Automated analysis and simulation of control systems using dataflow

S. (Silke) Hofstra

MSc Report

Committee:
Dr.ir. J.F. Broenink
Dr.ir. J.B.C. Engelen
Ir. J. Scholten

December 2018

050RAM2018
Robotics and Mechatronics
EE-Math-CS
University of Twente
P.O. Box 217
7500 AE Enschede
The Netherlands

Summary

While modern embedded systems development offers many opportunities, the design of real-time control systems also poses many challenges. An example of such a challenge is the development process: embedded software engineers and control engineers need to work together to realize a control system, which can be complex from both a control engineering and software engineering standpoint.

To ease this development process, methods like model-driven development are used. Fitting within this process, this research proposes concrete workflow steps that allow a decoupling between the tasks of the control engineer and the software engineer. The workflow allows autonomy in the development process for the control engineer, who can create a platform-independent model, a platform-specific model and a realization using the tools available.

The goal of this research is to provide a method for converting a platform-independent model into a platform-specific model automatically, and using this platform-specific model to reach conclusions about the future realization.

A look is taken into the existing software solutions and what value they provide in the proposed workflow. Three existing modelling techniques are compared for suitability in the representation of platform-specific models: block diagrams, communicating sequential processes and synchronous dataflow

The proposed solution for the creation of platform-specific models is to convert a block diagram together with a platform and parameters into a synchronous dataflow (SDF) diagram. This SDF model can then be analysed using existing SDF methods for verification of the model. Simulation of this model is used for validation of the model.

This process is implemented in a proof-of-concept tool that can perform the conversion, analysis and simulation of configured models. The proof-of-concept tool is used in several example systems to show the functionality, and limitations, of the conversion process. This tool uses a special representation for the plant in order to simulate a full system.

The tool is verified by analysis against models with known properties. The limitations of the conversion process are shown in the analysis of an ADC element, which has more complex behaviour. It is shown that the tool is able to perform analysis and simulation of the behaviour of a simple PID system. It is also able to perform analysis, but not simulation, on a more complex control system.

In the end it is concluded that the proposed methods for conversion of a block diagram to a SDF representation work, and fit within the proposed workflow. The methods are currently limited in the representation of the interaction with the outside world.

In the future, research needs to be done into improvement of the simulation through co-simulation, improvements of analysis using external libraries, better representation of effects of the platform, and improved representation of components like ADCs and webcams.

It is recommended to continue the research into the proposed workflow by working towards a realization of the systems. This will provide insight into the accuracy of the created platform-specific model. Conversion from models in existing software and formats usable by the proof-of-concept tool also need to be implemented.

Contents

1 Introduction 1
- 1.1 Development of a control system . . . 1
  - 1.1.1 Architecture . . . 1
  - 1.1.2 Validation and verification . . . 2
  - 1.1.3 Workflow . . . 2
  - 1.1.4 Separation of concerns . . . 3
- 1.2 Proposed workflow . . . 4
- 1.3 Research . . . 5
- 1.4 Outline . . . 5

2 State of the art 6
- 2.1 20-SIM . . . 6
- 2.2 Simulink . . . 7
- 2.3 TERRA / LUNA . . . 8
  - 2.3.1 CλaSH . . . 8
- 2.4 Comparison . . . 9

3 Modelling techniques 10
- 3.1 Block diagrams . . . 10
- 3.2 CSP . . . 11
- 3.3 Synchronous dataflow . . . 13
- 3.4 Overview . . . 14
- 3.5 Choice of modelling technique . . . 15

4 Model-to-model conversion 18
- 4.1 Procedure . . . 18
- 4.2 Example . . . 19
- 4.3 Example of performed analysis . . . 21

5 Tool design 22
- 5.1 Functionality . . . 22
- 5.2 Model representation . . . 23
  - 5.2.1 Model . . . 23
  - 5.2.2 Task . . . 23
  - 5.2.3 Edges, tokens and types . . . 24
  - 5.2.4 Platform . . . 24
- 5.3 Simulation method . . . 24
  - 5.3.1 Plant representation . . . 25
- 5.4 Overview of functionality . . . 26

6 Execution 27
- 6.1 SDF validity . . . 27
- 6.2 Quantization and delay . . . 30
- 6.3 PID controller . . . 31
- 6.4 Embedded systems laboratory . . . 34

7 Conclusion & recommendations 37

7.1 Recommendations 38

A Configuration **39**
A.1 Test cases 39
A.2 Sine wave generator 42
A.3 PID controller 42
A.4 ESL model 44

Bibliography **46**

Chapter 1

Introduction

Modern embedded systems offer a great deal of opportunities for control engineers: cheap microcontrollers (MCUs) and field programmable gate arrays (FPGAs) allow the design of modular hard real-time control systems with more sensor inputs and more actuation signals than ever before. This means that improved workflows and toolchains for getting controller designs onto those embedded systems are more important than ever. The design of such embedded control systems, however, currently requires deep knowledge of both control engineering and embedded systems engineering. Control engineers often design control systems using domain-specific models, which then have to be implemented on the targeted embedded system by (embedded) software engineers. This requires a process with standardised knowledge, and compatibility, between the worlds of control engineering and software engineering.

Integration of the design process, where a control engineer is able to use a model and transform this directly to a working implementation, can lead to large improvement in the development time and effort of control systems. There are several toolchains that aim to do just this, such as the *Twente Embedded Real-time Robotic Application* (TERRA) (Bezemer, 2013), *20-sim* (Kleijn, Groothuis and H.G, 2017), and *Simulink* (MathWorks, 2017). Most of these toolchains target only general purpose processors or specific platforms, and not generally available microcontrollers and FPGA platforms. The design of 'hybrid' controllers, which run parts on different (communicating) platforms, is not possible.

1.1 Development of a control system

The end goal of the development process of a control system is to design and realize a control system for a real-world application. This design is constraint by the reality of software and hardware, and how these interact.

1.1.1 Architecture

Figure 1.1 shows the system architecture of a cyber-physical system—such as a control system. The figure consists of three parts (Broenink, Ni and Groothuis, 2010):

- *Embedded software* contains all developed software for the control system, this part represents both the critical (e.g. safety,

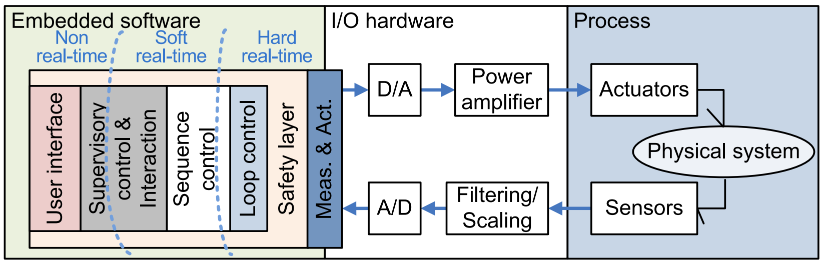


Figure 1.1: Detailed system architecture of a cyber-physical system (Broenink, Ni and Groothuis, 2010).

loop control) and non-critical (e.g. user interface) parts of the system. The embedded software runs on a platform, which can be anything from a single-board computer to a microcontroller to an FPGA. Note that, for the purpose of this report, this part includes any programmable part of an FPGA that does not perform input/output (I/O) directly.

- *I/O hardware* contains all interfacing from the digital to physical domain, and vice-versa. This hardware is usually present on the platform on which the software runs.
- *Process* or *plant* contains the physical system, the actuation that controls the physical system, and the sensors that measure the physical system.

Such a system is usually developed by a *control engineer* and an (embedded) *software engineer*: the control engineer creates models of both the controller and plant, and the software engineer creates an implementation of the controller on an embedded platform.

1.1.2 Validation and verification

Both the *validation* and *verification* of the system are important, these terms are defined as follows (IEEE, 2011):

- *Validation* is the assurance that the system meets the needs of the customer and other identified stakeholders. In concrete terms for a control system, this means that the system can control what is to be controlled (i.e. the plant). An important component of the validation process is simulation using a model of the plant, and testing of the system against the actual plant.
- *Verification* is the evaluation of whether or not the system complies with the requirements or specification. This is often an internal process. In the context of this report this is interpreted as verification that the realized system matches requirements and functionality of the modelled system.

1.1.3 Workflow

The development of a control system is often based on the principles of model-driven development (MDD) (Hailpern and Tarr, 2006) or model-driven engineering (MDE) (Schmidt, 2006). In this development process, the control engineer creates a platform-independent model (PIM) of the control system, and a model of the plant, using domain-specific modelling (DSM). In combination with the chosen platform, and working together with a software engineer, this information is then used to create a platform-specific model (PSM) as a basis of the implementation of the system.

Figure 1.2 shows the steps of the way of working to design control

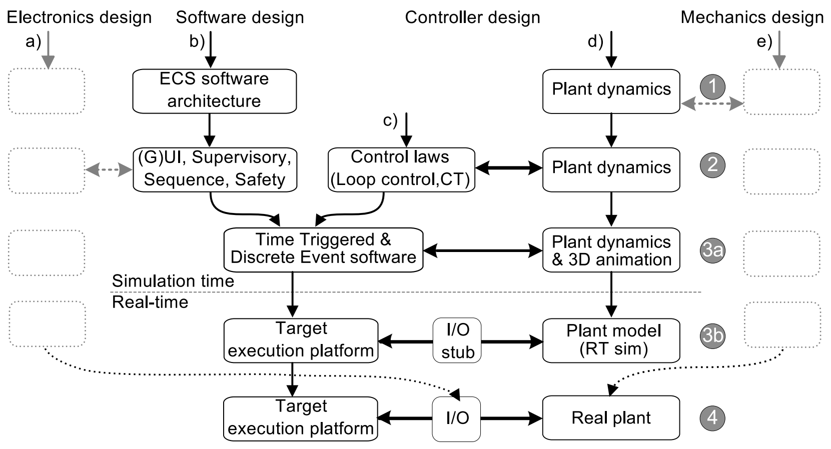

Electronics design
Software design
Controller design
Mechanics design
a)
b)
c)
d)
e)
ECS software architecture
(G)UI, Supervisory, Sequence, Safety
Control laws (Loop control,CT)
Time Triggered & Discrete Event software
Plant dynamics
Plant dynamics
Plant dynamics & 3D animation
Simulation time
Real-time
Target execution platform
I/O stub
Plant model (RT sim)
Target execution platform
I/O
Real plant
1
2
3a
3b
4

Figure 1.2: Concurrent design workflow (Broenink, Ni and Groothuis, 2010).

software for a cyber-physical system, as described in Broenink, Ni and Groothuis (2010). For the development of a control system into an application, branches *b* and *c* reflect the development of the controller. The model of the plant and then the plant itself, in the *d* branch, are used for *validation* of the steps: does the control system behave as required and intended? In the way of working shown, the work at the start of development is performed separately by the software engineer and control engineer: the software engineer (branch *b*) starts by designing an architectural overview and developing several high level tasks, while the control engineer is modelling the plant and applying control theory (step 1 and 2, indicated on the right). In the third step (3a and 3b) the controller design is realized for the target platform. The fourth and last step is the actual realization of the design. In this process, knowledge and experience about control engineering is required from the (embedded) software engineer, and knowledge about software engineering is required from the control engineer.

1.1.4 Separation of concerns

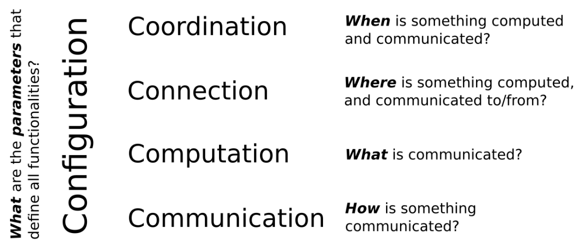

What are the *parameters* that define all functionalities?
Configuration
Coordination
When is something computed and communicated?
Connection
Where is something computed, and communicated to/from?
Computation
What is communicated?
Communication
How is something communicated?

Figure 1.3: The "Five Cs" of separations of concerns in the design of complex robotics systems (Bruyninckx et al., 2013).

The separation of concerns for a project can be described by the "5Cs", shown in figure 1.3 (Bruyninckx et al., 2013):

- *Computation* contains the implementation of the components of the system.
- *Communication* describes how data is communicated between components.
- *Connection* is the specification of how components should be connected.
- *Coordination* describes how the activities of components should work togther.
- *Configuration* contains the parameters of components, communication channels and other configurable elements of the system.

This is beneficial to structure the model-driven design of a control system, and provides a structure for the automation of (parts of) the process.

1.2 Proposed workflow

As a solution to the issues identified at the start of this chapter, a new workflow is proposed. The goals of this workflow are twofold: (i) give control engineers autonomy to transform models to code, and (ii) allow design of controllers spanning one or multiple platforms.

In order to reach these goals an approach is required that facilitates a structured and incremental approach to controller design, and allows control engineers to design a control system and implement it with minimal knowledge of the underlying platform.

The proposed workflow consists of the following stages:

1. *Controller design:* the control engineer designs a platform-independent model of the control system. This design is validated by simulation against a model of the plant.
2. *Target platform specification:* the target platform is specified by the control engineer, which—together with the platform-independent model of the previous step—leads to a platform-specific model of the control system that can be verified with analysis and validated by simulation against a model of the plant.
3. *Target platform code generation:* the platform-specific model is used to create a realization of the control system for the target platform. This realization can be deployed to the target platform, and tested against a model of the plant (e.g. via an I/O stub) or the real plant.

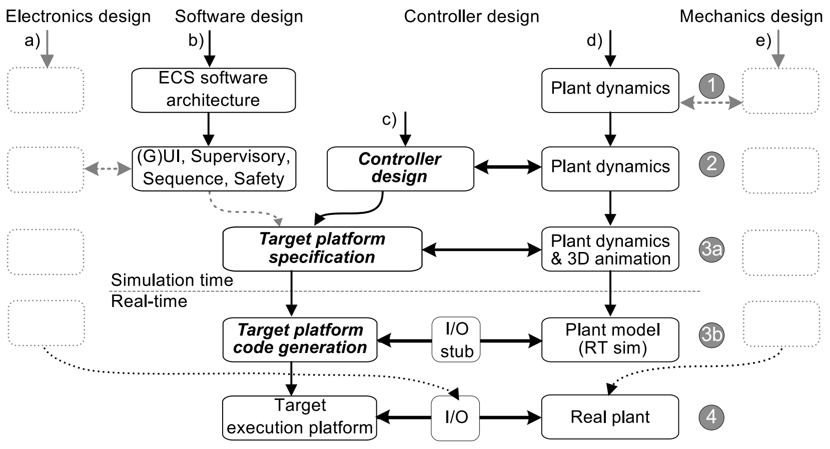


Figure 1.4: Proposed workflow shown in the workflow from figure 1.2. The three steps of the proposed workflow are highlighted in bold.

Figure 1.4 shows how this new workflow fits into the one shown in figure 1.2 with the proposed steps highlighted. While the steps remain mostly the same, the tasks of the software engineer and the control engineer shift: the software engineer designs and builds the tooling that allows the control engineer to apply their designs to the platform by themselves. The software engineer no longer has a direct part in the development of the system, which is represented by a grey, dashed, arrow from step 2 to 3a. In the original workflow, the software engineer and control engineer are both designing and developing the system, but in this workflow, the control engineer becomes the *user* of software to develop the system, while the software engineer becomes the *developer* of a toolchain that allows the control engineer to design the entire system by themselves.

1.3 Research

The focus of this report is on the second step of the proposed workflow: the analysis and validation of the system when given a platform-independent model and information about the platform. The first step—controller design—is covered by existing solutions (chapter 2), and the third step—target platform code generation—is left as future work. This means the following question needs to be answered: *how can one automatically create a platform-specific model based on a platform-independent model and platform information, and automatically perform analysis on, and simulation of, this platform-specific model?*

To answer this, the following questions need to be, and are, answered in the following chapters:

- What are existing software solutions that provide means to realize a platform-independent model on a platform, and in what regard are they suitable for analysis and simulation of the designed system? (Chapter 2)
- What is the most suitable modelling method for a platform-specific model for analysis and simulation? (Chapter 3)
- How can a platform-independent model be converted into an (analysable) platform-specific model? (Chapter 4)
- How can the constraints of the platform be represented in the platform-specific model? (Chapter 4 and 5)
- What information is required to automatically transform a platform-independent model into a platform-specific model? (Chapter 4)
- What meaningful feedback can be given to the control engineer based on analysis of the platform-specific model? (Chapter 4 and 5)
- How can the real-time behaviour of the platform-specific model be simulated? (Chapter 5)

1.4 Outline

Currently available software solutions that fit into the paradigms discussed in the previous sections are discussed in chapter 2.

Modelling methods used for the modelling of real-time and/or control-systems—block diagrams, communicating sequential processes (CSP) and synchonous dataflow (SDF)—are compared in chapter 3.

A way of converting a platform-independent model in the form of a block diagram into a platform-specific model in the form of SDF is proposed in chapter 4.

A proof-of-concept software tool that automatically performs this conversion and analysis on the resulting model is described in chapter 5.

The proposed methods and developed tool are evaluated in chapter 6 using various examples.

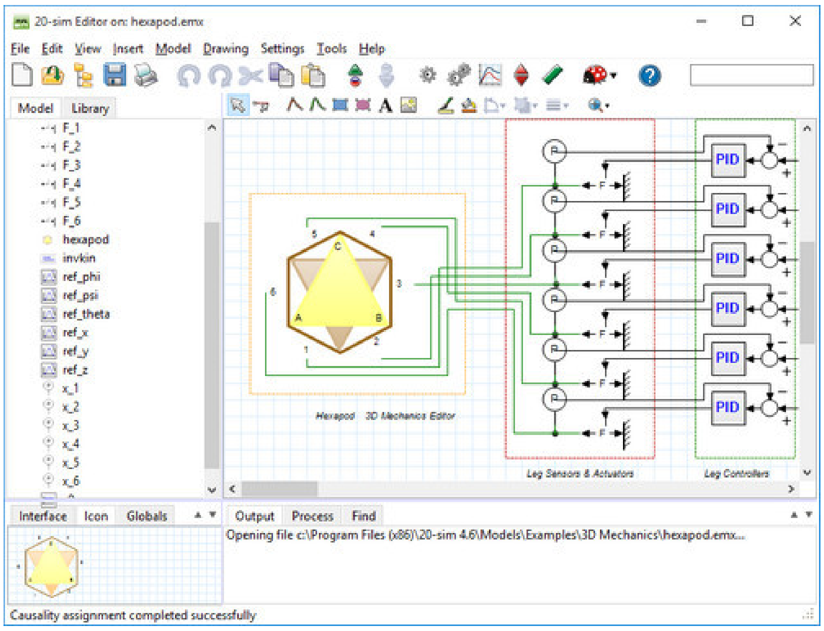
The results of this thesis are discussed in chapter 7.

Chapter 2

State of the art

The following sections describe the tools that are currently used for high-level—usually model-driven—design of (embedded) control systems. This includes both tools that are currently used for model-driven design of control systems, the techniques these tools use, and tools that exist for implementing control systems in software and/or hardware. The three tools in this comparison are *20-sim*, *Simulink* and *TERRA*. 20-sim and TERRA originate at the University of Twente and have been explicitly developed for control engineers (Broenink, 1997; Bezemer, 2013). 20-sim is being developed by Controllab Products. This contrasts with Simulink, the most popular commercial product in this space, which is a part of MATLAB and developed by MathWorks.

2.1 20-SIM



Opening file c:\Program Files (x86)\20-sim 4.6\Models\Examples\3D Mechanics\hexapod.emx...

Causality assignment completed successfully

Figure 2.1: Image of the 20-sim interface showing a robot model (Controllab, 2018).

20-sim is “a modelling and simulation program for mechatronic systems” currently developed and distributed by Controllab (Controllab, 2018), but originally developed at the University of Twente (Broenink, 1997). 20-sim allows the creation of models of both the controller and the physical domain using various types of diagrams like block diagrams and bond graphs, and enables in-depth analysis of the modelled control system and its properties. 20-sim can export models to MATLAB, Simulink, and C-code (Kleijn, Groothuis and H.G, 2017). The latter targets both embedded platforms like Arduino/AVR, and other applications.

20-sim 4C extends this functionality with a rapid-prototyping environment which allows real-time interaction with a control system, and allows the integration of generated C-code that can be used for interfacing with sensors and running the controller (Kleijn, 2013). The C-code requires an underlying operating system, and the rapid-prototyping platform requires a specific operating system with live communication.

20-sim (4C), therefore, has two supported approaches to deploying a model to a hardware platform:

1. Export of C-code and using this in a custom project. This still requires all details of the platform to be integrated by the developer, but can be used on any platform that can run C code: this includes soft-cores on FPGAs.
2. Using the 20-sim 4C rapid prototyping platform to run the model on a suitable platform. This requires almost zero knowledge of embedded systems, but only works on supported platforms like the TS-7300, or (with more knowledge required) a PC/104 compliant PC.
3. It is not possible to create a design targeting both embedded processor *and* FPGA.

2.2 Simulink

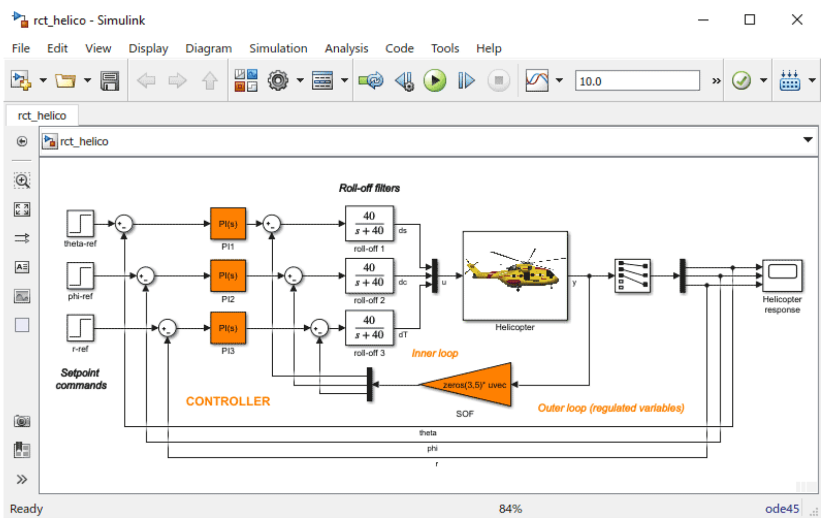


Figure 2.2: Image of the simulink interface showing a helicopter controller (Mathworks, 2018).

Simulink is an environment for modelling and simulating embedded systems, developed by Mathworks (MathWorks, 2017). Simulink uses block diagrams to model systems, where blocks can be integrated with custom-build MATLAB code.

Simulink Coder allows the generation and execution of C and C++ code from Simulink models. Supported hardware platforms for running code include many popular microcontrollers like *STM32 Nucleo* and *Arduino* boards (MathWorks, 2018). Simulink HDL Coder allows the generation of code targeting FPGAs (MathWorks, 2018).

This, again, allows two approaches for deploying models to a hardware platform:

1. Export of C, C++, VHDL or Verilog code based on the MATLAB/Simulink models to integrate in larger projects or run on unsupported hardware.
2. Using Simulink on a supported platform, which can be a GPP, MCU or FPGA.
3. It is not possible to create a design targeting both embedded processor *and* FPGA.

2.3 TERRA / LUNA

The Twente Embedded Real-time Robotic Application (TERRA) (Bezemer, 2013) is a tool suite for model-driven design of cyber-physical systems. It allows for design using *Communicating Sequential Processes* (CSP) and architecture models. TERRA aims to achieve a *first-time right* implementation. The model consists of a network of components with basic functionality provided by a *Generic Architecture Component* (GAC). TERRA has also been extended with simulation support (Lu, Ran and Broenink, 2016).

The LUNA Universal Network Architecture (LUNA) (Bezemer and Wilterdink, 2011) is an execution framework that enables the conversion of models to C++ code suitable for several operating systems. LUNA enables the hard real-time design for (CPU-based) hardware platforms, and operating systems. The code generation can be used to implement a controller on any (supported) platform.

2.3.1 CλaSH

CλaSH (clash-lang.org, 2018) is a functional hardware description language built on Haskell. CλaSH enables a relatively high-level approach to hardware design by allowing a functional behavioural description of the signal processing that should be performed. This high-level approach allows rapid development of any kind of signal processing, including control systems.

CλaSH has been integrated with the TERRA toolchain in order to easily develop a control system for it, but this has some limitations (Kuipers, 2017):

- Generated code still has to be adapted by the user to make it usable.
- Instrumentation has to be added by hand to make the setup testable.
- Adding instrumentation uses a relatively large amount of FPGA resources.
- It is not possible to create a design targeting both embedded processor *and* FPGA.

Summarised as (Kuipers, 2017):

> *FPGA design support in CλaSH is suitable [for, and] usable in[,] robotic applications, but at this point it is necessary for the user to have intricate knowledge of computer engineering. Furthermore, testing using co-simulation is not possible at this point.*

Solution	Controller design & simulation	Platform-specific design & simulation	Code generation
20-sim	Yes	No	Yes
Simulink	Yes	No	Yes
TERRA	No	Yes	Yes

Table 2.1: Overview of suitability of evaluated software for the creation and simulation of platform-independent controller models, platform-specific models and the generation of code for the target platform.

2.4 Comparison

All three software solutions fit within at most two of the steps in the workflow described in section 1.2, which can be seen in table 2.1. Both 20-sim and Simulink can perform modelling and simulation of given controllers and plants, and can create code for a limited number of platforms. TERRA can simulate a CSP equivalent of a controller and allows code generation.

Solution	GPP	MCU	FPGA
20-sim	C/C++[1]	C	Soft-core & C
Simulink	C/C++[1]	C[1]	VHDL/Verilog[1]
TERRA	C++[1]	n/a	n/a
TERRA+CλaSH	n/a	n/a	CλaSH

Table 2.2: Comparison of embedded systems knowledge required when using the examined software solutions for deploying models to platforms.

[1]: *Zero* embedded systems knowledge is required on a limited number of specifically supported platforms.

Table 2.2 shows the amount and kind of embedded systems knowledge required to implement a designed controller on general purpose processors (GPP, usually running Linux), microcontrollers (MCUs), and field programmable gate arrays (FPGAs). There is no software that fits completely within the workflow proposed in section 1.2.

Chapter 3

Modelling techniques

When modelling systems there is a trade-off between analysability and expressiveness: expressive models are hard to analyse, but allow modelling everything. Strict techniques are easy to analyse but can express less (Stuijk, Geilen, Theelen et al., 2011). For the modelling of control systems, three techniques are compared: block diagrams, communicating sequential processes and synchronous dataflow. All three are used in the development of control systems or the analysis of concurrent and/or real-time systems.

3.1 Block diagrams

Block diagrams are the most common way of modelling control systems. Block diagrams can represent the continuous-time and state space behaviour and operations of control systems in discrete elements, where the input-output relationships are indicated (Franklin, Powell and Emami-Naeini, 2006, p. 102).

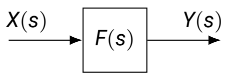


Figure 3.1: Example of a *block* in a block diagram with a transfer function $F(s)$ and mathematical relation $Y(s) = X(s)F(s)$.

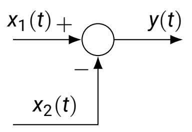


Figure 3.2: Example of a *summing point* in a block diagram, adding two signals with mathematical relation $y(t) = x_1(t) + x_2(t)$.

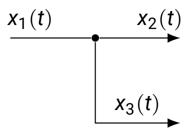
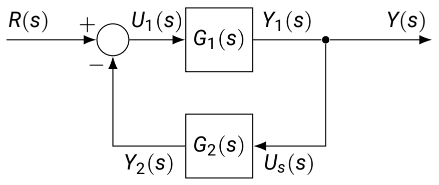


Figure 3.3: Example of a *take-off point* in a block diagram with mathematical relation $x_1(t) = x_2(t) = x_3(t)$.

The elements of block diagrams are composed of several elements:

- *blocks* (figure 3.1), which represent a transfer function operating on a signal.
- *arrows*, which indicate the signals and their direction.
- *summing points* (figure 3.2), represented by an open circle, which add or subtract signals. The circle is sometimes filled by a Σ or cross. Next to the incoming signals, addition or subtraction is indicated. This may also be used for *multipliers* (i.e. *modulators* and *mixers*), denoted with a cross.
- *take-off points* (figure 3.3), where a signal is split into multiple signals.

Figure 3.4 shows an example of a block diagram together with its corresponding equation. The equation can be derived from the block diagram by resolving the system of equations represented by the diagram (and vice-versa), which is left as an exercise to the reader.



$$Y(s) = \frac{G_1(s)}{1 + G_1(s)G_2(s)} R(s)$$

Figure 3.4: Block diagram representation and corresponding equation, adapted from Franklin, Powell and Emami-Naeini (2006).

Their usefulness in control engineering and other engineering disciplines is that block diagrams offer a useful insight into the functionality and behaviour of a system at a glance. This is not something the straight mathematical representation offers. Block diagrams, in this context, represent the continuous-time equivalent of both the control system and plant. However, because block diagrams capture a mathematical representation of the control system, no timing information is contained in the model.

The expressiveness of block diagrams allows modelling many software and hardware properties, as long as signal operations on a signal are involved. This applies equally to designing programs and FPGA solutions. Some software (see chapter 2) allows the designer to use block diagrams to develop for these platforms.

3.2 CSP

Communicating Sequential Processes (CSP) is a formal language for describing interaction in concurrent systems (Hoare, 1978). While these concurrent systems do not necessarily have to be computer systems, the application of CSP is often found in the creation, optimisation and analysis of concurrent computer programs. CSP allows a mathematical description of such systems, leading to provably correct designs.

CSP describes a system in terms of processes which communicate by sending or receiving data from unbuffered, synchronised, channels. CSP allows processes to run either sequentially or in parallel, and contains notation for boolean conditions, and (non)deterministic choice. CSP is usually written down as process algebra, but a graphical notation has also been developed (Jovanovic et al., 2004).

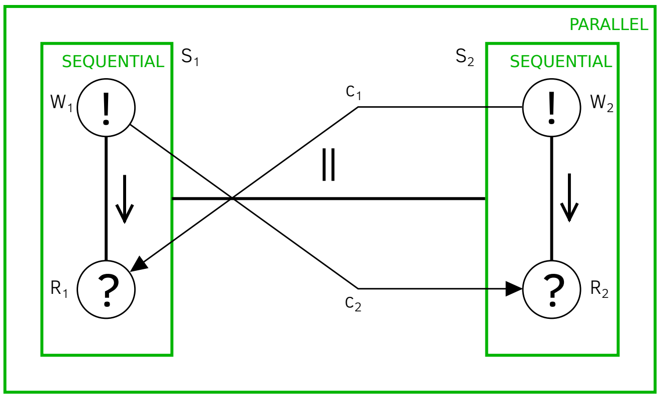


Figure 3.5: Example CSP diagram, adapted from Bezemer (2013).

Figure 3.5 shows a (graphical) example with two sequential groups (sequence direction indicated by a $\downarrow$) containing a reader process (R_1 and R_2, indicated by a ?), and a writer process (W_1 and W_2, indicated by a !). The sequential groups are executed in parallel (indicated by $||$). The groups communicate via two *channels* (notated with arrows). The equivalent (mathematical) notation is as follows:

$$P = S_1 || \{c_1, c_2\} || S_2 \tag{3.1}$$
$$S_1 = (c_2!x \rightarrow SKIP); (c_1?y \rightarrow SKIP) \tag{3.2}$$
$$S_2 = (c_1!p \rightarrow SKIP); (c_2?q \rightarrow SKIP) \tag{3.3}$$

This diagram contains a deadlock: the *writer* tasks cannot continue before data is received on the other end of the channel, but this requires the reader to be activated. The reader will not be activated because it can only be activated *after* the writer completes.

CSP finds it use in control engineering because (Bezemer, 2013):

- CSP allows synchronisation based on communication flow.
- CSP fits within the Component Port Connection (CPC) paradigm (Bruyninckx et al., 2013)
- A CSP model can be checked on correctness: formal verification allows checking for deadlocks or livelocks.

CSP applies natively to software application as it is a language that describes the behaviour of multi-process concurrent systems. In most programming languages the principles from CSP can be used to write and improve native applications, especially in languages with a concurrency model based on CSP like the Go language (Kourie et al., 2018).

When translating the CSP structures to an FPGA, the order of execution of the operations has to be preserved. In one application of CSP to FPGAs, this behaviour was preserved by creating a dataflow model for the synchronisation of processes (Kuipers, 2017):

The sequential and parallel structure data flow diagrams are shown in [figure 3.6]. The sequential operation is achieved by pipelining processes. When a sequential block receives a token, the token is forwarded to process P thereby activating it. When process P is finished it forwards the token to the next process in sequence, process Q. Finally, the last process returns its token to the sequential structure. The sequential structure then returns its token to its parent.

The parallel operator produces as much tokens as the amount of processes in parallel. This way all processes are activated simultaneously. After all processes in parallel have finished the parallel structure returns its token. This means the parallel structure has to collect all the tokens and return its own token only when all internal tokens are received.

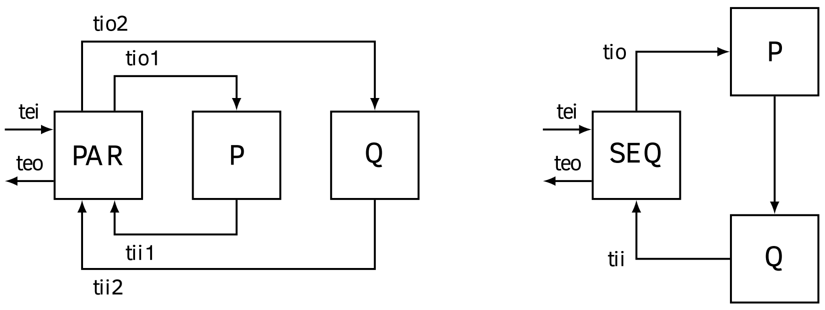


Figure 3.6: Data flow graphs of the parallel and sequential composition. Lines carry tokens. Processes are denoted as boxes (Kuipers, 2017).

Channels become bi-directional synchronous communication channels (Kuipers, 2017):

Although channels have one-way data communication, their synchronisation is bi-directional. A channel has bi-directional communication to ensure proper functionality. For example, a writer block may only finish (return its token) when its value is received.

3.3 Synchronous dataflow

Dataflow is a way of modelling systems by describing the operations of a stream of data, usually expressed via dataflow diagrams. The dataflow model is closely related to the Kahn Process Network (KPN) model (Lee and Parks, 1995). Dataflow is used for the design, analysis and optimisation of concurrent real-time systems, and data processing applications (M. Bekooij et al., 2005).

Dataflow diagrams—such as the one shown in figure 3.10—are expressed in terms of *actors* (also known as *tasks*) often indicated by v (figure 3.7). Actors perform an action taking a constant firing duration (ρ), edges (also known as *queues*) that represent buffered unbounded communication channels and *tokens* that represent a model of data in such a channel. Actors require a fixed number of tokens on all incoming edges before they can be executed (γ), and produce a fixed number of tokens on all outgoing edges (π). Edges can contain initial tokens (indicated by a dot and/or δ). No explicit value for π, γ and δ means the corresponding value is 1. A cyclic dependency of a number of actors is called a *cycle*.

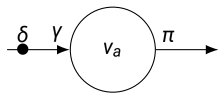


Figure 3.7: Dataflow task with a consumption of γ on the incoming edge, and a production of π on the outgoing edge. The incoming edge has δ initial tokens.

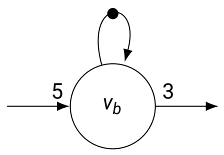


Figure 3.8: An SDF task with a self-loop in order to limit parallel execution. The consumption and production on the self-loop is 1.

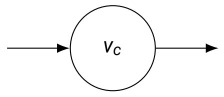


Figure 3.9: An HSDF task. The consumption and production on the incoming and outgoing edges is 1.

An actor can be executed multiple times in parallel, as long as a sufficient amount of tokens is available on all edges. The parallel execution of a task can be limited by adding an edge from the task to itself—a *self-loop*—with an initial token (figure 3.8). This limits the parallel execution of the task by creating a dependency on the previous execution of the task.

Synchronous dataflow (SDF) is a variant where the number of data samples that are produced and consumed is specified beforehand (Lee and Messerschmitt, 1987). Synchronous data flow allows algebraic analysis of real-time properties, including deadlock freedom, and periodic behaviour.

Homogeneous synchronous dataflow (HSDF) is a subset of SDF where the number of tokens produced and consumed by a task is always one (figure 3.9). HSDF is easier to analyse than SDF, but less expressive. SDF models can, however, be converted into an HSDF model (M. Bekooij et al., 2005, p. 83).

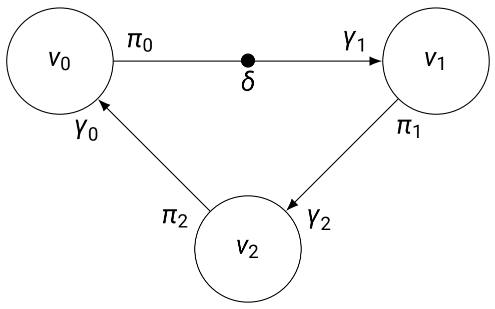


Figure 3.10: Example of an SDF diagram (based on M. Bekooij et al. (2005)) containing three actors, (v_0, v_1, v_2) and production (π), consumption (γ) and initial tokens (δ) indicated.

Figure 3.11 shows the example SDF diagram from figure 3.10 with concrete values for the production and consumption of tokens on the edges, and the firing duration for each actor indicated. This diagram has three actors (v_1, v_2 and v_3) and three edges.

Analysis of this graph shows that it is *consistent*: after a certain amount of firings the state will be back to its initial state. This is only possible if no tokens accumulate on an edge, and no task is starved of tokens (causing a deadlock). For any *consistent* graph it is possible to calculate the maximum number of tokens that will ever accumulate on an edge, allowing edges to be implemented using buffers of a fixed size.

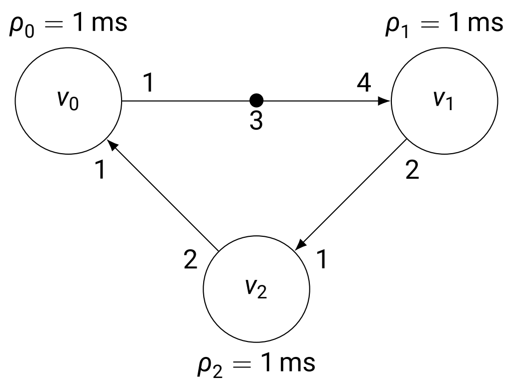


Figure 3.11: SDF diagram in figure 3.10 with firing durations indicated (ρ) and values for production, consumption and initial tokens filled in.

The graph, however, is not *deadlock free*: there are not enough initial tokens in the cycle (consisting of all tasks) to allow any actor to fire.

(H)SDF diagrams allow modelling of both concurrency—using parallelism—and task dependencies—using edges and production of tokens. This applies equally to software and hardware applications, as long as a system is *predictable* (M. Bekooij et al., 2005). Not every system is predictable: a hardware system that depends on non-regular input is not predicable, for example. Unpredictable systems cannot be modelled accurately. This is especially true for software running on general purpose processors: delays may be incurred when a memory cache is missed, or a task is preempted by the operating system. In software, therefore, the *worst case execution time* is used, which can be estimated from a statistical set of execution times.

3.4 Overview

Table 3.1 shows the analysability of the modelling techniques compared in the previous sections with respect to the following items to analyse:

- *Data dependency*: the dependencies of tasks with regards to the data they process. Does the diagram express that A should always be applied before B?
- *Deadlock*: is it possible for the system to be blocked, or stops executing, because of mutual data dependencies or other reasons? If a deadlock occurs a control system will halt completely.
- *Livelock*: is it possible for the system to be stopped because one item does not stop executing? If a livelock occurs the control system will seem to halt while the internal processes keep executing.
- *Scheduling*: is it possible to create a schedule for the execution of operations for tasks? Is it possible to check if a system can, in fact, be scheduled? This information is important for estimating if the system can be deployed on the target system. Without a schedule it is impossible to know if all tasks can be executed and what the input/output latency is.
- *Resource contention*: it is possible to analyse mutual access to a shared resource through the model? In a control system, resources like a communication bus or the same memory of the target platform may be accessed by the system. Some tasks may have to wait for a contended resource, leading to increased latencies.

- *Throughput*: can the throughput (i.e. data samples per second) of the model be calculated? The throughput puts an upper bound on the performance of the system. If this upper bound is too low for the needs of the control engineer, either the platform or the design need to be changed.
- *I/O latency*: can the latency between the inputs and outputs of the system be determined from the model? High I/O latency will interfere with the execution of the control system.

Analysis	Block Diagram	CSP	(H)SDF
Data dependency	✓	✓	✓
Deadlock	–	✓	✓
Livelock	–	✓	–[1]
Scheduling	–	✓[2]	✓
Resource contention	–	✓	✓[3]
Throughput	–	✓[4]	✓
Latency	–	–	✓

Table 3.1: Comparison of examined modelling methods.
[1] (H)SDF tasks always finish executing making livelock impossible.
[2] Timed CSP only (Oguz, Broenink and Mader, 2012).
[3] Needs to be explicitly modelled. Probabilistic contention is complex (Kumar et al., 2007).
[4] Complex (Woodside, 1989)

3.5 Choice of modelling technique

The chosen modelling technique needs to:

- Allow analysis of the feasibility of the system: is the system valid, does it perform the required functionality, does it contain deadlocks, etc.
- Allow analysis of the schedulability of the system. This allows insight into the question whether all the tasks that need to be performed can be performed in the available computing time.
- Allow analysis of the timing and delays. This allows insight into the delay between incoming and outgoing samples, and which operations cause the most delay.
- Allow translation into an optimal software solution.
- Allow translation into an optimal FPGA solution.

Requirement	Block Diagram	CSP	(H)SDF
Feasibility	$-$	$+$	$+$
Schedulability	$-$	$-$	$+$
Timing/delays	$-$	$-$	$+$
Software app.	$\pm$	$+$	$+$
FPGA app.	$\pm$	$\pm$	$+$

Table 3.2: Evaluation of the suitability of block diagrams CSP and (H)SDF.

Table 3.2 shows an evaluation of the suitability of block diagrams, CSP and (H)SDF as modelling methods for the analysis and simulation of a system. The suitability for a requirement is rated on a scale consisting of $-$ (unsuited), $\pm$ (somewhat suited), and $+$ (well-suited). Reasons for the ratings are as follows:

- *Block diagrams* only allow insight into the mathematical validity of the model. While data dependencies are modelled, no information is present to establish things like deadlock and consistency (in the SDF sense) of the model. Because of the lack of timing information, no schedule can be constructed and no delays determined. Block diagrams, being a series of operations on a signal, can be converted into software relatively easy. The resulting software, however, is not necessarily optimal. It is somewhat unsuited to a hardware application though, as there is no information about the execution time of operations. This means that there is no easy way to create a performant, well-synchronized system that conforms to the equations.

- *CSP* does allow insight into the validity of the model in the form of deadlock and livelock freedom. Constructing a schedule is somewhat more complex, but possible (Oguz, Broenink and Mader, 2012). Calculating the timing and delays may be possible from this schedule, but requires new research into this area. CSP, however, is especially suited to developing concurrent software, this being its *raison d'être*. There exists previous research, however, that shows that applying CSP to hardware does not necessarily lead to an efficient implementation (Kuipers, 2017).
- *HSDF and SDF* do allow insight into the deadlock-freedom of a model, and its *consistency*. SDF, being a more expressive form of dataflow, is slightly less easy to analyse. From a consistent model a self-timed schedule can be determined. Timing information is an explicit part of a dataflow model. Engineering software and hardware with dataflow models is also relatively straight forward.

Dataflow in the form of SDF or HSDF is more suitable than CSP for the (static) analysis of a platform-specific model, and is better suited for modelling hardware targets. Block diagrams are entirely unsuitable for analysis platform-specific models. However, as control engineers tend to model in block diagrams—block diagrams *are* more suitable for control theory—a model-to-model conversion from block diagrams to SDF will allow a 'best of both worlds' approach: control engineers can work with modelling methods that are suitable for their domain, while the real-time analysis and synthesis is performed using methods more suitable for real-time analysis.

In section 3.3 it was explained that a more expressive model usually leads to a model that is harder to analyse. This is also the case for SDF over HSDF: the *maximum cycle ratio* (MCR) of an HSDF diagram can be calculated with a polynomial algorithm, but this is not the case for SDF diagrams (M. Bekooij et al., 2005, p. 10). It would therefore be beneficial to limit the modelling technique to HSDF in order to maintain the most analysable model possible. For almost all equivalent elements in a block diagram, this is possible: a mathematical operation usually does not operate in a way where it consumes multiple values from the same input at once. Such mathematical operations can therefore be represented by an HSDF task (figure 3.12).

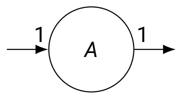


Figure 3.12: Example representation of the equivalent of a *block* in HSDF.

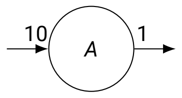


Figure 3.13: Example SDF representation of downsampling with a factor 10.

The main exception to this is resampling: as every sample is represented by a token in an SDF-model, an SDF task equivalent needs to consume and produce tokens in a ratio equal to the resampling ratio (figure 3.13). Resampling elements are sometimes explicitly defined in models, but can also be required when having a system with different sampling frequency domains that communicate between these domains. This means that the modelling technique has to be SDF, but HSDF analysis techniques can be applied as long as there are no resampling elements involved or the resampling elements are converted to HSDF first.

The chosen technique for modelling the system is therefore SDF, as this allows most analysis while still being able to represent the system to be realized.

Chapter 4

Model-to-model conversion

Because of the benefits of allowing control engineers to create platform-independent models in block diagrams, but creating the platform-specific model in SDF (see section 3.5), it is required to perform a *model-to-model* conversion from block diagrams to SDF.

4.1 Procedure

The procedure for translating a block diagram into SDF is as follows:

1. Ensure all elements in a diagram are defined. If the input and outputs of the system are not defined (but only marked as $u(t)$, for example), the system cannot be fully analysed.
2. Convert all blocks and points to tasks:
   - Blocks operate on a single input and have a single output (e.g. integrator).
   - Points have multiple inputs and a single output (e.g. summing point), or vice versa (e.g. take-off point). Because tasks consume and produce on all edges the former consume multiple tokens and produce one, and the latter consume a single token and produce multiple.

   Keep the connecting arrows as *edges*.
3. Add a self-edge to elements that cannot be executed multiple times in parallel with itself (e.g. sampling input). This also applies to tasks that operate on an internal variable (like integration elements) to avoid race conditions: the token on this self-edge represents the state of the internal variables of the task.
4. Add initial tokens to edges that require initial values. This includes all self-edges, but also one token per feedback loop.
5. Add execution times to the tasks. These execution times are based on the worst-case execution time of a task on the targeted platform.

4.2 Example

Figure 4.1 shows an example block diagram. The following steps are taken to convert this model:

1. The input and output signals, $u(t)$, $x(0)$ and $y(t)$, are converted to explicit *source* and *sink* elements (figure 4.2).
2. All blocks and points are converted to tasks (figure 4.3).
3. The *sources* ($u(t)$, $x(0)$) are given a self-edge (figure 4.4).
4. Initial tokens are given to the self-edges and the output of the integration task (figure 4.5).
5. Execution times from the task types, fictional in this example, are added to the diagram (figure 4.6).

This model is a valid HSDF model (because all tasks produce and consume one token), and can therefore be fully analysed.

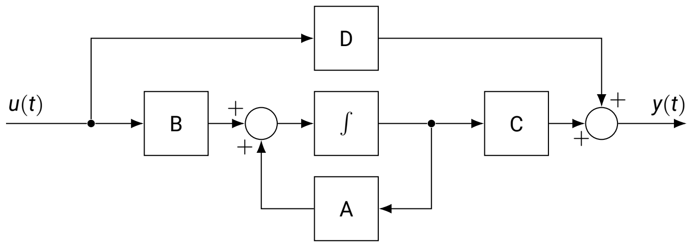


Figure 4.1: Example of a block diagram. Adapted from *Control Systems/Block Diagrams* (2018).

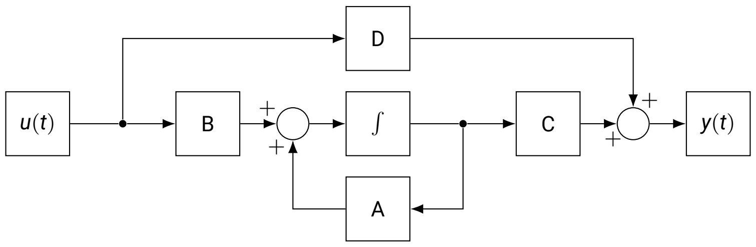


Figure 4.2: Block diagram of figure 4.1 with explicit source and sink elements.

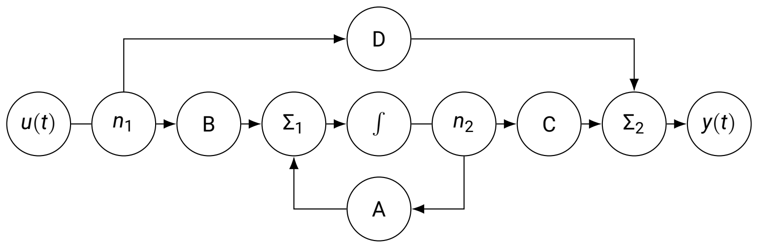


Figure 4.3: SDF diagram created from figure 4.2 by converting elements to tasks.

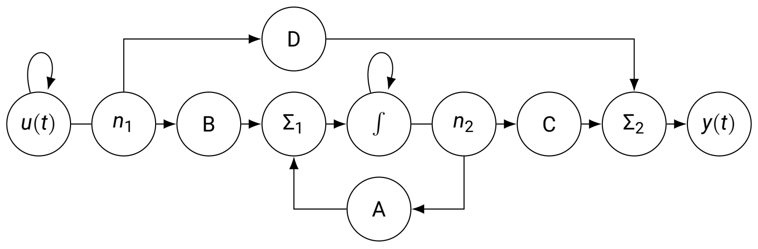


Figure 4.4: SDF diagram of figure 4.3 with self-loops.

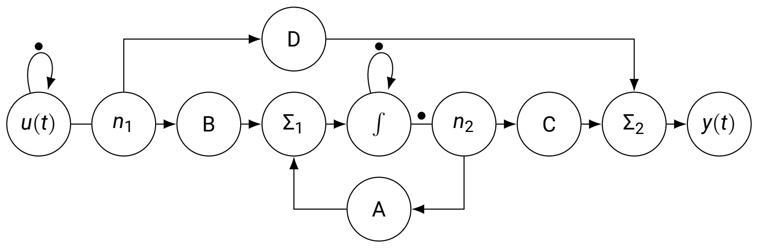


Figure 4.5: SDF diagram of figure 4.4 with initial tokens.

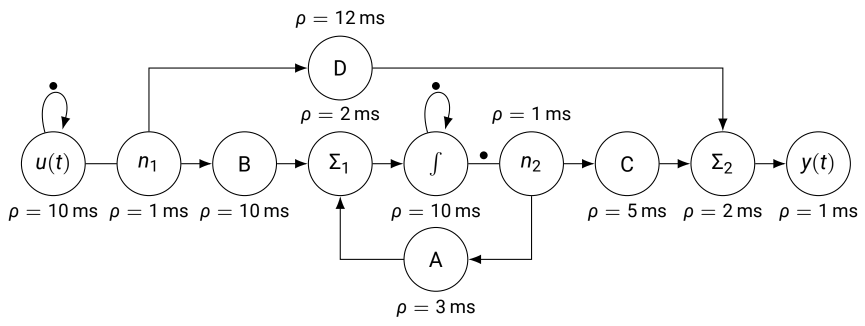


Figure 4.6: SDF diagram of figure 4.5 with firing duration.

4.3 Example of performed analysis

The model in figure 4.6 can be analysed by standard HSDF methods.

First the validity of the model is analysed. This is simply checking if all edges are connected, ergo: the model is *valid*.

Mathematically, a connected and consistent model is guaranteed to have topology matrix (Ψ) with a rank one less than the amount of tasks in the graph ($|V|$) (M. Bekooij et al., 2005, p. 79). This always holds true for an HSDF graph, which the diagram in figure 4.6 is.

A deadlock will occur when there is a cycle without initial tokens. The cycles in figure 4.6 are: $(u(t))$, $(\int)$, and $(\Sigma_1, \int, n_2, A)$. All of these cycles contain an initial token, meaning the model contains no deadlock.

The *maximum cycle ratio* (MCR, μ) for graph G (being an HSDF diagram), with cycles $C(G)$, tasks $V(G)$ and edges $E(G)$ can be calculated with the following equation (M. Bekooij et al., 2005, p. 10):

$$\mu(G) = \max_{c \in C(G)} \frac{\sum_{v \in V(c)} \rho_v}{\sum_{e \in E(c)} \delta_v} \tag{4.1}$$

This finds that the *critical cycle* of the system is $(\Sigma_1, \int, n_2, A)$ with an MCR of 16 ms. This represents the *minimum period* in the periodic schedule for this system. In this diagram this means that the control system should not have a higher sampling rate than $\frac{1}{16\,\text{ms}}$: because inputs $u(t)$ and $x(0)$ have a firing duration that is lower (ergo: a frequency that is higher), the number of tokens on *some* edge will grow exponentially.

By calculating the start-times of the actors in the SDF diagram, the input/output-delay can be calculated. This is straightforward in the case of the diagram in figure 4.6: assuming $u(t)$ starts at $t = 0$ ms, the finish time of $y(t)$ is $t = 42$ ms (the sum of all tasks in the longest path between input and output). This should be smaller than a single sample period as well, making the minimum sampling period 42 ms, equivalent to a maximum sampling frequency of approximately 23.8 Hz.

The proposed model-to-model conversion technique, therefore, leads to a more strictly defined, and platform-specific, model of the control system. This platform-specific model can be analysed using existing SDF and HSDF methods.

Chapter 5

Tool design

The technique for converting a block diagram is embedded into a proof-of-concept tool with the ability to perform static analysis and simulation. This allows control engineers to make use of the added insight provided by the model conversion to dataflow. The proof-of-concept tool is implemented using Python.

5.1 Functionality

In order to attain the goals set out in section 1.3, the tool should have the following functionality:

- Have an annotated explicit block diagram as input.
- Have a standardised system of annotations that include all information required for modelling and simulation.
- Be able to convert the given model into a dataflow (SDF) model automatically.
- Be able to perform static analysis on the real-time properties of the model.
- Be able to give clear feedback on the result of the analysis.
- Be able to simulate the real-time behaviour of the model.
- Be able to perform run-time sanity checks during the simulation.

This can be combined with the “5Cs” (section 1.1.4) to separate the components of the tool into the the following, modular, parts:

- A *configuration file* containing information provided by the control engineer: the blocks in the diagram with their types, parameters and connections, and the target platforms for the control system.
- A *component library* containing task abstractions and implementations of elements in the block diagram.
- The *core library* containing everything for creating and analysing the model.

In “5C” terms the *configuration* and *connection* of the system are provided by the control engineer. The *communication* and *coordination* is part of the *core library*, developed by the software engineer. The *computation* is part of the *component library*, which is also developed by the software engineer. The separate implementation of the component models from the core allows the range of available components to be extended without influencing core functionality.

5.2 Model representation

The representation of a *model* is composed of a set of *tasks* connected by *edges*. This representation is close to the SDF definition of the model, but extended by the properties required for simulation. Figure 5.1 shows an SDF model similar to the one from figure 3.10 as modelled by the tool.

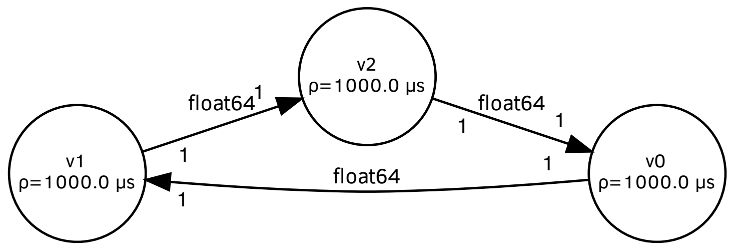


Figure 5.1: SDF model based on figure 3.10 as modelled by the tool. The production and consumption of tasks is indicated on the beginning and end of the edges. The type and number of initial tokens on edges (δ, omitted if zero) are indicated in the middle of the edges.

The inputs and outputs of tasks are not explicitly modelled as separate components, but instead kept in a list within the task itself. This allows tasks to communicate via these inputs and outputs, without having to add additional logic for interfacing inputs and outputs with the corresponding tasks and edges.

5.2.1 Model

The configuration contains three top-level sections:

- *name* contains a unique identifier for the model.
- *platform* configuration that configures what platforms the system is deployed to, and what the settings of this platform are. This configuration allows for more accurate estimation of the performance of a certain component based on the platform.
- *model* configuration that contains the definitions and configuration for the defined elements of the model (the *tasks*), and how they are connected.

In the example shown in listing 5.1, the name of the model is 'test'. Tasks will be analysed as if they run on a Raspberry Pi clocked at 1 GHz by default. The model contains two tasks.

```
name: test

platforms:
    raspberry_pi:
        default: true
        clock_speed: 1 GHz
    nucleo_f411re:
        clock_speed: 100
            MHz

model:
    task_1: ...
    task_2: ...
```

Listing 5.1: Example configuration for a model.

5.2.2 Task

A task is the representation of an SDF actor containing all information required for static analysis, and simulation.

A task is specified with the following required information:

- A *name* used for identification of the task.
- A *type* that determines the implementation of the task.
- *Parameters* that configure this task.
- Where the *outputs* of the task connect to.

Tasks are also coupled to a *platform*. This allows the tasks to use the properties of the platform to accurately represent the worst case execution time (WCET) and data type for analysis.

All *elements* are implemented as extensions of a basic *task* in the component library. This allows a few common parameters to be set, mainly *WCET* (equivalent to the firing duration), and value logging configuration. This basic task is part of the *core* of the tool, allowing elements to be created with minimal effort.

Listing 5.2 shows an example of the configuration for a PID element. The name of the element is determined by the key (`pid_1`) of this configuration, the type is `pid`. The PID element is configured with parameters $K_p = 1$, $T_i = 10$ and $T_d = 0.5$. The output of the PID controller is connected to the input '`in`' of the task named '`dac`'.

```
pid_1:
    type: pid
    platform: raspberry_pi
    params:
        kp: 1
        ti: 10
        td: 0.5
    outputs:
        out: dac.in
```

Listing 5.2: Configuration for a PID element.

5.2.3 Edges, tokens and types

An *edge* is modelled as a typed unbounded FIFO containing tokens. Having unbounded capacity is sufficient for the simulation purposes. Because a consistent SDF graph will have a limited number of tokens on any edge, this amount can be constraint for the synthesis. The edge is created in the source task and then connected to the destination task when the model is created, allowing only the output connections to be specified. Edges perform a type check when a token is added to the the edge, in order to detect programming or design errors.

Tokens are simple objects containing a data type, an inception time and a value. The token performs active conversion of the value on its creation in order to model behaviour like quantization. The inception time can be used to see the lifetime of tokens but also, more importantly, to calculate the start time of tasks.

How a value is converted is modelled by the *data type* that contains a type name, and parameters noting its properties—like the number of bits and if the data type is numeric. These properties allow tasks to change their execution model in order to better reflect reality: performing an operation on a thousand bits is often slower than performing it on eight.

5.2.4 Platform

Platforms contain the properties of a hardware platform. This is currently limited to the *clock speed* of the platform, but can be expanded to contain information about properties like the availability of floating point units, performance of specific operations (e.g. multiplication and addition), word size, and available inputs and outputs of the platform. This could also influence the type that a task uses for its calculations: a 32-bit floating point value on a 32-bit platform, but a 64-bit floating point value on a 64-bit platform, for example.

5.3 Simulation method

Existing simulation solutions for SDF, like HAPI (Kurtin, Hausmans and M. J. Bekooij, 2016), are mainly focussed on the activation of tasks and the resulting schedule. While this gives a good indication of the performance of a system modelled by SDF, this contains no information about the values embedded in the tokens.

Because of this, the tool contains custom-build simulation for the calculation of the values of tokens. These token values can then be shown in a plot. The properties of an SDF model are used to perform a simulation on the model:

- Before a task can be executed, all its incoming edges need to contain at least as many tokens as the task consumes on the respective edges.
- The task runs for a fixed amount of time, after which the task produces a certain amount of tokens on its outgoing edges.

Before simulation all tasks are added to a queue. Tasks in the queue are simulated as follows:

1. Check if the task is ready. If not, skip this task to simulate other tasks.
2. Consume all tokens and perform operations using the values.
3. Produce tokens with the result of the operations. The new tokens have an inception time that is the maximum of the inception times of the incoming tokens.
4. Remove this task from the queue if the inception time of the produced tokens is greater than or equal to the simulation stop time. Go back to step 1 otherwise.

This method results in simulation of the target system in a worst-case scenario. The accuracy of the simulation depends on the accuracy of the WCET estimates and the precision of the data types. Because of the software used, the accuracy of both time and simulated values, however, is limited to a double precision floating point value. No problems are expected because of this limitation.

All tasks have a common parameter named '`log`' that allows tasks to log values to either a file, or a plot. This plot is constructed after the simulation is finished, and contains all output values of the tasks that log measurement data for use in the plot.

5.3.1 Plant representation

Modelling a plant is required for creating an accurate simulation. There are two choices for simulation: (i) co-simulation against simulation software, or (ii) perform the full simulation of the plant within the tool itself. The first option has preference because there is sufficient software available for simulation, and this can perform simulation of significantly complex models and the internal dynamics of the plant. Not implementing simulation also has the benefit of keeping the tooling simple and to the point, applying the *KISS* principle (Hanik, 2006).

Because interfacing with simulation software brings along its own complexity, however, co-simulation has not yet been implemented.

In order to be able to perform simulation against a plant, a *plant* task (figure 5.2) needs to be created. This task is marked as being analog, and needs to use inputs and outputs of type *analog* as well. The plant outgoing edge is automatically given sufficient initial tokens (indicated by δ) as to limit the influence to the analysis of the rest of the system. With this implementation the plant is only influenced by tokens. Because of this, the assumption has to be made that the output of the controller is constant between updates. The plant also has a firing duration of 0 to reflect the analog nature of the plant.

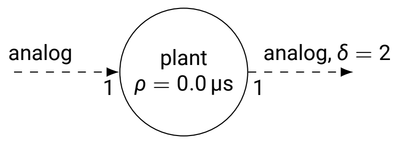


Figure 5.2: Plant 'task' with two initial tokens on the outgoing edge.

5.4 Overview of functionality

A proof-of-concept tool is implemented using the “5Cs” for the structure of the configuration files and program libraries. The proof-of-concept can perform the following functions:

- Create a representation of an SDF-based platform-specific model based on a configuration file with a platform-independent model and parameters.
- Create a representation of a plant in a limited capacity.
- Perform analysis on this platform-specific model using the methods described in section 4.3.
- Perform simulation on the given model.

Chapter 6

Execution

The following sections show some analysis of simple control systems and how analysis can be performed on them. All configuration for the models discussed can be found in appendix A.

6.1 SDF validity

To check if the SDF properties are correctly analysed, several examples from M. Bekooij et al. (2005) have been modelled as test cases. The definitions of these test cases can be found in appendix A.1. For each case the following happens when checking the model with the developed tool:

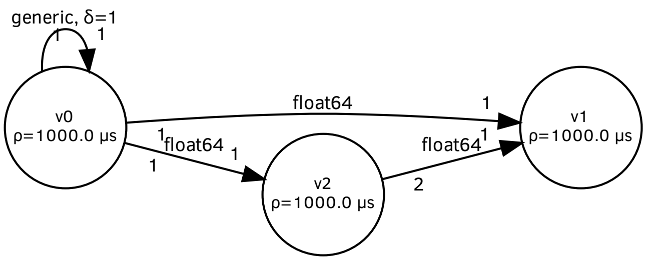


Figure 6.1: Example of an inconsistent model, adapted from M. Bekooij et al. (2005).

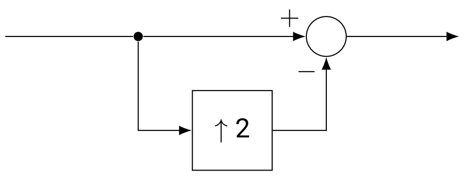


Figure 6.2: Example of a block diagram that can result in the situation in figure 6.1.

Figure 6.1 shows a model that is not *consistent*: the number of tokens on the edges do not return to the initial state after a certain number of executions. This can be caused, for example, by connecting a signal before upsampling to the signal after upsampling (figure 6.2). The tool gives the following result:

```
Result: model is not consistent
Stopping analysis because of problems with the model
```

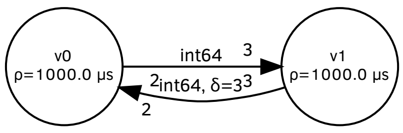


Figure 6.3: Example of a consistent model with deadlock, adapted from M. Bekooij et al. (2005).

Figure 6.3 shows a model that is consistent but contains a deadlock because of insufficient tokens available on the edge from v_1 to v_0. This results in the following output:

```
Result: model is not deadlock free
Stopping analysis because of problems with the model
```

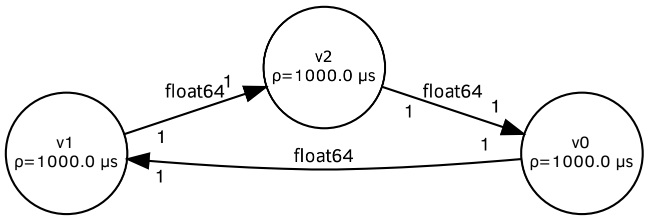


Figure 6.4: Example of a consistent model with deadlock, adapted from M. Bekooij et al. (2005).

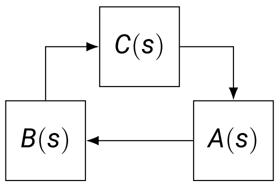


Figure 6.5: Example of a block diagram that leads to the situation in 6.4.

Figure 6.4, used as example SDF graph in earlier chapters, shows another model that is consistent but contains a deadlock. This deadlock is caused by a circular dependency without initial tokens. Depending on the exact type of blocks, this situation may arise in a block diagram with such a circular dependency (figure 6.5). This, once again, results in the following output:

```
Result: model is not deadlock free
Stopping analysis because of problems with the model.
```

Figure 6.6 shows a model that is consistent and deadlock free because there are sufficient initial tokens on the edge from v_1 to v_0 to allow execution. The result is shown below. Note that the slowest internal loop reflects the inverse of the throughput.

```
Result: model is valid
Slowest internal loop is 0.5 ms (tasks: ['v0', 'v1'])
```

Figure 6.7 shows another model that is consistent and deadlock free. This results in the following output:

```
Result: model is valid
Slowest internal loop is 1.0 ms (tasks: ['v0'])
```

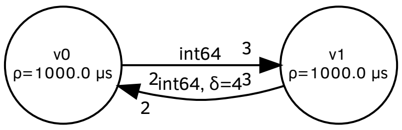


Figure 6.6: Example of a consistent model and deadlock-free model, adapted from M. Bekooij et al. (2005).

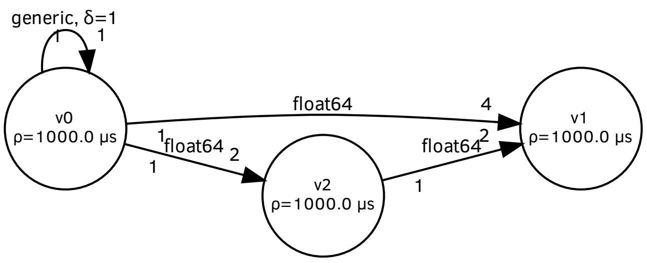


Figure 6.7: Example of a consistent model and deadlock-free model, adapted from M. Bekooij et al. (2005).

6.2 Quantization and delay

An ADC is a more complex component to model: it should produce tokens at a certain sampling rate (f_s). But a realistic ADC also produces the measured values with a certain sampling delay (τ_d).

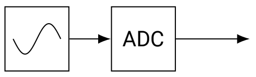


Figure 6.8: Block diagram for the system simulated in figure 6.9.

Figure 6.9 shows the result of simulations of an incoming 1 Hz sine signal and the digital value produced by both an ideal ADC with zero delay, and a slow ADC that requires 10 ms to process a sample. The corresponding block diagram is shown in figure 6.8. While the quantization is apparent in both ADC signals, only the slow ADC introduces a delay as well. This example shows that an incoming signal is distorted in the same way that one would expect to happen in reality.

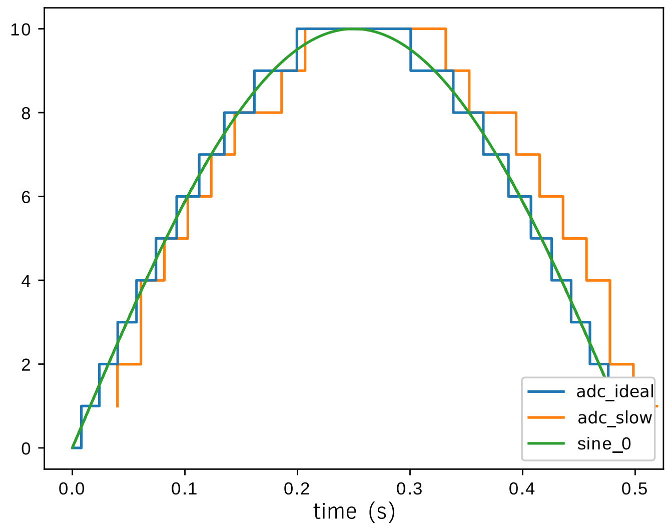


Figure 6.9: Simulation showing undelayed sampling (blue) and delayed sampling (orange) of an incoming signal (green).

This is not trivial to reproduce in SDF. Simply converting the ADC to a single HSDF task with a self-edge and a sine input (figure 6.10) leads to an inaccurate model: if the task has a firing duration of a single sampling period (T) all tokens produced by the ADC task will have delay of a full sampling period. There are three ways of resolving this:

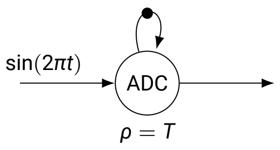


Figure 6.10: Example of an incorrectly modelled ADC task with a sine wave input.

1. Model the setup as an input that combines the sine wave and ADC (figure 6.11). In this case the produced tokens should contain a value that takes into account the sampling delay (τ_d): a token produced at t contains the quantized value of $\sin(2\pi(t - \tau_d))$. Although this is the correct model of the system, it requires that the ADC task has knowledge about the real-world, and that the real-world is not modelled in dataflow.

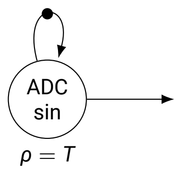


Figure 6.11: Model of a sine wave input/ADC combination that models a sine wave input and an ADC in a single task.

2. Take the behaviour of the ADC into account when modelling the plant, and manipulate the tokens so the timestamps stay realistic (figure 6.12). This can only be valid when the plant has no influence on the dataflow representation of the system, and requires knowledge of the ADC in the plant.

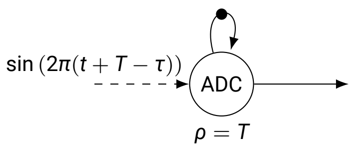


Figure 6.12: Single task model where the values produced by the plant compensate for the ADC delay.

3. Have the plant output lead by a single sampling period, and model the ADC as two tasks (figure 6.13). This is similar to the third solution, but requires no knowledge of the ADC in the plant.

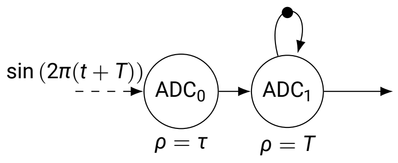


Figure 6.13: Single task model where the values produced by the plant compensate for the ADC delay.

The second solution is the one currently implemented when performing simulations against the *plant* task.

6.3 PID controller

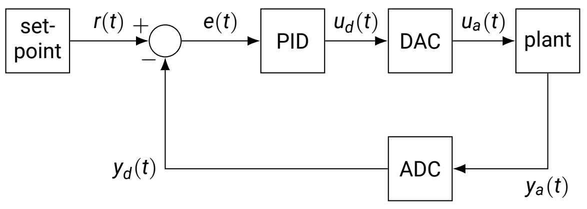


Figure 6.14: Example block diagram for a PID controller.

Figure 6.14 shows an example of a PID controller. As per the workflow in section 4.1, all elements of the block diagram have been made explicit: the block diagram contains a digital to analog converter (DAC), a analog to digital converter (ADC) and an explicit setpoint source.

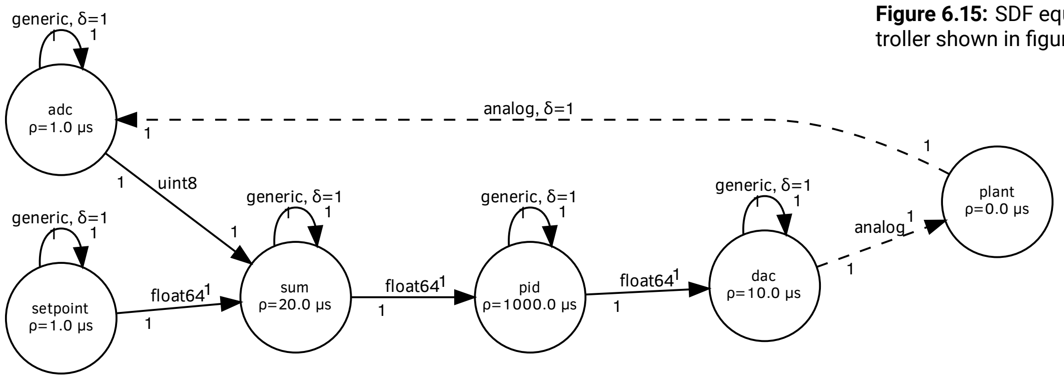


Figure 6.15: SDF equivalent of the controller shown in figure 6.14.

Creating a configuration for this model (see appendix A.3) results in the SDF diagram in figure 6.15. The configuration for this model is straightforward, with the following observations:

- For this example, none of the components have been set to be executed in parallel in order to simplify manual analysis of the model—all tasks are executed in sequence.
- The setpoint has a sample rate set identical to the ADCs sample rate in order to ensure that the time of production of the setpoint matches the time of production of the samples.
- In the model shown, the ADC is set to an extremely high sampling frequency (1 MHz) in order to observe what the tool has to say about the rest of the system. With such a high sampling frequency, the ADC will not be the limiting factor of the system, which it should be.
- Without changing the sampling frequency, the *PID* element is the slowest element with an execution time of 1 ms. This means that the minimum sampling frequency for a working system is 1 kHz, which is used in the simulation.

Figure 6.16 shows a plot of the setpoint and ADC signals in 'ideal' conditions: the PID-controller has a firing duration of 0 seconds (i.e. the calculations take *zero* time) in order to minimize the input/output delay, and ADC and DAC can deliver perfect analog signals. While the overshoot of the controller is large, the plant reaches the setpoint value in approximately 100 ms. Note that the delay of components other than the PID and ADC tasks is still present. This adds up to an input/output delay of 30 μs (caused by the *sum* and *dac* tasks), or 0.03 % of the sampling interval.

Figure 6.17 shows a plot of the same controller operating when the ADC

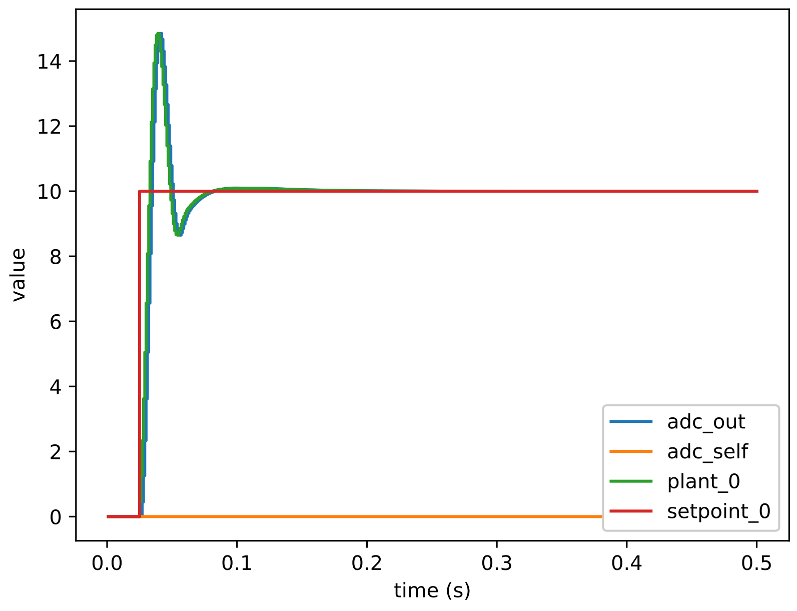


Figure 6.16: Plot of the performance of the PID controller in ideal conditions.

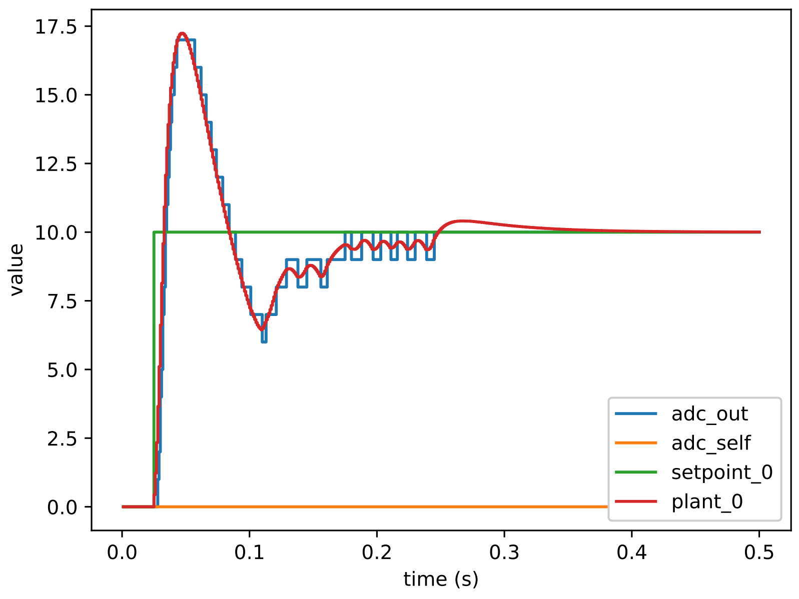


Figure 6.17: Plot of the performance of the PID controller with discrete ADC/-DAC values.

and DAC output discrete values. The performance of the controller is affected, now reaching the setpoint after approximately 350 ms.

Figure 6.18 shows a plot of the same controller, but now with the firing duration of the PID controller restored to its original value. The additional delay between the input and output has caused the controller to no longer be stable. The total input/output delay is now 1.03 ms, or 103 % of the sampling interval.

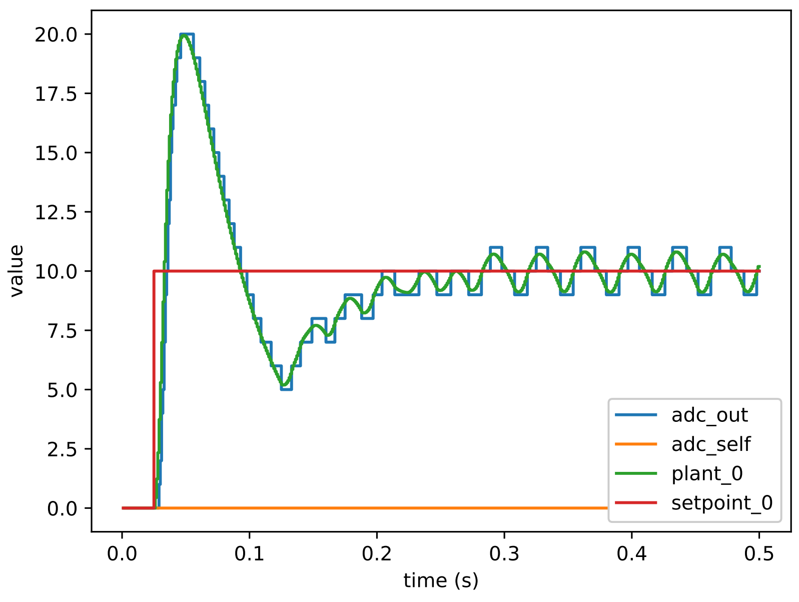


Figure 6.18: Plot of the performance of the PID controller with discrete ADC/-DAC values and a realistic firing duration for the PID task.

6.4 Embedded systems laboratory

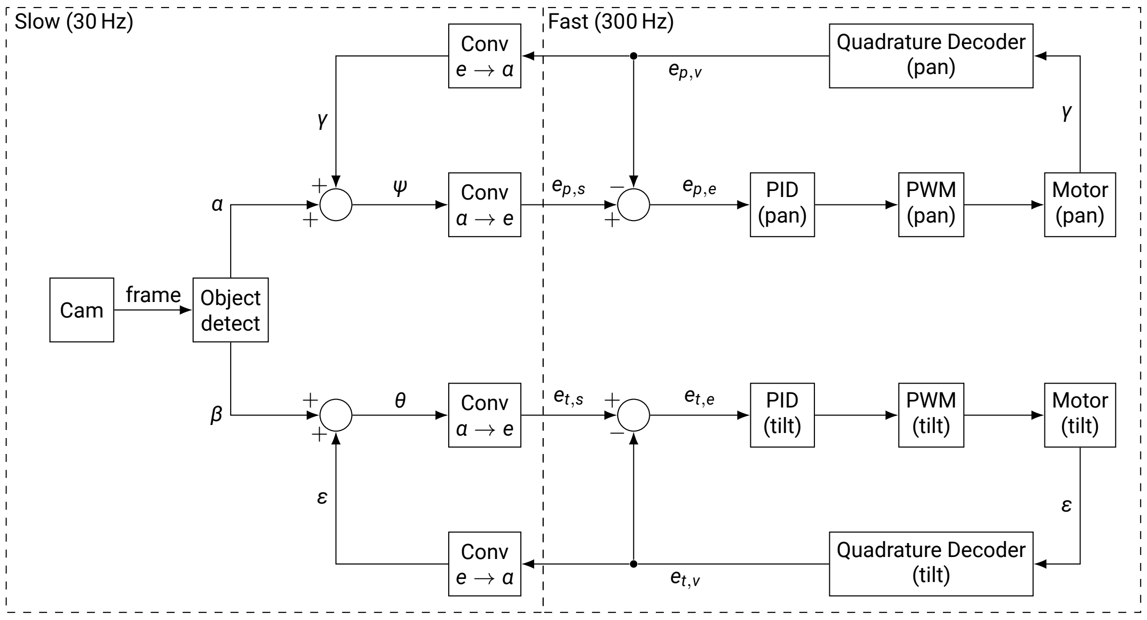


Figure 6.19: Block diagram of a controller developed for the *embedded systems laboratory* course.

During the *embedded systems laboratory* course at the University of Twente, students are given an assignment to develop a control system with *vision in the loop* using a webcam that can be controlled in two axes (pan and tilt). The available inputs are a pan and tilt angle, and images from the webcam. The development is done on a system containing both an ARM board (connected to the webcam) and an FPGA (connected to the pan and tilt motors). An often developed solution is detecting and tracking an object using the webcam.

Figure 6.19 shows a block diagram of a control system developed for the course. The controller contains a slow and a fast section:

- The *slow* part is limited by the frame rate of the webcam. In this section the *setpoints* for the controllers in the fast section are calculated from the detected position of the object and the pan/tilt angle: when the object is detected at 30° in the tilt axis and the current tilt angle is 10°, the setpoint will be set to 40°.
- The *fast* section is limited by the speed of the quadrature encoder, the motor and the connected physical system. This can be ten to a thousand times faster than the slow section. In the fast section the controller moves the webcam to the position set by the slow section.

In order to perform analysis on this control system, the following information is required:

- The frame rate of the webcam is 30 Hz.
- The output of the quadrature decoder is limited to 300 Hz.
- The transition between the fast and slow parts will require resampling.
- The *take-off point* on $e_{t,v}$ is converted to a multiplexer (*mux*) element.
- The *motor* is removed because there is no model available to simulate it. This element, containing the main part of the *plant* of this model, can be removed because this is not part of the software

or platform part of the system: the element is only required for simulation, not for analysis.

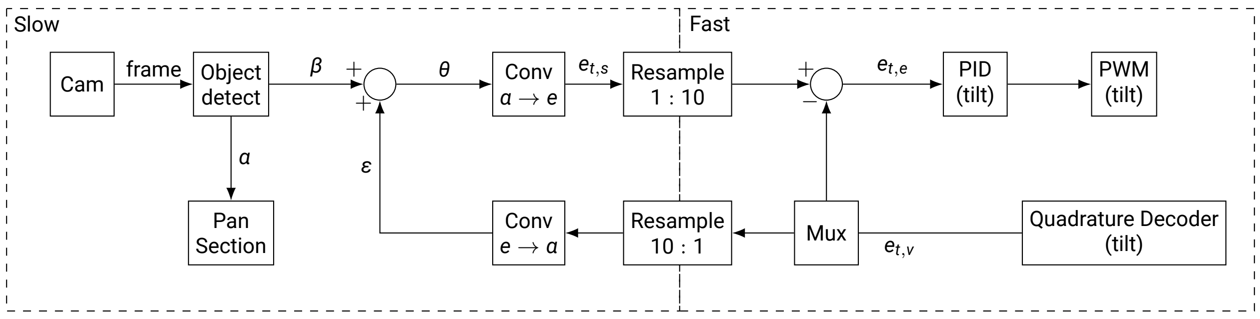


Figure 6.20: Block diagram of the tilt section of the control system in figure 6.19 with explicit resampling.

Making these elements explicit, and limiting the analysis to the *tilt* portion of the controller results in the block diagram shown in figure 6.20. This controller can be completely analysed by the tool (configuration in appendix A.4), noting the following:

- The slow section is analysed as if deployed on the ARM board.
- The fast section is analysed as if deployed on the FPGA. The FPGA is set to disallow executing tasks in parallel in order to model each task being implemented on the FPGA once.
- The *pan* section is represented by a sink, in order to have a valid model: the object detector always has a pan and a tilt output.
- The communication between the ARM board and the FPGA has not been modelled.

The resulting SDF model is shown in figure 6.21. Checking the model provides the following result:

```
Result: model is valid
Slowest internal loop is 33.3333 ms (tasks: ['cam'])
```

Because the webcam is the limiting component of the system, the model *should* be realizable on the target platforms. In order to confirm this, a schedule needs to be created that reflects the limitations of the target platform (e.g. 1 CPU core), and the communication between the ARM core and the FPGA needs to be included in the model. This requires knowledge of *how* the system is going to be realized in addition to the information currently available.

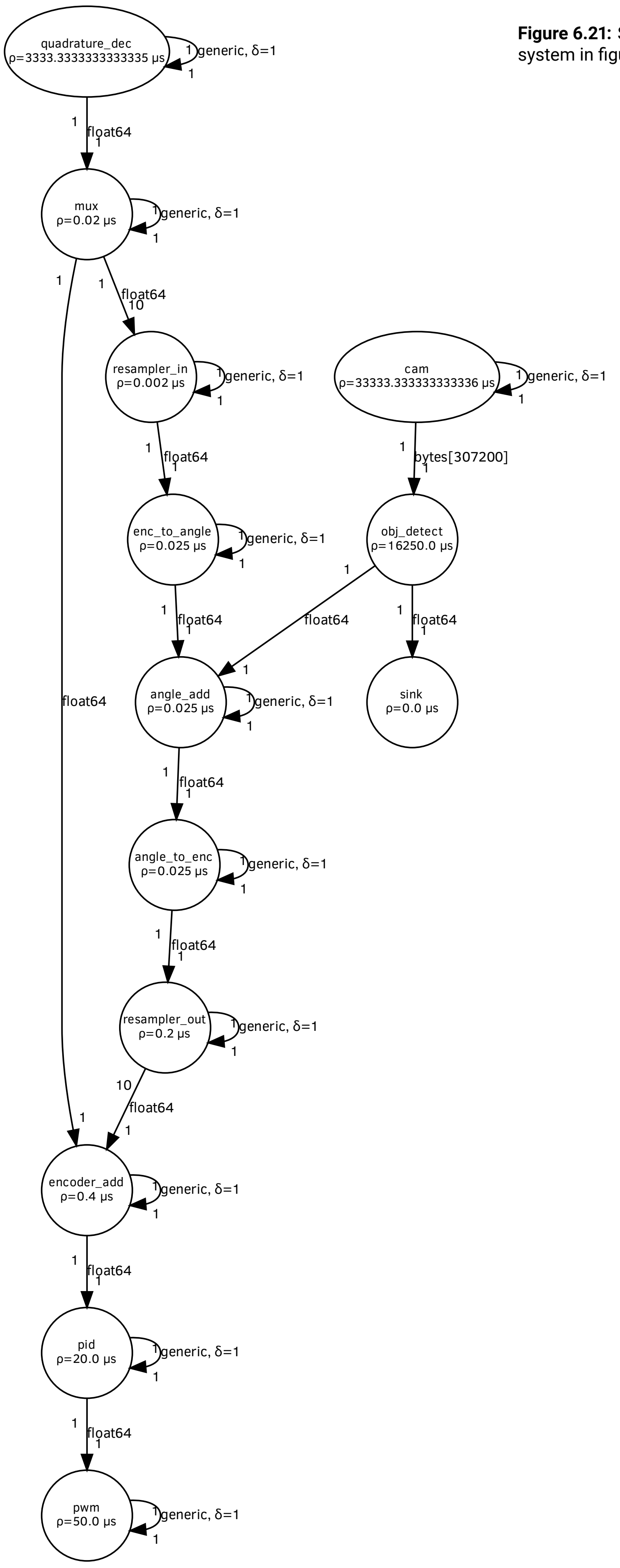


Figure 6.21: SDF diagram of the control system in figure 6.20.

Chapter 7

Conclusion & recommendations

The goal of this thesis is to make improvements in the way that control systems are built by improving the workflow for both the control engineer and software engineer. This is done by giving autonomy to the control engineer, and moving the software engineer in to the role of developer of a toolchain. The research shown in this thesis makes the first steps in making this feasible.

When comparing existing software solutions, all provide means to realize a platform-independent model on a platform. No software solution is suitable for the proposed workflow, however, as none provides analysis and simulation of the designed system as if it is realized on hardware.

For analysis of a platform-specific model of a system, synchronous dataflow (SDF) is chosen: having timing and the execution duration at its core allows for a real-time, though pessimistic, model of the system. Both block diagrams and communicating sequential processes (CSP) are not able to represent this information.

Using a model-to-model conversion from block diagrams to SDF allows for a platform-independent model, designed by a control engineer, to be converted into a platform-specific model in the form of SDF. This method allows the best of both worlds: the control engineer can keep working in block diagrams, but the dataflow equivalent of the system is used for analysis and more realistic simulation.

The constraints of the platform are represented in this platform-specific (dataflow) model in the form of an estimation of the time that it takes for tasks to be executed on that platform.

In order to transform a platform-independent model, in the form of a block diagram, to a platform-specific model, in the form of an SDF model, only two bits of information are required: (i) the type of the block in the block diagram, in order to select its implemented equivalent, and (ii) the clock speed of the specified platform.

On the basis of the analysis of the platform-specific model in SDF, some meaningful feedback can be given: the design can be verified to be deadlock-free, and the worst-performing loops in the system can be identified. By calculating the schedule it is possible to also provide information about the input/output delay.

This platform-specific model can then be simulated using the timing information inherent in the system to calculate the timing information of the inputs and outputs. Together with simulating the behaviour of components, this gives an accurate simulation of the expected real-time behaviour, though validation against the real-world is still required. Also missing is co-simulation, that would allow simulation against external software in order to simulate the behaviour of more complex models. The simulation is also limited in the representation of some tasks, like ADCs that have more complex SDF representations.

With this research it is possible to automatically create a platform-specific model from a platform-independent model and platform information by using dataflow, specifically SDF. This allows automated analysis and simulation of the expected real-time behaviour of the modelled system.

7.1 Recommendations

To improve the usefulness of the tool, it is recommended that the following functionality is added:

- Co-simulation with external simulation programs.
- Integration with the SDF3 toolchain (Stuijk, Geilen and Basten, 2006), to add additional analysis to the toolchain without adding significantly more software complexity.
- Add default platform models matching target platforms.
- Extend the level of integration of tasks and the platform. For example:
  - The automatic addition of tasks that model the communication channels between platforms, improving the level of accuracy to the real-world.
  - Improving the accuracy of the worst-case execution time estimates which may result in more realistic and less pessimistic estimates.
- Extend the analysis with schedule creation and perform feasibility analysis for running the system on *n* processor cores using existing techniques.
- Improve the representation of elements that produce values in a regular interval, but also incorporate a delay when producing these values, like ADCs and webcams.

The next steps for continuation of the project should lie in the work towards actual synthesis of the dataflow model in embedded software and hardware designs. This also has the effect of improving the modelling: creating runnable code allows analysis into the performance of that code, and the overhead of the communication channels.

In order to improve the workflow for control engineers using this tooling in the development of a control system, it is important that the ‘ideal’ situation can be reflected. To improve and ease the adoption by control engineers, some tooling has to be developed that allows conversion between models in external software (e.g. 20-sim), and converts this to the configuration format as currently specified.

Appendix A

Configuration

The following sections contain the configuration for the various examples and demos shown in the previous chapters.

A.1 Test cases

The following are the configuration files for the test cases shown in section 6.1.

```yaml
# This model is consistent and has a deadlock
name: consistent_deadlock

model:
  v0:
    type: fifo
    params:
      wcet: 1 ms
      parallel: true
      produce: 2
      consume: 2
    outputs:
      out: v1.in

  v1:
    type: fifo
    params:
      wcet: 1 ms
      parallel: true
      produce: 3
      consume: 3
    outputs:
      out: v0.in
    initial_tokens:
      out:
        - type: int8
          value: 0
        - type: int8
          value: 0
        - type: int8
          value: 0
```

test_models/consistent_deadlock_1.yaml

```yaml
# This model is consistent and deadlocks
name: consistent_deadlock

model:
  v0:
    type: resampler
    params:
      wcet: 1 ms
      parallel: true
      consume: 1
      produce: 1
    outputs:
      out: v1.in

  v1:
    type: resampler
    params:
      wcet: 1 ms
      parallel: true
      consume: 4
      produce: 2
    outputs:
      out: v2.in

  v2:
    type: resampler
    params:
      wcet: 1 ms
      parallel: true
      consume: 1
      produce: 2
    outputs:
      out: v0.in
```

test_models/consistent_deadlock_2.yaml

```
# This model is consistent and deadlock free
name: consistent_deadlockfree

model:
  v0:
    type: fifo
    params:
      wcet: 1 ms
      parallel: true
      produce: 2
      consume: 2
    outputs:
      out: v1.in

  v1:
    type: fifo
    params:
      wcet: 1 ms
      parallel: true
      produce: 3
      consume: 3
    outputs:
      out: v0.in
    initial_tokens:
      out:
        - type: int8
          value: 0
        - type: int8
          value: 0
        - type: int8
          value: 0
        - type: int8
```

```
          value: 0
```

test_models/consistent_deadlockfree_1.yaml

```
# This model is consistent and deadlock free
name: consistent_deadlockfree

model:
  v0:
    type: source
    params:
      outputs: 2
      wcet: 1 ms
      parallel: false
      produce: 1
      consume: 1
      samples: 1
    outputs:
      0: v1.0
      1: v2.in

  v1:
    type: sink
    params:
      wcet: 1 ms
      parallel: true
      consume: [4,2]

  v2:
    type: resampler
    params:
      wcet: 1 ms
      parallel: true
      ratio: 0.5 # Determines consume/production ratio
      consume: 2 # Overridden and determined by ratio
      produce: 1 # Overridden and determined by ratio
    outputs:
      out: v1.1
```

test_models/consistent_deadlockfree_2.yaml

```
# This model is inconsistent
name: inconsistent

model:
  v0:
    type: source
    params:
      outputs: 2
      wcet: 1 ms
      parallel: false
      produce: [1,1,1]
      consume: 1
      samples: 1
    outputs:
      0: v1.0
      1: v2.in

  v1:
    type: sink
    params:
      wcet: 1 ms
      parallel: true
      consume: [1,1]
  v2:
    type: resampler
    params:
      wcet: 1 ms
      parallel: true
      ratio: 2
      consume: 1
      produce: 2
    outputs:
      out: v1.1
```

test_models/inconsistent_1.yaml

A.2 Sine wave generator

The following is the configuration of the sine wave generator in section 6.2

```
name: Sine Model

model:
  sine:
    type: sine
    params:
      amplitude: 10
      frequency: 1.0
      phase: 0.0
      log: plot
      wcet: 0.1 ms
    outputs:
      0: resampler.in

  resampler:
    type: resampler
    params:
      ratio: 0.01
      parallel: true
    outputs:
      out:
        adc.in

  adc:
    type: adc
    params:
      sample_rate: 1 MHz
      bits: 8
      log: plot
      wcet: 10 ms
    outputs:
      out: sink.0

  sink:
    type: sink
    params:
      wcet: 0 ms
      parallel: true
```

examples/model_sine.yaml

A.3 PID controller

The following is the configuration of the PID controller example in section 6.3

```
name: pid

platforms:
  arm:
    default: true
    clock_speed: 1 MHz

model:
  setpoint:
    type: source
    params:
      type: float64
      sample_rate: 1 kHz # should be the same as the ADC
      outputs: 1
      samples: [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,10]
      log: plot
    outputs:
      0: sum.0

  adc:
    type: adc
    params:
      sample_rate: 1 kHz # set to 1 MHz to check rest of elements
      bits: 8
      log: plot
    outputs:
      out: sum.1

  sum:
    type: mac
    params:
      type: float64
      inputs: 2
      factors: [1, -1]
    outputs:
      0: pid.in

  pid:
    type: pid
    params:
      # wcet: 0 ms
      type: float64
      kp: 2
      ti: 50
      td: 0.01
    outputs:
      out: dac.in

  dac:
    type: dac
    params:
      bits: 8
    outputs:
      out: plant.0

  plant:
    type: plant
    params:
      initial_value: 0
      log: plot
      inputs: 1
      outputs: 1
    outputs:
      0: adc.in
```

examples/model_pid.yaml

A.4 ESL model

The following is the configuration of the PID controller example in section 6.4

```
name: test

platforms:
  arm:
    default: true
    clock_speed: 800 MHz
  fpga:
    clock_speed: 50 MHz
    allow_parallel: false

model:
  cam:
    type: webcam
    params:
      framerate: 30
      width: 640
      height: 480
    outputs:
      0: obj_detect.in

  obj_detect:
    type: object_angle_detector
    params:
      width: 640
      weight: 480
      fov: 90.0
    outputs:
      x: angle_add.0
      y: sink.0

  sink:
    params:
      inputs: 1
    type: sink

  angle_add:
    type: mac
    params:
      inputs: 2
      factors: [1, 1]
    outputs:
      0: angle_to_enc.0

  angle_to_enc:
    type: mac
    params:
      inputs: 1
      factors: [1000]
    outputs:
      0: resampler_out.in

  enc_to_angle:
    type: mac
    params:
      inputs: 1
      factors: [0.001]
    outputs:
      0: angle_add.1

  resampler_out:
    type: resampler
    platform: fpga
    params:
      ratio: 10
    outputs:
      out: encoder_add.0

  resampler_in:
    type: resampler
    platform: fpga
    params:
      ratio: 0.1
    outputs:
      out: enc_to_angle.0

  encoder_add:
    type: mac
    platform: fpga
    params:
      inputs: 2
      factors: [1, -1]
    outputs:
      0: pid.in

  pid:
    type: pid
    platform: fpga
    outputs:
      out: pwm.in

  pwm:
    type: pwm
    platform: fpga
    params:
      frequency: 20 kHz

  mux:
    type: mux
    platform: fpga
    params:
      outputs: 2
    outputs:
      0: resampler_in.in
      1: encoder_add.1

  quadrature_dec:
    type: adc
    platform: fpga
    params:
      sample_rate: 300 Hz
    outputs:
      out: mux.0
```

examples/model_esl.yaml

Bibliography

Bekooij, Marco et al. (2005). 'Dataflow analysis for real-time embedded multiprocessor system design'. In: *Dynamic and robust streaming in and between connected consumer-electronic devices*. Springer, pp. 81–108.

Bezemer, M.M. (Nov. 2013). 'Cyber-physical systems software development: way of working and tool suite'. Undefined. PhD thesis. University of Twente. ISBN: 978-90-365-1879-6. DOI: 10.3990/1.9789036518796.

Bezemer, M.M. and R.J.W. Wilterdink (June 2011). 'LUNA: Hard Real-Time, Multi-Threaded, CSP-Capable Execution Framework'. In: *Communicating Process Architectures 2011*. Ed. by Peter H. Welch et al. Concurrent System Engineering Series WoTUG-33. IOS Press, pp. 157–175. ISBN: 978-1-60750-773-4. DOI: 10.3233/978-1-60750-774-1-157.

Broenink, Johannes F. (1997). 'Modelling, Simulation and Analysis with 20-Sim'. Undefined. In: *Journal "A"* 38.3, pp. 22–25. ISSN: 0771-1107.

Broenink, Johannes F., Yunyun Ni and M.A. Groothuis (Nov. 2010). 'On model-driven design of robot software using co-simulation'. Undefined. In: *Proceedings of SIMPAR 2010 Workshops International Conference on Simulation, Modeling, and Programming for Autonomous Robots*. Ed. by E. Menegatti. TU Darmstadt, pp. 659–668. ISBN: 978-3-00-032863-3.

Bruyninckx, Herman et al. (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.

clash-lang.org (13th Nov. 2018). *CλaSH - From Haskell to Hardware*. URL: http://www.clash-lang.org/.

Control Systems/Block Diagrams (13th Nov. 2018). URL: hhttps://en.wikibooks.org/w/index.php?title=Control_Systems/Block_Diagrams&oldid=3322668.

Controllab (10th Nov. 2018). *20-sim*. URL: http://www.20sim.com/.

Franklin, Gene F, J David Powell and Abbas Emami-Naeini (2006). *Feedback control of dynamic systems*. 5th ed. Pearson Education, Inc.

Hailpern, Brent and Peri Tarr (2006). 'Model-driven development: The good, the bad, and the ugly'. In: *IBM systems journal* 45.3, pp. 451–461.

Hanik, Filip (3rd Mar. 2006). *The Kiss Principle*. URL: https://people.apache.org/~fhanik/kiss.html.

Hoare, Charles Antony Richard (1978). 'Communicating sequential processes'. In: *Communications of the ACM* 21.8, pp. 666–677.

IEEE (June 2011). 'IEEE Draft Guide: Adoption of the Project Management Institute (PMI) Standard: A Guide to the Project Management Body of Knowledge (PMBOK Guide)-2008 (4th edition)'. In: *IEEE P1490/D1, May 2011*, pp. 1–505. DOI: 10.1109/IEEESTD.2011.5937011.

Jovanovic, D.S. et al. (Sept. 2004). 'gCSP: A Graphical Tool for Designing CSP systems'. Undefined. In: *Proceedings Communicating Process Architectures 2004*. Concurrent Systems Engineering Series 62. IOS Press, pp. 233–251. ISBN: 9781586034580.

Kleijn, C. (2013). *20-sim 4C 2.1 Reference Manual*. Controllab Products B.V.

Kleijn, C., M.A. Groothuis and Differ H.G (2017). *20-sim 4.6 Reference manual*. Controllab Products B.V.

Kourie, Derrick G et al. (2018). 'Using CSP to Develop Quality Concurrent Software'. In: *Principled Software Development*. Springer, pp. 165–184.

Kuipers, FP (2017). 'FPGA design support using CλaSH and LUNA'. MA thesis. University of Twente.

Kumar, Akash et al. (2007). 'A probabilistic approach to model resource contention for performance estimation of multi-featured media devices'. In: *Design Automation Conference, 2007. DAC'07. 44th ACM/IEEE*. IEEE, pp. 726–731.

Kurtin, Philip S, Joost PHM Hausmans and Marco JG Bekooij (2016). 'HAPI: An event-driven simulator for real-time multiprocessor systems'. In: *Proceedings of the 19th International Workshop on Software and Compilers for Embedded Systems*. ACM, pp. 60–66.

Lee, E. A. and D. G. Messerschmitt (Sept. 1987). ‘Synchronous data flow’. In: *Proceedings of the IEEE* 75.9, pp. 1235–1245. ISSN: 0018-9219. DOI: 10.1109/PROC.1987.13876.

Lee, E. A. and T. M. Parks (May 1995). ‘Dataflow process networks’. In: *Proceedings of the IEEE* 83.5, pp. 773–801. ISSN: 0018-9219. DOI: 10.1109/5.381846.

Lu, Zhou, Tjalling Ran and Johannes F. Broenink (Aug. 2016). ‘Simulation and visualization tool design for robot software’. Undefined. In: *Communicating Process Architectures 2016: proceedings of the 38th WoTUG Technical Meeting*. Ed. by K. Chalmers and J.B. Pedersen. Open Channel Publishing Ltd, pp. 63–82. ISBN: 978-0-9934385-1-6.

MathWorks (2017). *MATLAB/Simulink User's Guide*.

– (10th Nov. 2018). *Simulink Hardware Support Package System Requirements*. URL: https://nl.mathworks.com/hardware-support/system-requirements.html.

Mathworks (10th Nov. 2018). *Mathworks Simulink*. URL: https://www.mathworks.com/products/simulink.html.

Oguz, Og˘uzcan, Johannes F. Broenink and Angelika H. Mader (Aug. 2012). ‘Schedulability analysis of timed CSP models using the PAT model checker’. Undefined. In: *Communicating Process Architectures 2012*. Ed. by P.H. Welch et al. Open Channel Publishing Ltd, pp. 65–88. ISBN: 978-0-9565409-5-9.

Schmidt, Douglas C (2006). ‘Model-driven engineering’. In: *COMPUTER-IEEE COMPUTER SOCIETY-* 39.2, p. 25.

Stuijk, Sander, Marc Geilen and Twan Basten (2006). ‘Sdf^3: Sdf for free’. In: *Application of Concurrency to System Design, 2006. ACSD 2006. Sixth International Conference on*. IEEE, pp. 276–278.

Stuijk, Sander, Marc Geilen, Bart Theelen et al. (2011). ‘Scenario-aware dataflow: Modeling, analysis and implementation of dynamic applications’. In: *Embedded Computer Systems (SAMOS), 2011 International Conference on*. IEEE, pp. 404–411.

Woodside, C Murray (1989). ‘Throughput calculation for basic stochastic rendezvous networks’. In: *Performance Evaluation* 9.2, pp. 143–160.