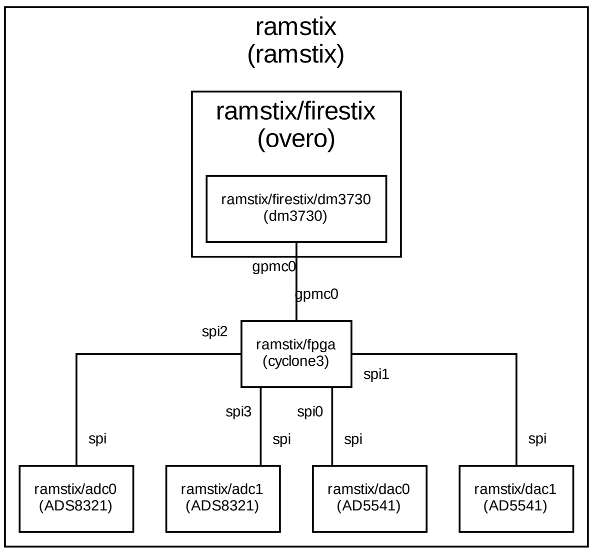

Figure 7.9: Representation of the platform model of the RaMstix board as described in (Listing A.17). Each component has an identifier and a name (enclosed in parentheses).

The RaMstix board as modelled with the relevant components is shown in Figure 7.9. Visible are the FPGA and DM3730 providing the compute components. Sensors and acutators are available via multiple ADS8421 and AD5541 instances on the RaMstix board. Specifications used with the components are retrieved from the datasheets of the respective components.

In Figure 7.10, a full platform including the RaMstix board and the plant is shown. The source of the platform model is available in Listing A.12. Visible is the RaMstix board and how the plant is connected to an A/D and D/A-converter of the RaMstix board.

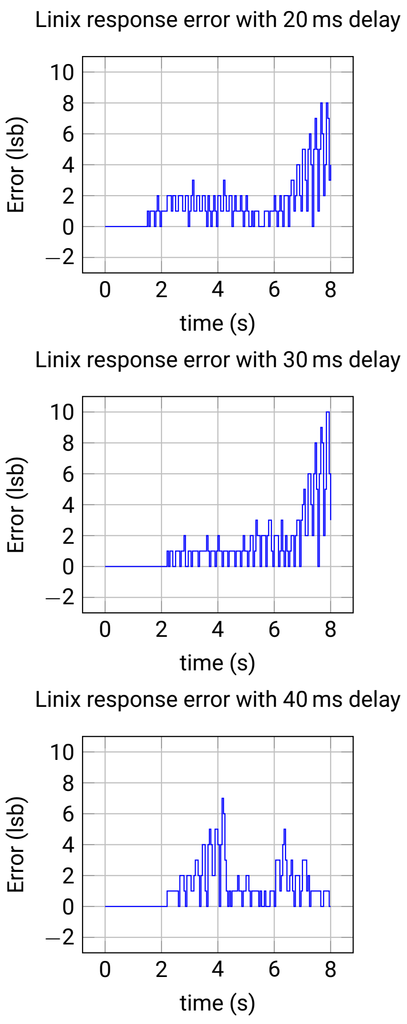

Figure 7.8: Simulation error results of the linix model with different delays. Shown is the error in the response compared to the 20-sim simulation in a/d-converter bits.

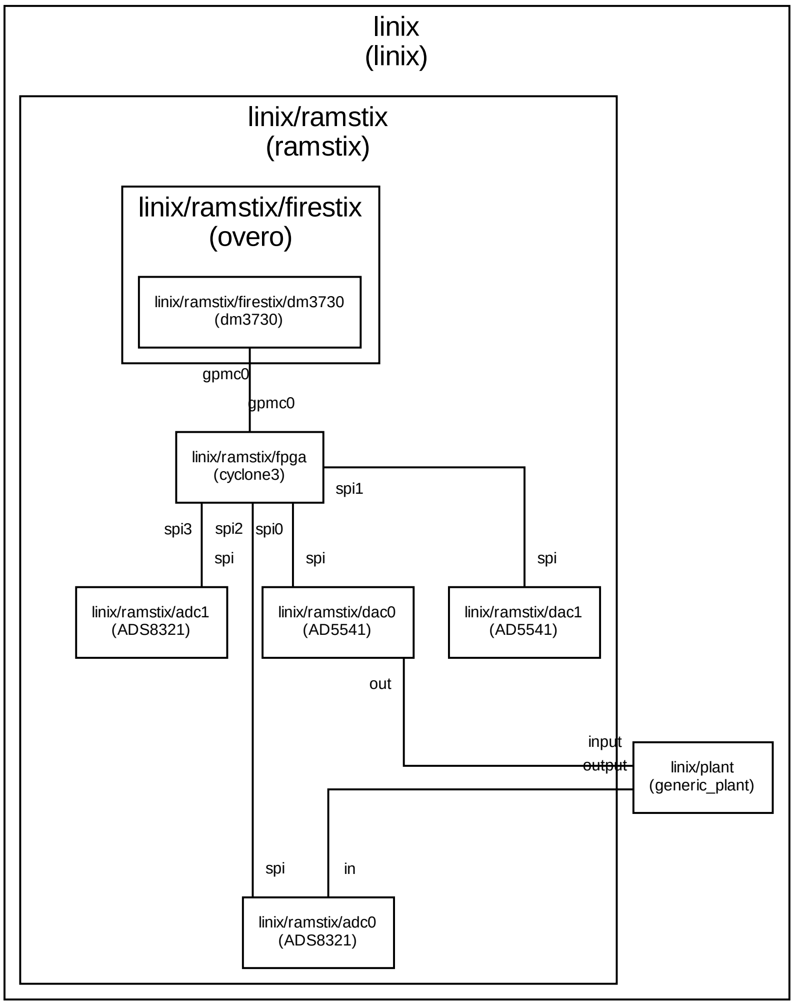

Figure 7.10: Graphical representation of the platform connecting a plant to the RaMstix board from Listing A.12.

The D/A and A/D-converters connected to the FPGA in the RaMstix model are connected via an SPI bus. This serial bus has a non-zero delay in the transfer of the values from and to the peripherals. These provide 16 bit D/A and A/D-converters for interfacing with the plant. When using the platform model as target for the PSM conversion, a communication delay is expected with data from the A/D-converter and data to the D/A-converter.

The resolution of both the A/D-converter and the D/A-converter is higher than used in the PIM, no additional sampling noise from the value quantization is expected.

7.4.2 Arduino Uno

The second board that is used as a target for the PSM conversion is a board based on the Arduino Uno[2]. The microcontroller, an Microchip Atmega328[3], provides the computational aspect of the platform, but also provides A/D and D/A peripherals.

[2] https://store.arduino.cc/arduino-uno-rev3-smd

[3] https://www.microchip.com/wwwproducts /en/ATmega328

The D/A-converter provided by the microcontroller is an 8 bit pulse width modulation (PWM) based conversion between 0 V and 5 V.

The A/D-converter offers 10 bit resolution between 0 V and 5 V. The A/D-converter available on the AVR microcontroller does have a non-zero delay, but in contrast to the RaMstix board, it is internal to the microcontroller and delays caused by an external bus are non-existent. The A/D-converter iself of the Atmega328 has a delay and is modelled to specification.

Sample delays of the A/D-converter are based on specifications of the Atmega328 component. For the MCU-based platforms, the A/D-converters are modelled as two connected actors where the first represents the setup time before a sample moment and the second represents the delay caused by the conversion process.

The platform model including the plant is shown in Figure 7.11.

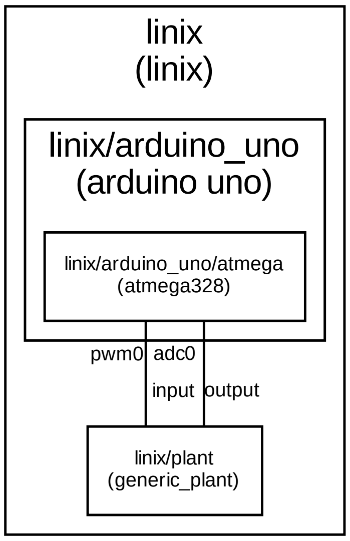

Figure 7.11: Graphical representation of the platform connecting a plant to the Arduino Uno board from Listing A.13.

The decrease in resolution with both the A/D and the D/A-converters is significant. With the conversion to the PSM, oscillations due to value quantization noise can occur in the PSM simulations.

7.5 PSM Conversion

Both platforms are used as conversion target for the PIM, to allow for a comparison of the dynamic response in both situations.

7.5.1 RaMstix Target

The PSM as generated based on the PIM and the RaMstix platform is shown in Figure 7.12. The mapping file used for the conversion is shown in Listing A.9. A number of actors have been added to increase the level of detail in modelling the communication bus between the different peripherals. The model actors are configured with an execution time. Execution times of the FPGA-allocated actors in the model are an estimate, accurate execution times for these function blocks are out of scope. Execution times of the A/D-converter and D/A-converter actors are based on the specifications from the datasheets of these components.

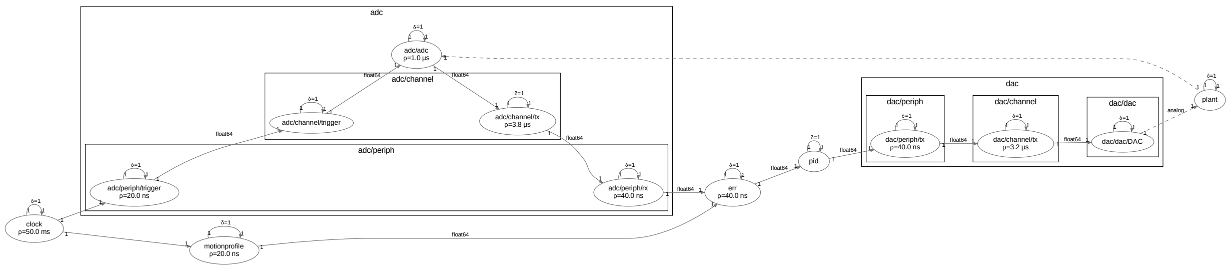

Figure 7.12: Graphical representation of the PSM of the Linix setup using the RaMstix as target platform.

The output values of the D/A-converter and the input values of the A/D-converter are scaled to correct for the power amplifier connected to the

Linix plant. This does not influence the resolution of the signal. Without this the performance evaluation would be influencend in a negative manner.

The PSM is verified and simulated using Deimos, the SDF model is confirmed to be consistent and deadlock-free. The response as visible in Figure 7.13 closely matches the initial response from the PIM, the settle time is slightly longer compared to the simulation results from the PIM. The timing diagram, as visible in Figure 7.14, shows that a significant portion of the processing time is occupied with the SPI channel transaction. The duration of the transfers and computations from A/D-converter to the D/A-converter is slightly longer than 80 µs.

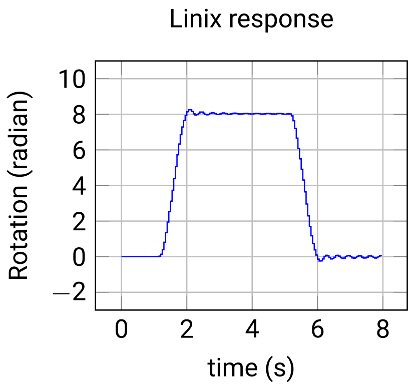

Figure 7.13: Simulation results of the Linix model when converted to a PSM based on the RaMstix platform. Shown is the response of the plant as sampled by the A/D-converter of the model.

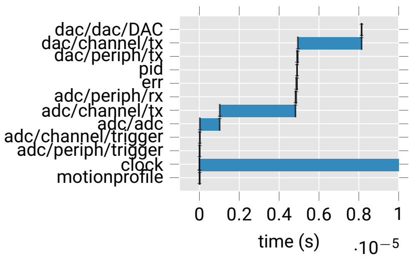

Figure 7.14: Timing diagram as result from the simulation of the Linix model with the RaMstix platform.

7.5.2 Arduino Uno Target

The PSM as generated based on the Arduino platform and the PIM is shown in Figure 7.15. The mapping file used for the conversion is shown in Listing A.10. All actors have an execution time. The A/D-converter is split in two actors where the first actor represents the setup time before sampling.

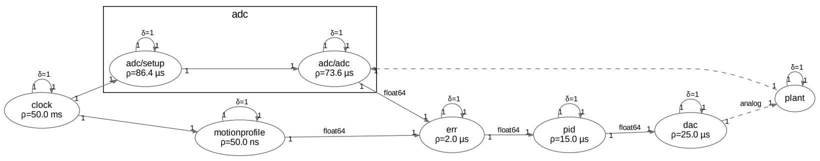

Figure 7.15: Graphical representation of the PSM of the Linix setup using the Arduino Uno as target platform.

The PSM is verified and simulated using Deimos, the SDF model is confirmed to be consistent and deadlock-free. The simulated response on this platform is shown in Figure 7.16. Where with the PIM model, the response would settle within the simulation time, the PSM based on the Arduino does not settle within the simulation time.

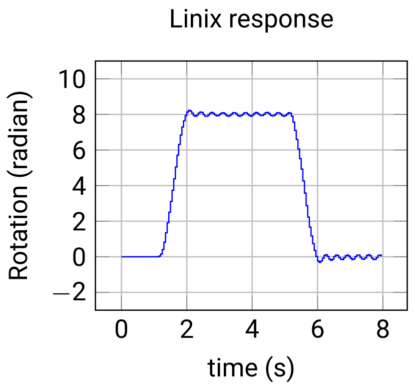

Figure 7.16: Simulation results of the Linix model when converted to a PSM based on the Arduino platform. Shown is the response of the plant as sampled by the A/D-converter of the model.

The timing diagram, as visible in Figure 7.17, shows that a significant portion of the processing time is occupied with the A/D conversion. The full transaction takes 202 µs, though the time from an A/D conversion sample to D/A-converter output is 116 µs. The A/D converter takes 86.4 µs to set up for a conversion which does not influence the latency between the feedback and the output of the system in this case.

7.6 Evaluation

Simulations of the dynamic behaviour of the Linix plant differs significantly between the two PSMs. Where the RaMstix-based model is able to sufficiently control the Linix system with a finite settle time, the Arduino-based PSM does not settle within the simulation time. Both PSMs have a latency well below the 10 % of the loop frequency and should be fast enough for stable loop control.

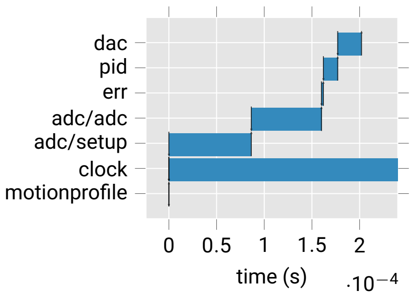

Figure 7.17: Timing diagram as result from the simulation of the Linix model with the Arduino platform.

The RaMstix platform closely matches the dynamic behaviour of the PIM simulation results. Assuming the platform model matches the real-world perfomance close enough, it would be a suitable platform as target for the Linix control system.

The Arduino-based model suffers from the limited A/D and D/A converter resolution. While it is able to control the system, oscillations within the signal remain and it would not be advisable to use the platform for a Linix control system in the current configuration. Thus this

controller should be redesigned to be robust enough to handle the 8-bit
converters though this might not be possible.

8 Discussion

The dataflow-based MDE solution as presented in this thesis generally offers sufficient capabilities for proper modelling a basis for a model driven design approach of control systems. However the design process allows for a number of observations based on the application requirements and test results.

SADF allows for complex systems with sufficient expressiveness for modelling control systems. As results from the tests and validation, using SADF, as required by requirement 1, as modelling backend fits the requirements for modelling control systems, the dataflow semantics map well to the control system requirements. When required, the run-time choice provided by SADF allows for modifying the controller in a number of different ways such as parameter modifications allowing flexibility and choice during the control system execution. It further allows for supervisory control based on state machines of the control systems.

One of the limitations of the dataflow semantics is that modelling asynchronous events is not compatible with the semantics and analysis options. Asynchronous signals such as interrupts from sensors can only be modelled by converting them to polling based system. This limits the systems that can be modelled to purely clocked and thus synchronous models. Other limitations in the modelling language is that mutual exclusive access to a resource can't be modelled. This can be resolved by modelling the resource access by a static schedule. Further validation of the expressiveness provided by Deimos and required by control systems is recommended to explore the limitations of the modelling semantics.

Model analysis allows for early detection of system issues. Analysis to prove a system is consistent and deadlock free are available with the current implementation, providing an implementation for both requirement 3 and 4. The model analysis integrated in the design allows for basic validation of the model providing proofs whether the model is both consistent and deadlock free. With additional extensions to the analysis, more elaborate timing analysis such as a slowest cycle analysis can be added such as proposed by requirement 5.

Given the latency-critical nature of control systems, an automated analysis of the latency between an input of one function block and the output of another is a useful addition to the analysis suite of the application. It would allow end users to both analyse the latency aspect of the loop, but also show if there is a component having a significant impact on the loop latency. This analysis is already somewhat available with the timing diagrams produced by Deimos, but a more automated structure would offer benefits.

Simulator performance: The simulator fully satisfies requirement 6 and the simulator performance is sufficient to simulate larger models on the desired time scale relevant for control systems. The discrete event-based nature of the simulator allows for dynamic time steps without a fixed discrete time step during simulation. It allows for functional verification of the full SMM as shown during the case study. The implementation allows for a flexible computational models and deterministic simulation. Different components including externally designed plants can be combined in complex SMMs. The included support for physical quantities in the simulator (requirement 2) adds valuable refinement to the models due to the added inherent verification of relations between the different dimension-related operations handled by the components. The FMI-based plant models integration provides the essential co-simulation required to verify the dynamic behaviour of the plant based on the designed control system.

There however seem to be small differences in the simulation results when comparing complex models with 20-sim. While not significant in the tested cases, care has to be taken that the simulator operates within reliable bounds.

The hierarchical nature of the platform meta-model allows for creation of a reusable platform model library. The PMM offers a structure to describe components and their specifications, as required by requirement 13. By explicitly distinguishing the different components into types the framework enforces a structured approach, where reuse of components is The meta-model provides a structure to describe interconnections between the components where it allows freedom within the conversions to adapt the PIM to the specified interfaces during the conversion.

Model-to-model transformations allow for freely replacing function blocks and adapting configuration parameters. The implementation behind the model-to-model conversions allow for a lot of freedom in replacing function blocks for their platform specific counterparts, satisfying requirement 7 through 10. It allows for both refinement of the actors already present in the PIM and for addition of new actors, for example to represent communication bus based delays in the model. The downside of this is that it leaves a lot of responsibility with the component design for the design of valid models. Verification of these platform specific function blocks requires careful testing and measuring. One of the limitations is the absence of dealing with the limited parallelization of the target platform, while a processor-based compute module has no or only limited options for parallelization, the model does not take this into account and will execute multiple function blocks concurrently.

9 Conclusion and Recommendations

The goal of this thesis is to improve the way control systems are designed and built, improving both the workflow of the control engineer and the software engineer. The proof of principle application presented in the previous chapters, together with the test results is used to answer the research questions from Chapter 1.

Which dataflow models of computation are suitable as basis for model-driven development?
The approach based on SADF with a function block oriented approach, allows for capturing both the required dynamic behaviour of the control system while allowing for detailed analysis. The model-driven engineering approach allow the end user to focus on the algorithmic aspects of the system, without having to take structural details into account in the early design step. The consistency and deadlock analysis offers feedback to the end-user in every step of the design process. When the dynamic behaviour is not required for modelling the control system, the more restricted SDF MoC can be used to simplify both the analysis and the code synthesis. However, the model design does not allow for modelling asynchronous events sources, limiting the ability to respond to external events from the plant.

Which simulation features are required to verify the supplied models?
The dataflow-based models are extended with computational function implementation for the application to allow functional verification. The physical quantity extension in the model allows for additional model refinement, and provides verification of the function block interaction based on dimensional analysis. Furthermore, the co-simulation module allows for simulating the control systems against plant models using the FMI specification. These features combined allow the simulator to give the end user extensive information about both the signal values and the timing of the system models, providing continuous verification of the functionality of the system model during each design steps.

Which platform specific information is required to transform the dataflow based model into a PSM?
The information provided by the platform models is twofold, it describes the timing and resolution specification of the physical components, but also provides information on the interconnections between the physical components on the hardware platforms. The components describe the timing and resolution specifications of the physical counterparts and their interfaces. The board and platform component meta models provide semantics reuse of component models in different configurations. The model-to-model conversions based on these platform models to generate the PSM only modify existing actors or add additional actors to the PIM, this way retaining the existing analysis and simulation options available.

When comparing the results of this thesis to previous work, a number of remarks can be made. The dataflow-based approach presented provides a workflow for modelling control systems with a strong focus on analysis and functional simulation. The dataflow MoC used, while originally designed for DSP systems, is successfully used to model control systems. It offers an alternative to the more often used UML-based or CSP-based approaches by using an algorithmic approach to the design workflow. As shown in the case study, the stepwise approach combined with dataflow-based models allow for verification of the modelled system during each of the design steps to detect problems in the control system design as soon as possible.

9.1 Recommendations

The recommendations for next steps on this topic are split into research-related recommendations, and more practical application-related recommendations. Two paths are recommended for extending the research into dataflow-based control system design:

- *Integration in other design applications*: Deimos is heavily geared towards latency analysis of control loops. Integration with other MDD applications can provide valuable interaction where the external application provides high-level supervisory control for which the analysis currently available in Deimos is less suitable.
- *Code synthesis*: Based on the stepwise model refinement workflow, the next step is platform code synthesis. Related work on the topic of code synthesis based on dataflow models is available, and should be applicable to the platform code generation required for control systems.

To further improve the application itself and allow for full validation of the application workflow, it is recommended to add the following functionality:

- *Extensive latency analysis*: Extending the already existing runtime schedule display with latency analysis should allow for identification of critical delays in the full control path.
- *Clock timing analysis*: In a clocked system, all clocked processes must have a worst-case execution time lower than the clock period. A maximum cycle mean analysis of the different cycles and clocked processes can offer proof that for the models the clock is the slowest process. This helps in validating that the used function blocks are fast enough for the required clock rate.
- *Function block execution scheduling on different processing types*: While an FPGA is able to execute any number of processes in parallel, a microcontroller can only execute a limited number of processes concurrently. The PSM models currently do not take this limitation into account when converting from the PIM. The model-to-model conversion can be extended with facilities to limit the parallel execution of a set of actors to take this limitation into account.

The recommendation here result into an improved workflow for the end user and add valuable extensions. The proposed extensions for the application further improves the use of dataflow models when applied to control system design.

Appendix A Model examples

A.1 SDF and SADF examples

```
name: deadlock_free

model:
  t2:
    type: resampler
    params:
      ratio: 0.25
      wcet: 1ms
    inputs:
      input:
        source: t1.out

  t1:
    type: resampler
    params:
      ratio: 4
      wcet: 1ms
    inputs:
      input:
        source: t2.out
        initial: [1]
```

Listing A.1: Deadlock free SDF model

```
name: consistent

model:
  t2:
    type: resampler
    params:
      ratio: 0.5
      wcet: 1ms
    inputs:
      input:
        source: t3.out
        initial: [1, 1]

  t1:
    type: resampler
    params:
      ratio: 3/2
      wcet: 1ms
    inputs:
      input:
        source: t4.out
        initial: [1]

  t3:
    type: resampler
    params:
      ratio: 8/3
      wcet: 1ms
    inputs:
      input:
        source: t1.out

  t4:
    type: mac
    params:
      wcet: 1ms
      inputs: 2
      factors: [1, 1]
      rates: [2, 1]
    inputs:
      0:
        source: t2.out
        initial: [1]
      1:
        source: t5.out
        initial: [1]

  t5:
    type: resampler
    params:
      wcet: 1ms
      ratio: 1/4
    inputs:
      input:
        source: t3.out
```

Listing A.2: Large deadlock free SDF model

```
name: deadlock_free

model:
  t2:
    type: mac
    params:
      inputs: 1
      factors: [-1/4]
      rates: [4]
      wcet: 1ms
    inputs:
      0:
        source: t1.out
        initial: [-1, -1, -1, -1]
      control:
        source: d1.state
    states:
      p:
        params:
          rates: [2]
          factors: [-1/2]
          wcet: 1.5ms
      n:
        params:
          rates: [4]
          factors: [-1/4]
          wcet: 0.5ms

  t1:
    type: resampler
    params:
      ratio: 4
      wcet: 1ms
    inputs:
      input:
        source: t2.out
      control:
        source: d1.state
    states:
      p:
        params:
          ratio: 2
          wcet: 1ms
      n:
        params:
          ratio: 4
          wcet: 1ms

  d1:
    type: controller
    params:
      wcet: 1ms
      states:
        - p
        - n
    inputs:
      detector:
        source: t2.out
        initial: [-1]
```

Listing A.3: Deadlock free SADF model

A.2 PID example

```
name: Plant FMU Model

globals:
  - &samplerate 200ms

model:
  Step Function:
    type: source
    params:
      samples: [0, 0, 0, 0, 0, 1]
      unit: meter
      repeat: false
    inputs:
      trigger:
        source: Clock.tick

  Clock:
    type: clock
    params:
      rate: *samplerate

  Error:
    type: mac
    params:
      inputs: 2
      factors: [1 per second, -1 per second]
      wcet: 0ms
    inputs:
      0:
        source: Step Function.samples
      1:
        source: ADConverter.digital

  PID:
    type: pid
    params:
      kp: 0.2
      ti: 0.5
      td: 0.0000001
      beta: 0.2
      wcet: 0ms
    inputs:
      in:
        source: Error.out

  DAConverter:
    type: dac
    params:
      bits: 12
      maximum: 10 m/s
      minimum: −10 m/s
      wcet: 0ms
    inputs:
      digital:
        source: PID.out

  Plant:
    type: fmu
    params:
      fmu: examples/fmu/transferfunction.fmu
      # Describes the units of the FMU ports
      ports:
        input:
          unit: meters/second
        output:
          unit: meters
    inputs:
      input:
        source: DAConverter.analog

  ADConverter:
    type: adc
    params:
      bits: 12
      maximum: 10
      minimum: −10
      wcet: 0ms
    inputs:
      analog:
        source: Plant.output
      trigger:
        source: Clock.tick
```

Listing A.4: FMU PID example

```
# filename of the target platform
platform: arduino_setup


elements:
  Step Function:
    target: arduino_uno/atmega

  Clock:
    target: arduino_uno/atmega

  Error:
    target: arduino_uno/atmega

  PID:
    target: arduino_uno/atmega

  DAConverter:
    target: arduino_uno/atmega
    specification:
      conversion: meter / (volt second)

  ADConverter:
    target: arduino_uno/atmega
    specification:
      conversion: meter / volt

  Plant:
    target: plant
```

Listing A.5: FMU PID example, arduino mapping file

A.3 Unit example

```
name: units example

globals:
  - &samplerate 10ms

model:
  Speed:
    type: motion_profile
    params:
      'off': 0 m/s
      'on': 2 m/s
      start_time: 1 second
      on_time: 2 second
      stop_time: 5 second
      off_time: 6 second
      wcet: 0ms
    inputs:
      trigger:
        source: Clock.tick

  Integration:
    type: mac
    params:
      inputs: 2
      factors: [1, *samplerate]
    inputs:
      0:
        source: Integration.out
        initial: [0 kilogram]
      1:
        source: Speed.samples

  Clock:
    type: clock
    params:
      rate: *samplerate
```

Listing A.6: Unit example

A.4 Linix demonstrator

```
name: Linix model

globals:
  - &samplerate 0.05s

model:
  motionprofile:
    type: motion_profile
    params:
      'off': 0 radians
      'on': 8 radians
      start_time: 1 second
      on_time: 2 second
      stop_time: 5 second
      off_time: 6 second
      wcet: 0ms
    inputs:
      trigger:
        source: clock.tick

  clock:
    type: clock
    params:
      rate: *samplerate

  err:
    type: mac
    params:
      inputs: 2
      factors: [1, -1]
      wcet: 0ms
    inputs:
      0:
        source: motionprofile.samples
      1:
        source: adc.digital

  pid:
    type: pid
    params:
      kp: 0.29 ampere / radians
      ti: 11.6
      td: 0.130
      beta: 0.001
      wcet: 0ms
    inputs:
      in:
        source: err.out

  dac:
    type: dac
    params:
      bits: 12
      maximum: 1 ampere
      minimum: −1 ampere
      wcet: 0ms
    inputs:
      digital:
        source: pid.out

  plant:
    type: fmu
    params:
      fmu: examples/linix/linix.fmu
      # Describes the units of the FMU ports
      ports:
        input:
          unit: ampere
        omega:
          unit: radians
    inputs:
      input:
        source: dac.analog

  adc:
    type: adc
    params:
      bits: 12
      maximum: 10
      minimum: −10
      wcet: 0ms
    inputs:
      analog:
        source: plant.omega
      trigger:
        source: clock.tick
```

Listing A.7: Linix demonstrator

```
name: Linix model

globals:
  - &samplerate 0.05s

model:
  step:
    type: motion_profile
    params:
      'off': 0 radians
      'on': 8 radians
      start_time: 1 second
      on_time: 2 second
      stop_time: 5 second
      off_time: 6 second
      wcet: 0ms
    inputs:
      trigger:
        source: clock.tick

  clock:
    type: clock
    params:
      rate: *samplerate

  err:
    type: mac
    params:
      inputs: 2
      factors: [1, -1]
      wcet: 0ms
    inputs:
      0:
        source: step.samples
      1:
        source: adc.digital

  pid:
    type: pid
    params:
      kp: 0.29 ampere / radians
      ti: 11.6
      td: 0.130
      beta: 0.001
      wcet: 0ms
    inputs:
      in:
        source: err.out

  delay:
    type: fifo
    params:
      wcet: 20ms
    inputs:
      in:
        source: pid.out

  dac:
    type: dac
    params:
      bits: 12
      maximum: 1 ampere
      minimum: -1 ampere
      wcet: 0ms
    inputs:
      digital:
        source: delay.out

  plant:
    type: fmu
    params:
      fmu: examples/linix/linix.fmu
      # Describes the units of the FMU ports
      ports:
        input:
          unit: ampere
        omega:
          unit: radians
    inputs:
      input:
        source: dac.analog

  adc:
    type: adc
    params:
       bits: 12
       maximum: 10
       minimum: −10
       wcet: 0ms
    inputs:
      analog:
        source: plant.omega
      trigger:
        source: clock.tick
```

Listing A.8: Linix demonstrator with delay function block

```
# filename of the target platform
platform: linix_ramstix

elements:
  motionprofile:
    target: ramstix/fpga

  clock:
    target: ramstix/fpga

  err:
    target: ramstix/fpga

  pid:
    target: ramstix/fpga

  dac:
    target: ramstix/dac0
    specification:
      conversion: 1 ampere / volt

  adc:
    target: ramstix/adc0
    specification:
      conversion: 4 radians / volt

  plant:
    target: plant
```

Listing A.9: Linix demonstrator, RaMstix mapping

```
# filename of the target platform
platform: linix_arduino

globals:
  datatype: float32

elements:
  motionprofile:
    target: arduino_uno/atmega

  clock:
    target: arduino_uno/atmega

  err:
    target: arduino_uno/atmega

  pid:
    target: arduino_uno/atmega

  dac:
    target: arduino_uno/atmega
    specification:
      conversion: 1 ampere / volt

  adc:
    target: arduino_uno/atmega
    specification:
      conversion: 8 radians / volt

  plant:
    target: plant
```

Listing A.10: Linix demonstrator, Arduino mapping

A.5 Platform model files

A.5.1 Platforms

```
type: platform
name: platform

components:
  board: !include ../boards/generic.yaml
```

Listing A.11: Test platform with FPGA and DAC file

```
type: platform
name: linix

components:
  ramstix: !include ../boards/ramstix.yaml
  plant: !include ../plants/generic.yaml

connections:
  - from: ramstix/dac0_analog
    to: plant/input
  - from: plant/output
    to: ramstix/adc0_analog
```

Listing A.12: RaMstix with plant platform file

```
type: platform
name: linix

components:
  arduino_uno: !include ../boards/arduino_uno.yaml
  plant: !include ../plants/generic.yaml

connections:
  - from: arduino_uno/pwm0
    to: plant/input
  - from: plant/output
```

```
    to: arduino_uno/adc0
```

Listing A.13: Arduino with plant platform file

```
type: platform
name: linix

components:
  arduino_uno: !include ../boards/arduino_uno.yaml
  plant: !include ../plants/generic.yaml

connections:
  - from: arduino_uno/pwm0
    to: plant/input
  - from: plant/output
    to: arduino_uno/adc0
```

Listing A.14: Arduino with plant platform file

A.5.2 Boards

```
type: board
name: board

components:
  fpga: !include ../compute/fpga.yaml
  dac: !include ../actuators/dac.yaml

interfaces:
  dac_analog:
    type: expose
    name: dac_analog
    specification:
      passthrough: dac/out

connections:
  - from: fpga/spi0
    to: dac/spi
```

Listing A.15: Generic test board with FPGA and DAC file

```
type: board
name: arduino uno

components:
  atmega: !include ../compute/atmega328.yaml

interfaces:
  spi0:
    type: expose
    name: spi0
    specification:
      passthrough: atmega/spi0
  spi1:
    type: expose
    name: spi1
    specification:
      passthrough: atmega/spi1
  adc0:
    type: expose
    name: adc0
    specification:
      passthrough: atmega/adc0
  pwm0:
    type: expose
    name: pwm0
    specification:
      passthrough: atmega/pwm0
```

Listing A.16: Arduino Uno board file

```
type: board
name: ramstix

components:
  firestix: !include overo.yaml
  fpga: !include ../compute/cyclone3.yaml
  adc0: !include ../sensors/ADS8321.yaml
  adc1: !include ../sensors/ADS8321.yaml
  dac0: !include ../actuators/AD5541.yaml
  dac1: !include ../actuators/AD5541.yaml

interfaces:
  dac0_analog:
    type: expose
    name: dac0_analog
    specification:
      passthrough: dac0/out
  dac1_analog:
    type: expose
    name: dac1_analog
    specification:
      passthrough: dac1/out
  adc0_analog:
    type: expose
    name: adc0_analog
    specification:
      passthrough: adc0/in
  adc1_analog:
    type: expose
    name: adc1_analog
    specification:
      passthrough: adc1/in

connections:
  - from: firestix/gpmc0
    to: fpga/gpmc0
  - from: fpga/spi0
    to: dac0/spi
  - from: fpga/spi1
    to: dac1/spi
  - from: fpga/spi2
    to: adc0/spi
  - from: fpga/spi3
    to: adc1/spi
```

Listing A.17: RaMstix board file

```
type: board
name: overo
components:
  dm3730: !include ../compute/dm3730.yaml
interfaces:
  gpmc0:
    type: expose
    specification:
      passthrough: dm3730/gpmc0
```

Listing A.18: Overo firestix board file

```
type: board
name: hypothetical

components:
  atmega: !include ../compute/atmega328.yaml
  adc0: !include ../sensors/ADS8321.yaml
  dac0: !include ../actuators/AD5541.yaml

interfaces:
  dac0_analog:
    type: expose
    name: dac0_analog
    specification:
      passthrough: dac0/out
  adc0_analog:
    type: expose
    name: adc0_analog
    specification:
      passthrough: adc0/in

connections:
  - from: atmega/spi0
    to: dac0/spi
  - from: atmega/spi1
    to: adc0/spi
```

Listing A.19: Hypothetical atmega based board file

A.5.3 Compute

```
type: compute
variant: fpga
name: compute
specification:
  clock: 100 kHz
model:
  variant: fpga
interfaces:
  gpmc0:
    type: mm
    variant: gpmc
    name: gpmc0
  spi0:
    type: serial
    variant: spi
    name: spi0
    specification:
      max_speed: 500 kHz
  spi1:
    type: serial
    variant: spi
    name: spi1
    specification:
      max_speed: 5MHz
```

Listing A.20: Generic FPGA compute file

```
type: compute
variant: fpga
name: cyclone3
specification:
  clock: 50 MHz
model:
  variant: fpga
interfaces:
  gpmc0:
    type: mm
    variant: gpmc
    name: gpmc0
  spi0:
    type: serial
    variant: spi
    name: spi0
    specification:
      max_speed: 5MHz
  spi1:
    type: serial
    variant: spi
    name: spi1
    specification:
      max_speed: 5MHz
  spi2:
    type: serial
    variant: spi
    name: spi2
    specification:
      max_speed: 5MHz
  spi3:
    type: serial
    variant: spi
    name: spi3
    specification:
      max_speed: 5MHz
```

Listing A.21: cyclone3 FPGA compute file

```
type: compute
variant: mcu
name: atmega328
specification:
  clock: 20 MHz
  adc:
    bits: 10
    maximum: 2.5V
    minimum: −2.5V
    trigger_delay_ticks: 13.5
    delay_ticks: 11.5
    clock: 156.25 kHz
  dac:
    bits: 8
    maximum: 2.5V
    minimum: −2.5V
model:
  variant: mcu
interfaces:
  spi0:
    type: serial
    variant: spi
    name: spi0
    specification:
      max_speed: 10MHz
  spi1:
    type: serial
    variant: spi
    name: spi1
    specification:
      max_speed: 10MHz
  adc0:
    type: analog
    variant: analog
    specification:
      direction: in
  pwm0:
    type: analog
    variant: analog
    specification:
      direction: out
```

Listing A.22: Atmega328 microcontroller compute file

```
type: compute
variant: linux
name: dm3730
specification:
  clock: 1 GHz
model:
  variant: linux
interfaces:
  gpmc0:
    type: mm
    variant: gpmc
    name: gpmc0
  eth0:
    type: network
    variant: ethernet
    name: eth0
    specification:
      throughput: 100 Mbit/s
```

Listing A.23: DM3730 processor compute file

A.5.4 Actuators

```
type: actuator
variant: DAC
name: actuator
specification:
  provides: dimensionless
  samplerate: 1 MHz
  bits: 16
  maximum: 5
  minimum: −5
model:
  name: dac
  variant: spidac
interfaces:
  spi:
    type: serial
    variant: spi
    name: spi
    port: digital
    specification:
      bits: 16
      max_speed: 50MHz
  out:
    type: analog
    variant: analog
    name: out
    port: analog
    specification:
      direction: out
      balanced: true
```

Listing A.24: Generic DAC actuator file

```
# The DAC available on the Ramstix board
type: actuator
variant: DAC
name: AD5541
specification:
  provides: voltage
  samplerate: 1 MHz
  bits: 16
  maximum: 5 volt
  minimum: −5 volt
model:
  name: dac
  variant: spidac
interfaces:
  spi:
    type: serial
    variant: spi
    name: spi
    port: digital
    specification:
      bits: 16
      max_speed: 50MHz
  out:
    type: analog
    variant: analog
    name: out
    port: analog
    specification:
      direction: out
      balanced: true
```

Listing A.25: AD5541 DAC actuator file

A.5.5 Sensors

```
# The ADC available on the ramstix board
```

```
type: sensor
variant: ADC
name: ADS8321
specification:
  provides: voltage
  samplerate: 100 KHz
  bits: 16
  trigger_delay_ticks: 5
  maximum: 5 volt
  minimum: −5 volt
model:
  name: adc
  variant: spiadc
interfaces:
  spi:
    type: serial
    variant: spi
    name: spi
    port: digital
    specification:
      bits: 24
      max_speed: 50MHz
  in:
    type: analog
    variant: analog
    name: in
    port: analog
    specification:
      direction: in
      balanced: true
```

Listing A.26: ADS8321 ADC actuator file

10 References

Arm Limited (2019). *Mbed OS*. URL: https://www.mbed.com/ (visited on 06/06/2019).

Ben-Kiki, Oren, Clark Evans, and Brian Ingerson (2005). “Yaml ain’t markup language (yaml™) version 1.1”. In: *yaml. org, Tech. Rep*, p. 23.

Bezemer, Maarten M (2013). “Cyber-physical systems software development: way of working and tool suite”. In:

Bilsen, Greet, Marc Engels, Rudy Lauwereins, and Jean Peperstraete (1996). “Cycle-static dataflow”. In: *IEEE Transactions on signal processing* 44.2, pp. 397–408.

Blochwitz, Torsten, Martin Otter, Johan Akesson, Martin Arnold, Christoph Clauss, Hilding Elmqvist, Markus Friedrich, Andreas Junghanns, Jakob Mauss, and Dietmar Neumerkel (2012). “Functional mockup interface 2.0: The standard for tool independent exchange of simulation models”. In: *Proceedings of the 9th International MODELICA Conference; September 3-5; 2012; Munich; Germany*. 076. Linköping University Electronic Press, pp. 173–184.

Broenink, Jan F and Yunyun Ni (2012). “Model-driven robot-software design using integrated models and co-simulation”. In: *2012 International Conference on Embedded Computer Systems (SAMOS)*. IEEE, pp. 339–344.

Broenink, Jan F, Yunyun Ni, and Marcel A Groothuis (2010). “On model-driven design of robot software using co-simulation”. In: *Proceedings of SIMPAR 2010 workshops international conference on simula-*

tion, modeling, and programming for autonomous robots. TU Darmstadt, Darmstadt, pp. 659–668.

Broenink, Tim Gerhard and Johannes F Broenink (June 2018). “A variable detail model simulation methodology for cyber-physical systems”. English. In: pp. 219–225.

Bruyninckx, Herman, Markus Klotzbücher, Nico Hochgeschwender, Gerhard Kraetzschmar, Luca Gherardi, and Davide Brugali (2013). “The BRICS component model: a model-based development paradigm for complex robotics software systems”. In: *Proceedings of the 28th Annual ACM Symposium on Applied Computing*. ACM, pp. 1758–1764.

Buck, Joseph Tobin and Edward A Lee (1993). “Scheduling dynamic dataflow graphs with bounded memory using the token flow model”. In: *1993 IEEE international conference on acoustics, speech, and signal processing*. Vol. 1. IEEE, pp. 429–432.

Dhouib, Saadia, Selma Kchir, Serge Stinckwich, Tewfik Ziadi, and Mikal Ziane (2012). “Robotml, a domain-specific language to design, simulate and deploy robotic applications”. In: *International Conference on Simulation, Modeling, and Programming for Autonomous Robots*. Springer, pp. 149–160.

Diego, Alonso, Cristina Vicente Chicote, Ortiz Francisco, Pastor Juan, and Álvarez Bárbara (2010). “V3cmm: A 3-view component metamodel for model-driven robotic software development”. In:

Harrand, Nicolas, Franck Fleurey, Brice Morin, and Knut Eilif Husa (2016). “Thingml: a language and code generation framework for heterogeneous targets”. In: *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems*. ACM, pp. 125–135.

Jensen, Jeff C, Danica H Chang, and Edward A Lee (2011). “A model-based design methodology for cyber-physical systems”. In: *2011 7th International Wireless Communications and Mobile Computing Conference*. IEEE, pp. 1666–1671.

Kahn, Gilles (1974). “The semantics of a simple language for parallel programming”. In: *Information processing* 74, pp. 471–475.

Lee, Edward A (1991). “Consistency in dataflow graphs”. In: *IEEE Transactions on Parallel & Distributed Systems* 2, pp. 223–235.

Likely, Grant and Josh Boyer (2008). “A symphony of flavours: Using the device tree to describe embedded hardware”. In: *Proceedings of the Linux Symposium*. Vol. 2, pp. 27–37.

Liu, He, Xiaojun Liu, and Edward A Lee (2001). “Modeling distributed hybrid systems in Ptolemy II”. In: *Proceedings of the 2001 American Control Conference.(Cat. No. 01CH37148)*. Vol. 6. IEEE, pp. 4984–4985.

Nicolescu, Gabriela, H Boucheneb, L Gheorghe, and F Bouchhima (2007). “Methodology for efficient design of continuous/discrete-events co-simulation tools”. In: *High Level Simulation Languages and Applications-HLSLA. SCS, San Diego, CA*, pp. 172–179.

Schlegel, Christian and Robert Worz (1999). “The software framework SMARTSOFT for implementing sensorimotor systems”. In: *Proceedings 1999 IEEE/RSJ International Conference on Intelligent Robots and Systems. Human and Environment Friendly Robots with High Intelligence and Emotional Quotients (Cat. No. 99CH36289)*. Vol. 3. IEEE, pp. 1610–1616.

Selic, Bran (2003). “The pragmatics of model-driven development”. In: *IEEE software* 20.5, pp. 19–25.

Singhoff, Frank, Jérôme Legrand, Laurent Nana, and Lionel Marcé (2004). “Cheddar: a flexible real time scheduling framework”. In: *ACM SIGAda Ada Letters*. Vol. 24. 4. ACM, pp. 1–8.

Stuijk, Sander, Marc Geilen, Bart Theelen, and Twan Basten (2011). “Scenario-aware dataflow: Modeling, analysis and implementation of dynamic applications”. In: *2011 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation*. IEEE, pp. 404–411.

Theelen, Bart D, Marc CW Geilen, Twan Basten, Jeroen PM Voeten, Stefan Valentin Gheorghita, and Sander Stuijk (2006). “A scenario-aware data flow model for combined long-run average and worst-case performance analysis”. In: *Fourth ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2006. MEMOCODE’06. Proceedings*. IEEE, pp. 185–194.

Vries, Theo JA de, Paul BT Weustink, and Johannes A Cremer (1997). “Improving dynamic system model building through constraints”. In: *submitted to CACD’97 Lancaster International Workshop on Engineering Design, Lancaster, UK*.

Wiggers, Maarten H, Marco JG Bekooij, and Gerard JM Smit (2008). “Buffer capacity computation for throughput constrained streaming applications with data-dependent inter-task communication”. In: *2008 IEEE Real-Time and Embedded Technology and Applications Symposium*. IEEE, pp. 183–194.