UNIVERSIDADE ESTADUAL DE CAMPINAS

FACULDADE DE ENGENHARIA MECÂNICA

Ricardo de Souza Bonna

A Runtime Reconfigurable Embedded Systems Design Methodology based on Formal Models of Computation

Uma Metodologia para Projeto de Sistemas Embarcados com Reconfiguração em Tempo de Execução baseada em Modelos Formais de Computação

CAMPINAS

2021

Ricardo de Souza Bonna

A Runtime Reconfigurable Embedded Systems Design Methodology based on Formal Models of Computation

Uma Metodologia para Projeto de Sistemas Embarcados com Reconfiguração em Tempo de Execução baseada em Modelos Formais de Computação

Thesis presented to the School of Mechanical Engineering of the University of Campinas in partial fulfillment of the requirements for the degree of Doctor in Mechanical Engineering, in the area of Mechatronics.

Tese apresentada à Faculdade de Engenharia Mecânica da Universidade Estadual de Campinas como parte dos requisitos exigidos para obtenção do título de Doutor em Engenharia Mecânica, na Área de Mecatrônica.

Orientador: Prof. Dr. Denis Silva Loubach

ESTE EXEMPLAR CORRESPONDE À VERSÃO FINAL DA TESE DEFENDIDA PELO ALUNO RICARDO DE SOUZA BONNA, E ORIENTADA PELO PROF. DR. DENIS SILVA LOUBACH.

CAMPINAS
2021

Ficha catalográfica
Universidade Estadual de Campinas
Biblioteca da Área de Engenharia e Arquitetura
Rose Meire da Silva - CRB 8/5974

B642r

Bonna, Ricardo de Souza, 1990-
A runtime reconfigurable embedded systems design methodology based on formal models of computation / Ricardo de Souza Bonna. – Campinas, SP : [s.n.], 2021.

Orientador: Denis Silva Loubach.
Tese (doutorado) – Universidade Estadual de Campinas, Faculdade de Engenharia Mecânica.

1. Sistemas embarcados (computadores). 2. Linguagem de modelagem. 3. Design. I. Loubach, Denis Silva, 1982-. II. Universidade Estadual de Campinas. Faculdade de Engenharia Mecânica. III. Título.

Informações para Biblioteca Digital

Título em outro idioma: Uma metodologia para projeto de sistemas embarcados com reconfiguração em tempo de execução baseada em modelos formais de computação
Palavras-chave em inglês:
Embedded systems (computers)
Modeling languages
Design
Área de concentração: Mecatrônica
Titulação: Doutor em Engenharia Mecânica
Banca examinadora:
Denis Silva Loubach [Orientador]
Eurípedes Guilherme de Oliveira Nóbrega
Rodolfo Jardim de Azevedo
Juliana de Melo Bezerra
Osamu Saotome
Data de defesa: 07-07-2021
Programa de Pós-Graduação: Engenharia Mecânica

Identificação e informações acadêmicas do(a) aluno(a)
- ORCID do autor: https://orcid.org/0000-0002-9212-4304
- Currículo Lattes do autor: http://lattes.cnpq.br/3312289892869275

UNIVERSIDADE ESTADUAL DE CAMPINAS
FACULDADE DE ENGENHARIA MECÂNICA

TESE DE DOUTORADO

A Runtime Reconfigurable Embedded Systems Design Methodology based on Formal Models of Computation

Uma Metodologia para Projeto de Sistemas Embarcados com Reconfiguração em Tempo de Execução baseada em Modelos Formais de Computação

Autor: Ricardo de Souza Bonna
Orientador: Prof. Dr. Denis Silva Loubach

A banca examinadora composta pelos membros abaixo aprovou esta tese:

Prof. Dr. Denis Silva Loubach
Instituto Tecnológico de Aeronáutica

Prof. Dr. Eurípedes Guilherme de Oliveira Nóbrega
FEM/UNICAMP

Prof. Dr. Rodolfo Jardim de Azevedo
IC/UNICAMP

Profa. Dra. Juliana de Melo Bezerra
Instituto Tecnológico de Aeronáutica

Prof. Dr. Osamu Saotome
Instituto Tecnológico de Aeronáutica

A Ata da defesa com as respectivas assinaturas dos membros encontra-se no SIGA/Sistema de Fluxo de Dissertação/Tese e na Secretaria do Programa da Unidade.

Campinas, 07 de Julho de 2021

To my parents Kathia and Marcus, who invested heavily on my education, and to my girlfriend Elisa, who has been by my side since the beginning.

Acknowledgements

First of all, I would like to thank professor Denis Loubach for his support in all aspects of this PhD project.

I thank professor Ingo Sander, from KTH, for accepting me as a foreign researcher and sharing his expertise in the area of formal systems design during my sandwich doctorate at Sweden.

I thank my colleagues from ACCES Laboratory at Unicamp and from Embraer for the shared daily experiences.

I thank my family and friends for all the support throughout my life.

This PhD research work was partially supported by the Call of Projects CISB Saab 04/2016.

This study was financed in part by the São Paulo Research Foundation (FAPESP), grants #2014/24855-8 and #2019/27327-6.

Resumo

Sistemas embarcados estão presentes em toda a parte hoje em dia. Muitos sistemas dependem de algum tipo de dispositivo eletrônico programável para operar corretamente. A execução correta e esperada desses sistemas depende da implementação correta de muitos recursos necessários diferentes, cada um dependendo de vários detalhes de implementação e propriedades que devem ser tratadas em tempo de projeto. Contudo, lidar com todos os detalhes de implementação nas fases iniciais do projeto não é produtivo, e muitas vezes até impossível. Para sistemas com reconfiguração em tempo de execução, um tipo de sistema embarcado que pode se reconfigurar em tempo de execução, o projeto é ainda mais desafiador devido ao processo de reconfiguração, que é outro recurso a ser levado em consideração.

Até hoje, o projeto de sistemas com reconfiguração em tempo de execução é, na maioria dos casos, feito por meio de procedimentos ad hoc, sem uma metodologia formal. Começar o projeto em um alto nível de abstração, com modelos de aplicação e plataforma, e reduzindo progressivamente a abstração, incorporando detalhes de implementação aos modelos, é uma estratégia bem conhecida para lidar com o alto nível de complexidade no projeto de sistemas embarcados. Essa estratégia também poderia ser aplicada a sistemas com reconfiguração em tempo de execução.

Como principal contribuição, esta tese fornece os primeiros passos para uma metodologia de projeto formal para sistemas com reconfiguração em tempo de execução por meio do desenvolvimento de um sistema de classificação, na forma de ontologias de domínio, para modelos de aplicação e plataforma, bem como um conjunto de regras de mapeamento que mescla modelos de aplicação e plataforma em um modelo virtual de implementação. Ambas as ontologias de aplicação e plataforma são gerais o suficiente para serem usadas como sistemas de classificação para diversos modelos de computação e plataformas comerciais de prateleira, respectivamente. Esta tese também apresenta a modelagem de aplicações com reconfiguração em tempo de execução usando os modelos de computação síncrono e *scenario-aware dataflow*.

Palavras-chave: Modelos de computação; reconfiguração em tempo de execução; metodologia de projeto; sistemas embarcados

Abstract

Embedded systems are ubiquitous in today's life. Many systems rely on some kind of programmable electronic device to operate properly. The expected and correct execution of the system relies on the correct implementation of many different needed features, each one depending on several implementation details and properties that must be dealt with in design time. However, dealing with all implementation details at the early stages of the design process is not productive, sometimes not even possible. For runtime reconfigurable systems, a type of embedded system that can reconfigure itself in runtime, the design process is even more challenging due to the reconfiguration process, which is yet another feature to be taken into consideration.

Up until today, in most cases, the design of runtime reconfigurable systems is done via ad hoc procedures, lacking a formal methodology. Starting the design in a high level of abstraction, with application and platform models, and progressively reducing the abstraction by incorporating implementation details to the models is a well-known strategy to cope with the high level of complexity in the design of embedded systems. Such a strategy could be also applied to runtime reconfigurable systems.

As the main contribution, this thesis provides the first steps towards a formal design methodology for runtime reconfigurable systems through the development of a classification system, in the form of domain ontologies, for both application and platform models, as well as a set of mapping rules that merges application and platform models into a virtual implementation model. Both application and platform domain ontologies are general enough to be used as classification systems for many different models of computation (MoCs) and commercial off-the-shelf platforms, respectively. This thesis also presents the modeling of runtime reconfigurable applications using the synchronous and scenario-aware dataflow MoCs.

Keywords: Models of computation; runtime reconfiguration; design methodology; embedded systems

List of Figures

List of Tables

Acronyms

ASIC	Application-Specific Integrated Circuits.
COTS	Commercial Off-The-Shelf.
CPS	Cyber-Physical Systems.
CSSM	Current State Shared Memory.
EDSL	Embedded Domain Specific Language.
FIFO	First-In First-Out.
FMS	Flight Management System.
ForSyDe	Formal System Design.
FPGA	Field-Programmable Gate Arrays.
FSM	Finite State Machine.
GPP	General-Purpose Processors.
GPU	Graphics Processing Unit.
HDL	Hardware Description Language.
MoC	Model of Computation.
PASS	Periodic Admissable Sequential Schedule.
ProgDev	Programmable Device.
ReconDev	Reconfigurable Device.
RISC	Reduced Instruction Set Computer.
RTR	Runtime Reconfiguration.
RTRP	Runtime Reconfigurable Processor.
SADF	Scenario-Aware Dataflow.
SDF	Synchronous Dataflow.
SoC	System on Chip.
SY	Synchronous.

Contents

1 INTRODUCTION

Embedded systems are present in all levels of our modern society. From small devices such as calculators and digital watches to larger ones such as cars, airplanes and power plants. All of these systems have some sort of embedded digital processing unit in which they rely on.

Although embedded systems exist since the '70s, in most of its history it has been viewed as small computers with limited processing, small memory and limited power consumption (Lee and Seshia, 2015). An *embedded* computational *system* consists in a set of processors embedded within a larger system that communicates with the physical environment (Radojevic and Salcic, 2011). Differently from general-purpose computers, embedded systems are normally dedicated to specific tasks and have a strong coupling between software and hardware. Concurrency and time-related problems are also characteristics of embedded systems. When an embedded system needs to respond to events within a well-defined time interval, this embedded system is said *real-time* (Furht et al., 1991). Signal control and processing systems fall into this category.

In the design of embedded systems it is often necessary to consider key constraints such as performance, power consumption and cost. One does not desire to connect a smart watch to a power supply at every couple of hours, or wait a couple of seconds for it to respond commands. The same can be said about most of the systems relying on embedded processors.

New approaches for embedded systems design become attractive, such as *reconfigurable systems*, by means of reconfigurable devices such as field-programmable gate arrays (FPGAs). One of the most important benefits of reconfigurable systems is narrowing the gap between general-purpose processors (GPPs) and application-specific integrated circuits (ASICs) (Chattopadhyay, 2013). GPPs are capable to perform many functionalities, but that flexibility comes at a cost in performance. ASICs have excellent performance but are dedicated to the task it was designed for. With reconfigurable systems the goal is to extract the best of both concepts: the flexibility of GPPs with the performance of ASICs, as shown in Figure 1.1.

Reconfigurable platforms contribute to the development of software simultaneously with hardware in embedded systems design. FPGAs are widely used in the industry as a prototype platform, in which various design concepts can be explored and tested in early stages of system development before a final architecture can be achieved. Moreover, with the usage of a reconfigurable hardware platform, embedded systems manufacturers can keep the system always up to date by providing software and hardware optimizations, bug fixes, and new added features throughout updates. This helps to reduce obsolescence, *i.e.* the property of a system to become outdated with time.

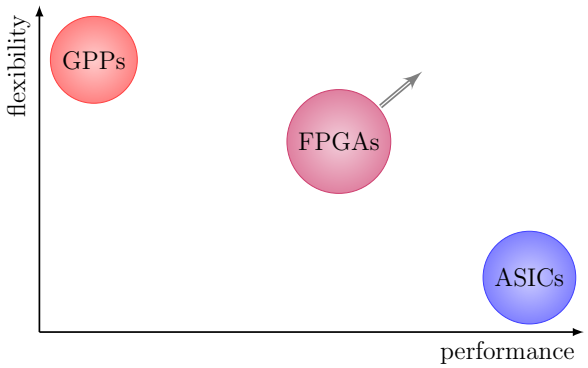


Figure 1.1. Performance *vs.* flexibility of GPPs and ASICs. Reconfigurable devices, such as FPGAs, could narrow the gab between these devices.

The aerospace industry provides good examples on the importance of reduced obsolescence. In a typical airliner, a computational device, such as the flight management system (FMS) has the same hardware architecture through the entire life span of the aircraft, which can easily be more than 30 years. Reconfigurable devices could be used to keep the airplane attractive to buyers looking for systems with modern features without the need to physically change the hardware platform.

Within reconfigurable systems lies a class of embedded systems called *runtime reconfigurable embedded systems*. They are computing devices capable of reconfiguring its own hardware in runtime without the need to halt the application running on it. To perform such feature, the reconfigurable device should be able to change parts of their hardware area without interfering with the remaining areas. Such a feature is called *partial reconfiguration*. FPGAs capable of performing partial reconfiguration have been around for more than a decade, but the concept of a computing device that can change its hardware configuration in runtime is considerably older.

In the '60s, Estrin published a work (Estrin, 1960) about a conceptual computer with a fixed and a variable part. Such computer could have its hardware (the variable part) reconfigured in runtime, and, at the time it was proposed, such property could provide an increase in performance by several orders of magnitude. However, with the fast evolution of silicon electronics design following Moore's law and Dennard scaling, the performance of general purpose processors increased exponentially, and the concept of a fixed plus variable computer became irrelevant for performance increase. In the beginning of the 21st century, silicon electronics development started to experience a slowdown in Moore's law and a breakdown of Dennard scaling, as the size of transistors and clock frequency speeds approached physical barriers. So, the strategy of increasing transistor density and clock frequencies to provide performance increase was not sufficient anymore, and different strategies needed to be explored. Estrin's fixed plus variable computer is one of such strategies that could be revisited in order to increase performance in embedded systems by exploring the flexibility vs performance advantage of reconfigurable systems shown in Figure 1.1.

Besides an increment in performance, runtime reconfigurable systems could provide an interesting solution for the design of safety-critical systems. An embedded system is considered safety-critical when it can directly or indirectly contribute to put lives at risk, damage the environment, or cause big economical losses (Bozzano and Villafiorita, 2011). According to Bieber et al. (2012), runtime reconfiguration is still a big challenge for future safety-critical systems, such as integrated modular avionic systems. In the event of a hardware failure, the system could be able to reallocate the functionalities from the faulty module into a safe module, thus limiting the effects and propagation of a hardware failure on aircrafts. It also increases usability, since the safe module could be used for less critical tasks before being reconfigured to assume more critical functionalities, thus avoiding the necessity of using spare modules that are only used in hardware fault.

The literature related to reconfigurable processors is filled with terminologies and keywords that often make it difficult to distinguish whether a processor is reconfigurable or not. Next, there are some definitions of key terms from (Chattopadhyay, 2013) and (Loubach, 2016), to establish a common understanding related to this matter and used along this thesis.

- *Programmable device*: A device controlled by a high level language which does not involve any hardware's or device structure's change. Here, the software has to adapt to the hardware. Usually, that is the microprocessors class. The property of being programmable is called programmability.

- *Reconfigurable device*: A device that can have its hardware changed, *i.e.* low-level switches, to perform a different task. Currently, this is performed by technologies such as modern FPGAs. In this case, the hardware is generally optimzed to run a specific functionality. The property of being reconfigurable is called reconfigurability.

- *Partial reconfiguration*: A reconfigurable processor or reconfigurable device supports partial reconfiguration if it allows its programmable area to be divided into one or more partitions, and each partition can be reconfigured, *i.e.* changed, independently from each other and without stopping the other ones.

- *Full reconfiguration*: A reconfigurable processor or reconfigurable device supports full reconfiguration when it considers the whole programmable area as just one reconfiguration block. In this case, the functionality can be changed at the cost of stopping the entire programmable area.

- *Runtime reconfiguration (RTR)*: Is the possibility to change the functionality, *i.e.* configuration, of a system in runtime without halting the execution. Although the term is widely associated with hardware reconfiguration using partial reconfiguration, this thesis extends the concept to consider RTR in software for programmable devices. The rationale behind this decision is that, from the application standpoint, it does not matter if the reconfiguration is done in hardware or software, as reconfiguration is represented as a change in the

application's functionality in the same way for both. This thesis considers two key ways to perform RTR, enumerated below:

1. *Mode reconfiguration:* The system, or part of it, is divided into modes with only one mode active at the same time. In each mode, the system performs different functionalities, and the mode switching is performed by switches and selects. A special signal is used to select the mode using switch and select processes. Can be implemented in both software and hardware.
2. *Function reconfiguration:* The different functionalities of processes in a function reconfigurable system are not implemented in the processes at the same time. Instead, they are stored in a memory as bitstreams and, when requested, a hardware reconfiguration is performed in a partition of the reconfigurable area using one of the functionalities stored in the memory. It can only be implemented in reconfigurable devices, preferably with support for partial reconfiguration.

The complexity of current embedded systems is considered high, especially when regarding RTR. In this sense, designing such systems is a considerable challenge. Due to the increasing popularity of reconfigurable systems and their wide range of applications, it is necessary to develop a formal methodology for RTR and real-time embedded systems design. In view of this, a formal methodology that enables the design and guarantees the properties and functionality desired for such systems is still lacking. Therefore, this PhD research work aims to provide the first steps towards a formal methodology for RTR embedded systems design.

1.1 Research Objective

The main goal of this doctoral research work is to **develop a methodology for runtime reconfigurable embedded systems design based on the theory of formal models of computation**, aiming the reconfiguration of embedded systems in runtime and minimizing the traditional testing needs.

1.2 Research Overview

According to Edwards et al. (1997), the design of embedded systems should begin at a high level of abstraction, since it is difficult to capture implementation-level functionality, namely lower abstraction level, at first. In this case, the process begins with an *application model*. Thus, the

designer develops a model, written in a formal language, according to the design requirements. The formal language has formal syntax and semantics, allowing the detection of inconsistencies or ambiguities already in the application modeling phase. The higher the abstraction level of the model, the less implementation details this model will have, leading to an *abstraction gap* (Sander, 2003) that separates a single application model from all feasible implementations of such model. Higher abstraction gaps leads to a higher number of feasible implementations, also called design space. The application model is then combined with a *platform model* to produce a correct and feasible *virtual implementation model*, a high-level model representing not only application and platform, but also where each element from the application is implemented into the platform.

The combination between application and platform models is performed by following a number of defined *mapping rules*, *i.e.* a set of directives and constraints that specify where each element of the application must be implemented in a given platform. Such mapping rules are defined on top of a *classification system*, presented as a pair of *domain ontologies*, one for the application and another one for the platform model. Thus, this can classify elements of each of these models based on their common properties and definitions. Both platform and application models must comply with the classification system so that the mapping rules can be applied to combine them into a feasible virtual implementation model.

To illustrate such design process proposal overview, Figure 1.2 shows a design flow for RTR systems represented as a Petri-Net[1]. The blue circles represent places which are pools of tokens representing elementary products, and the gray squares stand for transformations between tokens. The main idea is to have an output *virtual implementation model* as a result of formally defined *mapping rules* applied to an *application model* and a *platform model* complying with *application and platform ontologies*.

For a design flow proof of concept, the application model is defined using both the *synchronous* (SY) model of computation (MoC) and the *scenario-aware dataflow* (SADF) MoC. Selecting between one of these two MoCs depends on the kind of system being modeled. For real-time applications, the synchronous MoC is a good choice due to the *timed* characteristics of such MoC. SADF is better suited for streaming applications, although it can also be used for real-time applications, but it requires special concern when dealing with time due to the fact that it is an *untimed* MoC. It is important to point out that this design flow does not limit the application model to either the synchronous or SADF MoCs. Other models of computation can also be used in the proposed design flow, provided they comply with the application domain ontology.

[1]Petri-Net is a model of computation best suited for modeling processes in which products, represented as tokens, are consumed to generate other kinds of products. Places represent a pool of products of a given type, and transitions represents transformations of products. This model of computation is used in this thesis only for modeling the overview of the design flow, and no further knowledge about it is required.

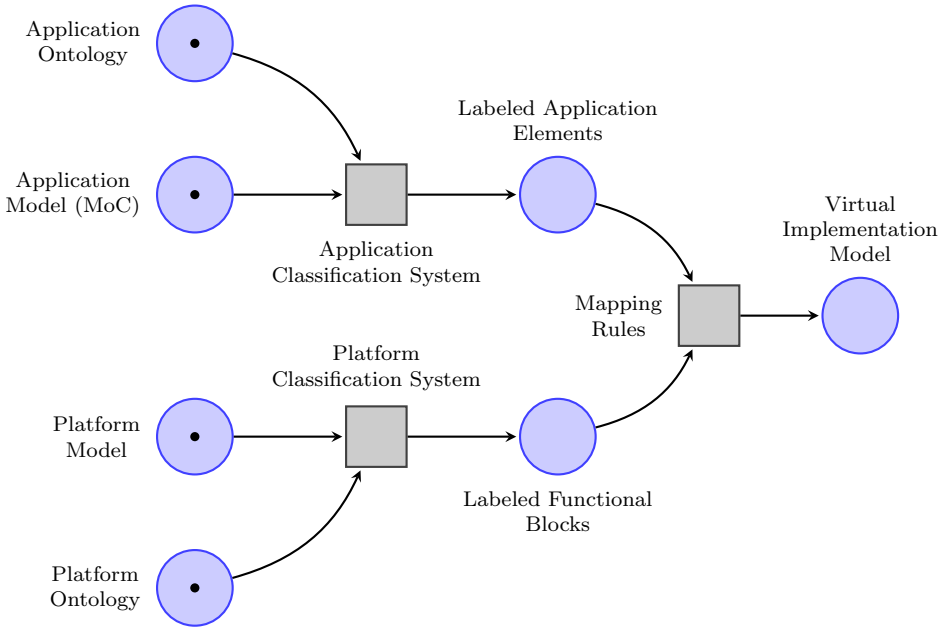


Figure 1.2. Petri-Net diagram representing the proposed design flow. Transitions are represented by gray squares, places are represented by blue circles and tokens are represented by bullets •.

1.2.1 Research Originality

This research work's originality is concentrated on:

1. the development of a *classification system* for both platform and application models;
2. the synchronous and scenario-aware dataflow *model of computation (MoC) extension* to support runtime reconfiguration; and
3. the *mapping rules development* enabling the merge of the application and platform models into a single virtual implementation model in a systematic way.

1.2.2 Research Challenges

The main contribution of this research work is to provide a step towards the runtime reconfigurable embedded systems design automation supported by a rigorous formal basis. To achieve this, some challenges had to be overcome.

Regarding the application model, the synchronous MoC does not specify a process to represent reconfigurable devices, yet, it is general enough to allow for the definition of one. The *definition of a reconfigurable process*, using the synchronous MoC semantics, that includes

properties such as reconfiguration memory and reconfiguration time, is one of the challenges of this thesis. The SADF MoC, on the other hand, does specify a process that represents reconfigurable devices. Unfortunately, SADF is only represented in terms of its extra functional properties as either structural or performance models, lacking a functional representation. Defining a *functional semantics* for SADF is another one of this thesis' challenges.

Besides the application model, a platform model must also be provided as an input to the proposed design flow. This research work provides a possible platform model for runtime reconfigurable embedded systems that is general enough to represent most commercial off-the-shelf (COTS) reconfigurable devices.

The last challenge considered here concerns the definition of a *minimum set of mapping rules* to combine the application and platform models without necessarily forcing the developer to use application and platform models presented in this thesis. To overcome this challenge, the development of a classification system is necessary to abstract away specific semantics from the MoCs chosen for the application model (synchronous and SADF) as well as specific functional blocks from the platform model provided in this thesis, so that mapping rules can be build on top of such classification system, allowing any other MoC or platform model, complying with the classification system, to use the same set of mapping rules.

1.3 Related Work

This section presents related works on the three pillars of this PhD thesis: models in system design, design of runtime reconfigurable (RTR) systems and functional languages for system design.

1.3.1 Models in System Design

Modeling is probably the most important tool in any engineering discipline. Models are used to describe engineering artifacts, to validate and test them. In the development of cyber-physical systems, a combination of computers and physical systems, models of computation are extensively used along with models of physical phenomena. Lee (2015) argues, however, that such models do not combine well in terms of determinism. The notion of time is very different between computers and physical systems. For a computer, time is dictated by discrete clock pulses, and computer tasks are often subjected to race conditions which may lead to different execution orders for the same processes. For physical systems, however, time is continuous

and there is always a global order of events in the system. As a consequence, Lee argues that cyber-physical systems comprise a new engineering discipline which demands new models and methods.

Following the same line of thought, Tripakis (2016) discusses the different disciplines that together may form a science of systems design. According to him, system design, like any other design activity, is partly an art. Among the disciplines mentioned, modeling is a significant challenge when it comes to selecting the right modeling language, *i.e.* the best way to describe cyber-physical systems. He states that modeling languages must be executable, which means, at minimum, allowing simulation. Also, a key principle in system design is compositionality, in which complex systems are composed of smaller and more simple systems connected with each other following a semantics of composition.

Teich (2012) states the importance of starting system design with a functional and executable system specification, which brings benefits to the software development process by acting as a golden reference for the test environment. It also states the importance of formal models of computation which can be used to prove important system properties such as boundedness, timing and cost.

Sifakis (2015) presents a survey on system design automation. He presents the key aspects of the success of electronic design automation in the circuit industry, among them building faithful system models, and to what extent these aspects could be transposed to general system design. He also presents the challenges and limitations that need to be overcome in order to achieve system design automation.

1.3.2 Design of RTR Systems

Nguyen et al. (2019) quantify the benefits of runtime reconfiguration for two embedded vision applications. Both applications are composed of many different tasks, however all tasks are not required at the same time. The strategy used was to map only the tasks needed at a given time to a smaller FPGA area, instead of mapping all tasks statically to a large FPGA area. Such strategy reduced resource utilization up to 3.2 times, device cost up to 10 times and power consumption up to 30%. This work evidences the great benefit of using runtime reconfiguration for large applications composed of many tasks that can be scheduled over time, allowing them to be deployed in smaller FPGA areas, thus reducing cost and power consumption.

Implementing runtime reconfigurable systems in an FPGA with dynamic partial reconfiguration requires floorplanning, the task which defines a region of the reconfigurable area for each functional block in the system. Sadeghi et al. (2020) proposed a method for simultaneous floorplanning and placement for runtime reconfigurable systems using genetic algorithm. The application of the proposed method resulted in improvements in both wirelength and critical

path delay over Xilinx's early access partial reconfiguration design flow with a small increment in used area and runtime.

Still on the subject of floorplanning, Dorflinger et al. (2017) proposes an algorithm for assigning reconfigurable modules to reconfigurable regions at runtime. Three approaches were proposed: fixed uniform reconfigurable region size, fixed non-uniform reconfigurable region size and adaptive reconfigurable region size.

Runtime reconfiguration is an interesting strategy to increase fault tolerance in cyber-physical systems. Thomas et al. (2015) presents a flight control system for multirotor unmanned aerial vehicles that can reconfigure itself in runtime in case of a motor failure. The idea is simple: if a motor in the multirotor presents malfunction, the system can automatically switch to a configuration in which such motor is not used. Experiments were performed comparing three designs: mode reconfiguration, function reconfiguration and function reconfiguration with PID control to simulate a practical application. Although function reconfiguration provided a better resource usage, the impact of the switching process was clearly observed, as it was considerably slower than mode reconfiguration in the design with the PID control.

1.3.3 Functional Languages for System Design

The functional programming paradigm is showing to be a powerful tool for embedded systems design due to its declarative nature. Applications can be modeled and simulated using models of computation, and platforms can be designed using functional languages as a hardware description language (HDL).

Formal System Design (ForSyDe) is an example of functional languages used for modeling and simulation. It is a framework for modeling applications using formal models of computation created by Sander (2003). It is implemented as an embedded domain specific language (EDSL) in Haskell and utilizes the concept of process constructors, higher-order functions that outputs processes, as the means to implement specific MoC semantics. It supports several models of computation including the synchronous, Synchronous Dataflow (SDF), and now, as a contribution of this PhD thesis, scenario-aware dataflow (SADF) MoCs (Bonna et al., 2019b). ForSyDe has a spin-off denoted ForSyDe-Atom (Ungureanu et al., 2021), also implemented as a EDSL in Haskell, that utilizes a different approach for implementing MoC semantics and a layered system that explores orthogonalization of concerns.

Regarding functional languages used as HDL, Baaij et al. (2010) developed a language, named CλaSH, for describing synchronous hardware using the syntax and semantics of Haskell. The language has a prototype compiler that translates descriptions written in CλaSH to synthesizable VHDL. Similarly, Aronsson and Sheeran (2017) developed a library for programming FPGAs using Haskell. Such library supports hardware software co-design, and the same Haskell

code generates the software, in C, the hardware, in VHDL, and the code to support communication between software and hardware. The usage of the library is demonstrated in a cryptographic example.

1.4 Document Structure

This document is structured as follows: Chapter 2 presents a theoretical background on the concepts used along with the research; Chapter 3 introduces the application domain ontology as well as the application model using both the synchronous and scenario-aware dataflow models of computation; Chapter 4 brings the platform domain ontology, as well as a platform model general enough to represent most COTS reconfigurable devices; Chapter 5 presents the mapping rules used to combine both application and platform models; and Chapter 6 holds the conclusion of this research work, as well as possible future research.

2 THEORETICAL BACKGROUND

This chapter presents the theoretical background used throughout the thesis. First, a preliminary section containing the common notation is presented, followed by an introduction on Models of Computation (MoCs) with the *tagged signal model*. Then, a theoretical overview on Synchronous (SY), Synchronous Dataflow (SDF) and Scenario-Aware Dataflow (SADF), the three MoCs used in this thesis, is presented with comprehensive examples. Finally, a brief introduction on ForSyDe and Haskell, the MoC framework used to implement the models in this thesis, is given.

2.1 Common Notation

This section introduces the notations adopted in this thesis.

- Sets of numbers:
  - Natural numbers: Represented by $\mathbb{N}$ including 0, and $\mathbb{N}_+$ excluding 0;
  - Integers: Represented by $\mathbb{Z}$;
  - Real numbers: Represented by $\mathbb{R}$.
- Matrices, vectors and scalars:
  - Matrices: Represented by bold capital letters, *e.g.* $\mathbf{A}$;
  - Vectors: Represented by bold lowercase letters, *e.g.* $\mathbf{a}$;
  - Scalars: Represented by math style lowercase letters, *e.g.* a.

The finite sets of input and output ports of a process ρ are denoted by $\mathcal{I}_\rho$ and $\mathcal{O}_\rho$ respectively, and define $\mathcal{I}$ and $\mathcal{O}$ to be the union of all input and output ports of all processes in a model.

2.2 Models of Computation (MoC)

In systems design, the most powerful tools available for a designer are models. For embedded systems design, it is no different. Designing such systems comprises understanding the physical world, since most embedded systems interact with the physical world as part of *cyber-physical systems* (CPS), as well as computational systems. Models of computation provides a way to design embedded systems as a composition of sequential and concurrent processes, with well defined semantics, so that models can be executed (*i.e.* simulated) and analysis techniques can be used to verify relevant properties.

Models of computation are a collection of rules describing what constitutes processes, how processes communicate with each other, and the execution and concurrency mechanisms (Ptolemaeus, 2014). To the present days, there is no better way to describe and classify certain models of computation than using the *tagged signal model*.

2.2.1 Tagged Signal Model

The *tagged signal model* is a denotational framework for comparing different models of computation introduced by Lee and Sangiovanni-Vincentelli (1998). In such framework, signals are defined as a set of events as in (2.1), while events are elementary unities of information composed by a tag, belonging to the set of tags T, and a value, belonging to the set of values V as in (2.2). The set of signals S of a model is the powerset of $T \times V$ (2.3). A signal, whose values belong to a set $V_\alpha \subset V$ is represented by (2.4) (called signal of V_α). For simplicity, (2.5) is used to represent the value $v_k \in V$ associated with the tag $k \in T$ in signal $\sigma \in S$.

$$\sigma = \{e_0, e_1, \dots\} \in S \tag{2.1}$$

$$e_i = (t_i, v_i) \in T \times V \tag{2.2}$$

$$S = \mathcal{P}(T \times V) \tag{2.3}$$

$$\mathcal{S}(V_\alpha) = T \times V_\alpha \subset S \tag{2.4}$$

$$\sigma[k] = v_k \in V,\ k \in T \tag{2.5}$$

The purpose of the tag is to introduce order between events. When it is possible to order every event of a model according to its tags, *i.e.* T is a *totally ordered set*, then the model belongs to the class of *timed MoCs*. In such class, the tags also introduce a notion of time instants, and sometimes are referred as time stamps. The synchronous (SY), the continuous time and the discrete time MoCs are examples of timed MoCs. On the other hand, if it is not

possible to establish an order for all events in a model, *i.e.* T is a *partially ordered set*, then the model belongs to the class of *untimed MoCs*. Dataflows are examples of untimed MoCs since it is only possible to order events in the same signal, but not on different signals.

In the tagged signal model, a process P is defined as a subset of S^N, for some $N \in \mathbb{N}$ defining the number of inputs and outputs. A particular $\boldsymbol{\sigma} \in S^N$ satisfies a process P when $\boldsymbol{\sigma} \in P$. In this case, such $\boldsymbol{\sigma}$ is called a behavior of the process. For $N \geq 2$, the process may be viewed as a relation between the N signals of $\boldsymbol{\sigma}$. A set of inputs to P is an externally imposed constraint $A \subseteq S^N$ such that $A \cap P$ is the set of acceptable behaviors. A process is determinate if, for any set of input signals A, it has exactly one behavior or no behavior, *i.e.* $|A \cap P| = 1$ or $|A \cap P| = 0$, and a process is functional if there is a single-valued mapping $F : S^m \to S^n$, with m being the number of inputs and n being the number of outputs ($m + n = N$), dictating the behavior of the process. If a model is composed by only functional processes, then the model is functional.

2.3 Synchronous (SY) MoC

In the tagged signal model, the Synchronous MoC belongs to the class of timed MoCs. Before formally defining the synchronous model, it is important to formally define what synchronous means in the tagged signal model.

Definition 2.1 (Synchronous)**.** Two events are synchronous if they have the same tag. Two signals are synchronous if, for every event in one signal, there is a synchronous event in the other signal and vice versa. A process is synchronous when all signals in any behavior of the process are synchronous between themselves. A model is synchronous when it is a composition of synchronous processes.

As a consequence of Definition 2.1, for every two events in a synchronous model, they either have the same tag, or one unambiguously precedes the other. This concept of order between events gives the synchronous model tag system a notion of time. In fact, the synchronous MoC is the most elementary timed MoC there is. Discrete and continuous time models are nothing but synchronous models with respectively discrete and continuous sets of tags[1]. This thesis considers only discrete time synchronous models, which are more appropriate for modeling computational systems, as part of the research scope.

The SY MoC relies on the perfect synchrony hypothesis, which states that neither communication nor computation consumes time. Although such hypothesis is technically unfeasible in real systems, SY MoC is very useful for modeling *logically timed* circuits, where time proceeds

[1] This notion is not common sense in the literature. Some authors do not consider continuous time models to be a type of synchronous MoCs.

as a sequence of discrete steps called "ticks", provided that the tick intervals are long enough so that all computation can be performed between two sequential ticks.

Another important property of the synchronous MoC is that the absence of an event is well defined. In fact, the absence of an event is defined as an event with a special value $\perp \in V$, called "bottom". The example 2.1 illustrates how a synchronous process works.

Example 2.1

Consider the synchronous process $P \subset S^3$, shown in Figure 2.1, that represents a two input adder.

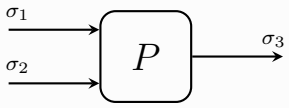


Figure 2.1. Process P with two inputs and one output.

The process P adds the values of every synchronous event between σ_1 and σ_2. As a functional process, its behavior is dictated by the function

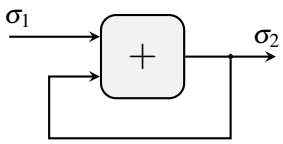
$$\sigma_3[k] = \begin{cases} \sigma_1[k] & \text{if } \sigma_2[k] = \perp \\ \sigma_2[k] & \text{if } \sigma_1[k] = \perp \\ \sigma_1[k] + \sigma_2[k] & \text{otherwise} \end{cases}$$

with $\sigma_i[k]$ being the value from signal σ_i associated with tag k. P is also determinate because for any input signals σ_1 and σ_2, there in only one possible σ_3.

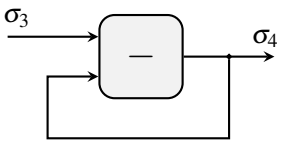
2.3.1 Feedback Loops and Causality

Feedbacks are elementary compositions of processes that connect one or several outputs of a process or a composition of processes to one or several inputs of itself. Due to the fact that computation and communication take no time in synchronous models, causality issues may arise when dealing with feedback loops with zero-delays. The output of a process with a time stamp (tag) k depends on itself in the same time stamp, resulting in an algebraic loop that may or may not have a single solution. Figure 2.2 shows two synchronous processes with feedback connections.

Consider the model shown in Figure 2.2a. The process is an adder, similar to the one in Example 2.1. In this model, the relation between σ_1 and σ_2 is given by (2.6). As one can



(a) Algebraic loop without single solution.



(b) Algebraic loop with single solution.

Figure 2.2. Synchronous models with zero-delay feedback loops.

notice, the output signal σ_2 at an instant k is a function of itself at the same time instant. This configures an algebraic loop that may lead to multiple behaviors. In this particular case, the model admits many possible behaviors given by $\sigma_1[k] = 0, \sigma_2[k] = a, \forall k \in T, a \in V$. Since σ_1 is an external input, it is not possible to constrain it to 0 for all $k \in T$.

$$\sigma_2[k] = \sigma_1[k] + \sigma_2[k] \tag{2.6}$$

Consider now the model shown in Figure 2.2b, in which the process subtracts the input events instead of adding them. The relation between σ_3 and σ_4 is given by (2.7). In this model, the output signal σ_4 at an instant k is also a function of itself at the same time instant, however the resulting algebraic loop admits a single solution that defines the behavior $\sigma_4[k] = \sigma_3[k]/2$, $\forall k \in T$.

$$\sigma_4[k] = \sigma_3[k] - \sigma_4[k] \tag{2.7}$$

As shown in the previous examples, zero-delay feedback loops can lead to multiple behaviors. For functional processes, the existence of a single solution for the algebraic loop may depend on the function associated with the processes composing the model. One conservative approach to deal with feedback loops and guarantee a single behavior to any feedback composition of processes is to insert a delay process to all feedback channels. Such delay process is defined as follows:

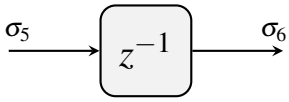


Figure 2.3. Synchronous delay process.

$$\sigma_6 = \texttt{delay}(a, \sigma_5)$$
$$\sigma_6[k] = \begin{cases} a & \text{if } k = 0 \\ \sigma_5[k-1] & \text{otherwise} \end{cases} \tag{2.8}$$

One important remark when using this approach is that the delay process added to feedback channels affects the behavior of the composed process, even when such composed process admits a single behavior, *i.e.* has a single solution to the resulting algebraic loop. Take the model of Figure 2.2b for instance, the behavior described by (2.7) is completely changed to $\sigma_4[k] = \sigma_3[k] - \sigma_4[k-1]$ when a delay is added to the feedback channel. In other words, by simply adding a delay in a feedback channel in order to avoid algebraic loops, one can lead to undesired behavior. Therefore, the delay must be taken into account when designing a composition of

processes with feedback loops.

One common use of delayed feedback channels in synchronous models is to store data between different time instants, simulating a memory. Example 2.2 shows a synchronous model of a Tustin integrator in which a feedback loop with a delay is used to store the previous output of the integrator.

Example 2.2

The figure below represents a synchronous model of a Tustin (trapezoidal) integrator.

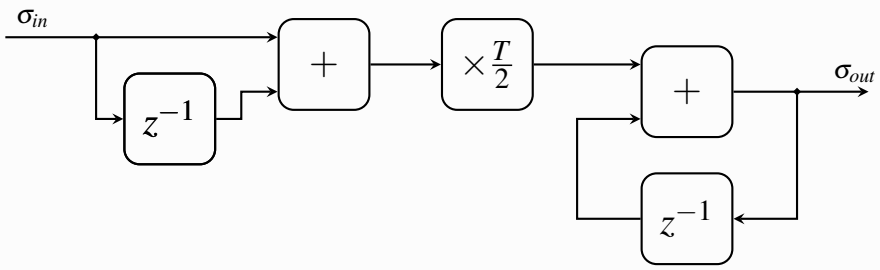


Figure 2.4. Synchronous model of a Tustin integrator.

The behavior of the model is represented by the equation below with T the sampling period, provided that the initial condition of the first delay equals $-\sigma_{in}[0]$, and the initial condition of the second delay (in the feedback loop) equals σ^*_{out}.

$$\sigma_{out}[k] = \begin{cases} \sigma^*_{out} & \text{when } k = 0 \\ \sigma_{out}[k-1] + \frac{T}{2}(\sigma_{in}[k] + \sigma_{in}[k-1]) & \text{otherwise} \end{cases}$$

2.4 Synchronous Dataflow (SDF) MoC

Synchronous dataflow belongs to the family of untimed models of computation named Dataflow. Dataflows are directed graphs where each node represents a process, called an actor, and each arc represents a signal path behaving as a first-in first-out (FIFO) channel. Each actor, when activated, *i.e.* fired, consumes a certain amount of tokens, elementary unities of information, from its input ports and generates a certain amount of tokens for its output ports. Each input and output ports have a positive integer associated with the number of tokens consumed by the input ports and produced by the output ports, also called token rates. An actor can only fire if input signal paths have enough tokens to supply the amount needed by the input ports of the actor, meaning that no signal path can have a negative amount of tokens. In general Dataflow models,

such as Dynamic Dataflows, the number of tokens consumed/generated by each actor port may not be constant, however, in SDF, this number is constant and defined beforehand. The fact that an SDF actor always consumes and generates the same amount of tokens allows efficient solutions to problems like finding a static schedule for single and multi-processor implementations and buffer size.

The formal definition of an SDF model is as follows.

Definition 2.2 (SDF model)**.** An SDF model is a 3-tuple $(P, S, \boldsymbol{\zeta}_0)$ where:

- P is a non-empty finite set of actors. Each actor $p_i \in P$ has a set of input ports $\mathcal{I}_i \subset \mathcal{I}$ and output ports $\mathcal{O}_i \subset \mathcal{O}$ and each element of $\mathcal{I}_i$ and $\mathcal{O}_i$ is associated with a fixed natural number representing the token consumption/generation rate;
- S is a finite set of signals. Each signal $\sigma_i \in S$ is defined by the 2-tuple (u_i, v_i), with $u_i \in \mathcal{O}$ and $v_i \in \mathcal{I}$. Signals store elementary units of information called *tokens* and behave like *first-in first-out* (FIFO) channels; and
- $\boldsymbol{\zeta}_0 \in \mathbb{N}^m$, with $m = |S|$, is the initial state of the model, *i.e.* the initial number of tokens stored in each signal σ_i.

2.4.1 SDF Properties

Many different properties can be derived from SDF models. Among them, the most important one is called *consistency*, and it serves as a necessary condition to many other properties such as absence of deadlocks and buffer boundness. This section presents the concept of consistency in an SDF model, and its relation to the existence of a schedule for the model.

An SDF model can be characterized by a matrix in a similar fashion to the incidence matrix associated with directed graphs in graph theory Lee and Messerschmitt (1987b). The procedure to construct this incidence matrix, called *topology matrix*, starts by enumerating each node (actor) and each arc (signal) and assigning a column to each actor and a row to each signal. The element $\Gamma_{i,j}$ of the topology matrix $\boldsymbol{\Gamma}$ denotes the number of tokens produced by the actor j for the signal i whenever fired (token rate). If the actor j consumes tokens from i, the number is negative. If the actor j has no connection to signal i, then the number is zero. If a signal connects an actor to itself, *i.e.*, a self-loop connection, only one entry in $\boldsymbol{\Gamma}$, given by the difference between produced and consumed tokens, describes this link.

Consider an SDF model with m signals and n actors characterized by the topology matrix $\boldsymbol{\Gamma} \in \mathbb{Z}^{m \times n}$. Consider also that only one actor may fire at a time instant, and time instants are represented by natural numbers. At an instant k the actor number l (with enough input tokens)

fires, and this firing is represented by the vector $\mathbf{v}(k) = [v_1, \ldots, v_n]^T$, with $v_i = 0$, $\forall i \neq l$ and $v_l = 1$.

The state of the model, *i.e.* the number of tokens of each signal, before actor number l fires (at instant $k-1$) is given by $\boldsymbol{\zeta}(k-1) \in \mathbb{N}^m$. Knowing $\mathbf{v}(k)$ and $\boldsymbol{\zeta}(k-1)$, the state of the model at instant k is, then, given by

$$\boldsymbol{\zeta}(k) = \boldsymbol{\zeta}(k-1) + \boldsymbol{\Gamma}\mathbf{v}(k) \tag{2.9}$$

A general solution to (2.9) can be found recursively, knowing the initial state $\boldsymbol{\zeta}_0$. Such solution is given by

$$\boldsymbol{\zeta}(k) = \boldsymbol{\zeta}_0 + \boldsymbol{\Gamma}\mathbf{q}(k) \tag{2.10}$$

where $\mathbf{q}(k) \in \mathbb{N}^n$ is the total firing vector, defined as the number of times each actor fired until instant k, given by

$$\mathbf{q}(k) = \sum_{i=1}^{k} \mathbf{v}(i) \tag{2.11}$$

As actors are fired in an SDF model, *i.e.* as k increases, so do the elements of the total firing vector $\mathbf{q}(k)$, making them unbounded as $k \to \infty$. Is it possible for the actors to keep firing indefinitely while the number of tokens in signal channels remains bounded? The answer to this question relies on an important property called *consistency*, defined below.

Definition 2.3 (SDF consistency)**.** A connected SDF model is consistent if there exists a sequence of actor firings $\mathbf{q}$ such that all actors fire at least once and the number of produced tokens is equal to the number of consumed tokens, *i.e.* if there exists some $\mathbf{q} \in \mathbb{N}_+^n$ such that

$$\boldsymbol{\Gamma}\mathbf{q} = \mathbf{0}$$

with $\boldsymbol{\Gamma} \in \mathbb{Z}^{m \times n}$ the topology matrix of the SDF model. In other words, if there exists some $\mathbf{q} \in \mathbb{N}_+^n \cap \mathcal{N}(\boldsymbol{\Gamma})$, with $\mathcal{N}(\boldsymbol{\Gamma})$ the null space of $\boldsymbol{\Gamma}$.

Consider a vector $\mathbf{q}(k)$, representing a sequence of k actor firings satisfying the condition presented on Definition 2.3, *i.e.* $\mathbf{q}(k) \in \mathbb{N}_+^n \cap \mathcal{N}(\boldsymbol{\Gamma})$. Consider also that such sequence is repeated α times. Then, from (2.10), the final state $\boldsymbol{\zeta}(\alpha k)$ is given by

$$\begin{aligned} \boldsymbol{\zeta}(\alpha k) &= \boldsymbol{\zeta}_0 + \boldsymbol{\Gamma}\alpha\mathbf{q}(k) \\ &= \boldsymbol{\zeta}_0 + \alpha\boldsymbol{\Gamma}\mathbf{q}(k) \\ &= \boldsymbol{\zeta}_0 \end{aligned}$$

For an α arbitrarily big, the state $\boldsymbol{\zeta}(\alpha k)$ will always return to the initial condition $\boldsymbol{\zeta}_0$. This is an important result since it proves that the number of tokens in the signals remains bounded, provided the SDF model is consistent and the actors are fired in a cyclic way such that, for every cycle, each actor is fired a certain number of times defined by its equivalent element in the vector

$\mathbf{q}$, called *repetition vector*. Moreover, consistency is also a necessary condition for the existence of a *Periodic Admissible Sequential Schedule* (PASS), which is defined as a sequence of actor firings (schedule) that, when executed periodically, the SDF model will never deadlock, and the signals will remain with a bounded number of tokens.

It would be extremely valuable if a simple method to check if a given SDF model is consistent or not existed. Fortunately, such method exists at the form of Theorem 2.1, which provides a necessary and sufficient condition for consistency in an SDF model.

Theorem 2.1 (SDF consistency condition (Lee and Messerschmitt, 1987a))**.** A connected SDF model with n actors and m signals, characterized by the topology matrix $\mathbf{\Gamma} \in \mathbb{Z}^{m \times n}$, is consistent if, and only if

$$\text{rank}(\mathbf{\Gamma}) = n - 1$$

With Theorem 2.1, consistency and, consequently, the existence of a repetition vector $\mathbf{q} \in \mathbb{N}_+^n \cap \mathcal{N}(\mathbf{\Gamma})$ are provable throughout a simple rank operation. Note, however, that for any consistent SDF model, there are an infinite number of vectors that are candidates to be used as a repetition vector. This is due to the fact that, for any repetition vector candidate $\mathbf{q}$, $\alpha\mathbf{q}$, with $\alpha \in \mathbb{N}_+$, is also a repetition vector candidate. Gaussian elimination can be used to find a repetition vector candidate and, by dividing all of its terms by their greatest common divisor, the minimum repetition vector can be found. The PASS requiring the smallest buffer size for the signals in the model is found considering the minimum repetition vector.

This section presented the property of consistency, and its importance for SDF models. A summary of how consistency is related to other important properties in SDF models is shown next:

- Consistency is a necessary condition for signal buffer boundness;
- Consistency is a necessary condition for the existence of a PASS;
- Consistency is a necessary and sufficient condition for the existence of a repetition vector; and
- The minimum repetition vector defines the PASS that requires the minimum signal buffer size.

Automated tools, such as SDF^3 (Stuijk et al., 2006), are useful to verify consistency and other related properties of SDF models.

Example 2.3 shows the concepts presented in this section applied to a small scale SDF model.

Example 2.3

Consider the SDF graph given by Figure 2.5. The numbers next to the input and output ports are the token rates of the ports.

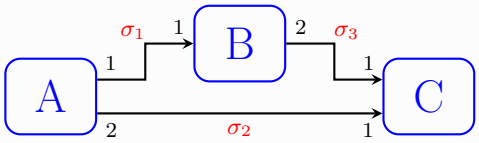


Figure 2.5. SDF model adapted from Ptolemaeus (2014).

Considering that the first column of $\mathbf{\Gamma}$ represents the actor A, followed by actors B and C in the second and third columns, and the first row represents signal σ_1, followed by signals σ_2 and σ_3 in the second and third rows, the topology matrix $\mathbf{\Gamma}$ is, then, given by

$$\mathbf{\Gamma} = \begin{bmatrix} 1 & -1 & 0 \\ 2 & 0 & -1 \\ 0 & 2 & -1 \end{bmatrix}$$

Assume zero initial state ($\boldsymbol{\zeta}_0 = \mathbf{0}$) and an arbitrary firing schedule given by $\langle$A,A,A,A,B,B,C$\rangle$ (fire four times A, followed by two times B and one time C). Also assume that only one actor can fire at a time, meaning that the firing of actor C is going to occur at the instant $k = 7$. The total firing vector $\mathbf{q}$, for $k = 7$, is $\mathbf{q}(7) = [4,2,1]^T$. The state of the model after the firing of C is given by

$$\boldsymbol{\zeta}(7) = \boldsymbol{\zeta}_0 + \mathbf{\Gamma}\mathbf{q}(7)$$
$$\boldsymbol{\zeta}(7) = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 1 & -1 & 0 \\ 2 & 0 & -1 \\ 0 & 2 & -1 \end{bmatrix} \begin{bmatrix} 4 \\ 2 \\ 1 \end{bmatrix} = \begin{bmatrix} 2 \\ 7 \\ 3 \end{bmatrix}$$

The SDF model is consistent since $\text{rank}(\mathbf{\Gamma}) = 2$ and its minimum repetition vector is $\mathbf{q} = [1,1,2]^T$. Therefore, a periodic sequence schedule for the model consists in firing A and B once, and C twice every execution cycle. Note that the only possible schedule that is deadlock free is $\langle$A,B,C,C$\rangle$.

2.5 Scenario-Aware Dataflow (SADF) MoC

As a general rule, the more expressive a model of computation is, the less analyzable it becomes Stuijk et al. (2011). Expressing dynamic behavior using SDF is impossible for most applica-

tions due to the restrictions imposed to the SDF MoC, *i.e.* fixed token rates. However, such restrictions ensure that a number of properties can be verified through analysis tools. Many extensions of SDF were proposed in order to increase expressiveness while trying to maintain it analyzable to properties such as consistency and absence of deadlocks. The *scenario-aware dataflow* (SADF) MoC is one of such extensions. This model of computation was firstly presented by Theelen et al. (2006) with the novelty of possessing two kinds of actors, *kernels* and *detectors*, and a finite state machine like behavior in which on each state the model operates as an SDF submodel.

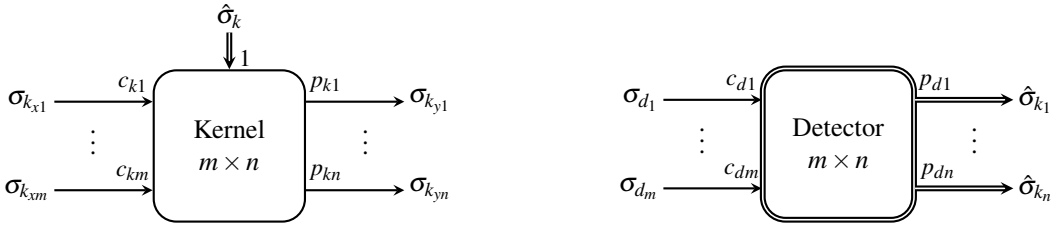
An SADF model is defined as follows:

Definition 2.4 (SADF model (Theelen et al., 2006))**.** An SADF model is described by a 4-tuple $(\mathcal{K}, \mathcal{D}, S, \boldsymbol{\zeta}_0)$ where:

- $\mathcal{K}$ is the finite set of kernels;
- $\mathcal{D}$ is the finite set of detectors;
- $S = S_d \cup S_c$ is a set of signals (FIFO channels) with S_d the set of data channels and S_c the set of control channels; and
- $\boldsymbol{\zeta}_0 \in \mathbb{N}^{|S|}$ is the initial state of the model, *i.e.* number of tokens in each channel.

A *kernel* (Figure 2.6a) is a reconfigurable actor whose behavior is defined by a kernel scenario. It has multiple data input and output ports and a single control input port, denoted by θ. The token consumption rate of the control port is always one, while the token consumption and production rates of the data ports are defined by the current kernel scenario. Each kernel $k \in \mathcal{K}$ has its own set of kernel scenarios Ψ_k and the union of all kernel scenarios in an SADF model is denoted by Ψ.

A *detector* (Figure 2.6b) is an actor that represents the control part of the SADF model. It has multiple data input ports, with fixed token consumption rates, and multiple control outputs with variable token production rates. Its behavior is dictated by a finite-state machine (FSM), and every time it fires, a transition of state is performed and the new state defines the scenario the SADF model will operate. To differentiate between the kernel scenario, which defines the behavior of a kernel, the concept of scenario referred here is going to be called detector scenario. Each detector scenario, represented by ϕ, can be seen as both an SDF subgraph and as a set of control tokens outputted by the detector. Any detector $d \in \mathcal{D}$ has a finite set of detector scenarios Φ_d and the set of all detector scenarios in an SADF model is denoted by Φ.



(a) Kernel with m inputs and n outputs.

(b) Detector with m inputs and n outputs.

Figure 2.6. SADF actor types.

2.5.1 SADF Properties

As mentioned before, SADF was developed to preserve some SDF analysis techniques to verify important properties. In previous sections, it was introduced the concepts of repetition vector and consistency for SDF, and how they are connected to other important properties. SADF possesses similar concepts, however, for SADF, the repetition vector is not constant, and consistency is not a unique property, but a family of properties. This section presents the SADF concepts of repetition vector and strong consistency, one of the forms of consistency for SADF models.

The definition of a repetition vector for SADF is dependent on the concept of *scenario synchronous*, defined below.

Definition 2.5 (Scenario Synchronous (Theelen et al., 2006))**.** A detector and all kernels connected to it via control signals operate in the same detector scenario. They are said to be mutually *scenario synchronous*.

The concept of repetition vector here is similar to SDF, in which each element q_i is the number of times the actor i must be fired within an evaluation cycle, so that all tokens produced can be consumed. However, differently from SDF, the repetition vector for SADF is not constant, but a function of the detector scenarios of the model. A definition of repetition vector for SADF is provided next.

Definition 2.6 (Repetition Vector (Theelen et al., 2006))**.** A repetition vector for an SADF model $(\mathcal{K}, \mathcal{D}, S, \boldsymbol{\zeta}_0)$ is a function $\mathbf{q} : \mathcal{K} \cup \mathcal{D} \times \Phi \to \mathbb{N}$ that assigns to all processes $\rho_1, \rho_2 \in \mathcal{K} \cup \mathcal{D}$ and all their detector scenarios, an element of $\mathbb{N}$ such that if output port $o \in \mathcal{O}_{\rho_1}$ is connected to input or control port $i \in \mathcal{I}_{\rho_2} \cup \theta_{\rho_2}$:

- if ρ_1 and ρ_2 are scenario synchronous, then $p(o,\phi)\mathbf{q}(\rho_1,\phi) = c(i,\phi)\mathbf{q}(\rho_2,\phi)$, for all $\phi \in \Phi$;
- if ρ_1 and ρ_2 are not scenario synchronous, then $p(o,\phi_1)\mathbf{q}(\rho_1,\phi_1) = c(i,\phi_2)\mathbf{q}(\rho_2,\phi_2)$, for all $\phi_1, \phi_2 \in \Phi$.

with $p(o,\phi)$ the token production rate of output port o and $c(i,\phi)$ the token consumption rate of input port i, operating in detector scenario ϕ.

The repetition vector $\mathbf{q}$ is said to be non-trivial if $\mathbf{q}(\rho,\phi) > 0$ for all $\rho \in \mathcal{K} \cup \mathcal{D}$ and $\phi \in \Phi$.

Consistency is also an important property for SADF models. However, differently from SDF, there is no unique notion of consistency for SADF. Next is the definition of strong consistency, one of such notions of consistency for SADF.

Definition 2.7 (Strong Consistency (Theelen et al., 2006))**.** An SADF model is strongly consistent if, and only if, it has a non-trivial repetition vector $\mathbf{q}$ such that for each detector $d \in \mathcal{D}$, $\mathbf{q}(d,\phi) = 1$ for all detector scenarios $\phi \in \Phi_d$. For a strongly consistent SADF, there is a unique minimum non-trivial repetition vector, which is designated as the repetition vector of the SADF model.

Weaker forms of consistency can be defined. What makes strong consistency special, however, is the fact that it avoids an undesired situation in which some detector d may fire more than once for a given scenario. Such situation is undesired due to the fact that every time d fires, it performs an state transition to a new state, and consequently, a new scenario, which may cause tokens to be consumed in a scenario different from the one that generated them.

Properties such as boundedness and absence of deadlocks can be verified for strongly consistent SADF models. Similar to SDF, this SADF background section will be limited to the strong consistency property. For more information regarding boundedness, absence of deadlocks and several other properties, refer to (Theelen et al., 2006).

2.6 Formal System Design (ForSyDe) and Haskell

Formal System Design (ForSyDe) is a philosophy for system design that aims to push design to a higher level of abstraction by combining functional programming with models of computation Sander et al. (2017). It is implemented as an embedded domain-specific language in the functional programming language Haskell. Several models of computation are implemented within ForSyDe, including the synchronous and SDF models. The ForSyDe Haskell library can be found in ForSyDe (2018).

In ForSyDe, a system is modeled as a network of concurrent processes communicating with each other through signals, where each signal has only one process writing on it. It uses the concept of process constructors, higher-order functions that have several arguments including functions and values and generate processes of an specific MoC.

The basic data structure used in ForSyDe is a signal, which is defined analogous to a list, where each element represents a value and its position represents its tag. In ForSyDe, a signal of data type `a` is defined as

```
data Signal a = NullS | a :- Signal a
```

Any signal of type a can be either empty (NullS) or a value of type a appended (:-) with another signal of type a. The notation $\mathcal{S}(\mathtt{a})$, introduced in (2.4), will be used to represent the Haskell Signal a data type as mathematical notation.

2.6.1 ForSyDe Synchronous Library

The ForSyDe synchronous library has several pre-defined process constructors that are used to model synchronous processes. Some of these process constructors are presented as follows.

comb*m*SY

The comb*m*SY process constructor is used to build synchronous processes with $m \in \{1,2,3,4\}$ inputs and one output. When $m = 1$, the number is omitted from the name of the process constructor. Considering $m = 2$ as an example, comb2SY takes a function $f : \mathtt{a} \times \mathtt{b} \rightarrow \mathtt{c}$ and returns a process of type $\mathcal{S}(\mathtt{a}) \times \mathcal{S}(\mathtt{b}) \rightarrow \mathcal{S}(\mathtt{c})$. Its type signature in Haskell is defined as follows.

```
comb2SY :: (a -> b -> c) -> Signal a -> Signal b -> Signal c
```

Process constructors combSY, comb3SY and comb4SY have equivalent implementation, but for 1, 3 and 4 inputs respectively.

unzip*n*SY

The unzip*n*SY process is used to split a signal composed of n different values into n signals, with $n \in \{2,3,4,5,6\}$. When $n = 2$, the number is omitted from the name, resulting in unzipSY. Considering $n = 3$ as an example, the process unzip3SY takes as input a composed signal $\mathcal{S}(\mathtt{a} \times \mathtt{b} \times \mathtt{c})$ and outputs three signals $\mathcal{S}(\mathtt{a})$, $\mathcal{S}(\mathtt{b})$ and $\mathcal{S}(\mathtt{c})$. Its type signature in Haskell is defined as follows.

```
unzip3SY :: Signal (a, b, c) -> (Signal a, Signal b, Signal c)
```

Combined with comb*m*SY, the unzip*n*SY process is useful to define synchronous processes with m inputs and n outputs.

delaySY

The delaySY process constructor is used to build delay process (z^{-1}). It takes a value $v \in \mathtt{a}$ and a signal $s \in \mathcal{S}(\mathtt{a})$ and returns a signal of type $\mathcal{S}(\mathtt{a})$ given by v:-s. Its type signature in Haskell is defined as follows.

```
delaySY :: a -> Signal a -> Signal a
```

2.6.2 ForSyDe SDF Library

Just as the synchronous library, the ForSyDe SDF library has several process constructors that can be used to implement most SDF based models. In total, there are 16 process constructors in the form of `actor`*mn*`SDF`, with $m,n \in \{1,2,3,4\}$, and a `delaySDF` with the same implementation as `delaySY` that is useful to define initial tokens to signals.

Taking as an example the process constructor `actor11SDF`, it takes two integer numbers c and p (token consumption and production rates respectively) and a function $f : \mathtt{a}^c \rightarrow \mathtt{b}^p$ and returns an SDF process of type $\mathcal{S}(\mathtt{a}) \rightarrow \mathcal{S}(\mathtt{b})$. Its type signature in Haskell is defined as follows.

```
actor11SDF :: Int -> Int -> ([a] -> [b]) -> Signal a -> Signal b
```

Note that differently from the functions passed as arguments for the synchronous process constructors, the functions used as behaviors for SDF processes are of type `[a] -> [b]`, representing a mathematical function of type $\mathtt{a}^c \rightarrow \mathtt{b}^p$, since it consumes c elements from the input port and produces p elements in the output port every time it is fired.

For multiple inputs and outputs, the first two arguments of the process constructor becomes tuples of integers. As an example, the type signature of process constructors `actor22SDF`, in Haskell, is defined as follows.

```
actor22SDF :: (Int, Int) -> (Int, Int) -> ([a] -> [b] -> ([c], [d]))
           -> Signal a -> Signal b -> (Signal c, Signal d)
```

3 APPLICATION MODEL

The *application model* captures functional requirements for the desired system. The designer describes the needed system behavior as a network of concurrent processes communicating through signals on a MoC-based framework. This given framework provides extensions that describe reconfiguration in an abstract way, thus hiding specific details about the implementation technology from the designer's perspective. In this way, one can focus on developing a *functional executable specification* without restrictions imposed by the available technology.

This chapter also presents an *application domain ontology*, *i.e.* a classification system that provides relations between different application model elements, and the *application semantics*, in which Synchronous and the scenario-aware dataflow MoCs are used to provide the execution constraints for the application model.

3.1 Application Domain Ontology

The design flow proposed in this thesis considers two branches as starting points, as shown in Figure 1.2, one for the platform and one for the application. The branches are connected with each other via mapping rules. This section presents a classification system, derived from a domain ontology, for the application branch that serves as a meta-model in which different runtime reconfigurable embedded systems can be built on top of it. The mapping rules defined in Chapter 5 consider, as entry point, application models that are in accordance with the classification system presented here.

To define such classification system, it is necessary to identify the basic entities that compose a runtime reconfigurable application and explore the common properties of these entities by grouping them into classes. As a first classification level, each entity from an application model is classified as either a *value*, representing a unity of information, a *procedure*, representing a process that transforms values, or a *path*, representing a unidirectional communication path that allows procedures to exchange values. This first subdivision level allows the representation of application models as directed graphs in which nodes represent procedures, and arcs represent paths.

Procedures, paths and values are further divided into subclasses for better representation of runtime reconfigurable systems. The domain ontology illustrated in Figure 3.1 represents the

proposed classification system for runtime reconfigurable systems as a tree, with the application as its root and the realizable entity classes as its leaves. There are two possible relations between two different classes, represented by dashed and continuous arrows. The dashed arrow represents a relation of the type *is part of*, meaning that a class is composed by other classes, while the continuous arrow represents a relation of the type *is a*, which indicates that a class is a sub-class of another class.

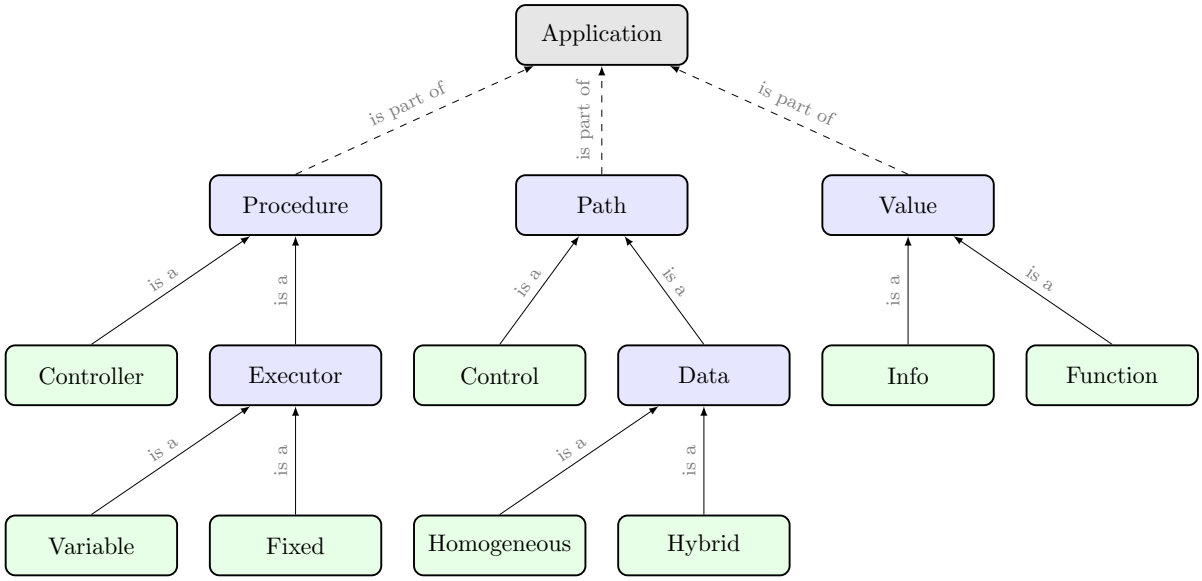


Figure 3.1. Application domain ontology. Nodes in green color represent classes with realizable entities.

In terms of sets, saying that the Procedure set *is part of* the Application set means that the Procedure set is one term of a Cartesian product that defines the Application set. Therefore, the Application set is the Cartesian product of the Procedure, Path and Value sets, as described below:

$$\text{Application} = \text{Procedure} \times \text{Path} \times \text{Value} \tag{3.1}$$

In a similar fashion, the *is a* relation stands for a sub-set operation in term of sets. Therefore, saying that the Controller set *is a* Procedure set, for instance, means that the Controller set is a sub-set of the Procedure, as described below:

$$\text{Controller} \subset \text{Procedure} \tag{3.2}$$

Entities from the three main classes, *i.e.*, procedure, path and value, as well as their sub-classes, are described in details as follows:

1. Procedure: Entity whose purpose is to transform values. It has input ports to import values from the exterior and output ports to export transformed values. Its main attribute

is a behavior, which defines the rules of transformation between the input and output values. It is divided between two different subclasses: Controller and Executor.

(a) Controller: A procedure whose behavior defines a transformation between values from the Info domain subclass to values from the Function domain subclass (and possibly values from the Info domain subclass as well). Controllers are entities responsible for defining behaviors to variable executors.

(b) Executor: A procedure whose behavior defines a transformation between values from the Info domain class to values from the Info domain class. It is divided into two different subclasses: Variable and Fixed.

i. Variable: An executor procedure whose behavior is a meta-behavior, *i.e.* it transforms values from the Info domain class using, as rules of transformation (*i.e.* behaviors), values from the Function domain class. It has a unique input port, denoted as control port, for values of the Function domain. Every variable executor is connected to a single controller via its control port.

ii. Fixed: A executor procedure whose behavior is fixed and only operates using values from the Info domain class.

2. Path: Entity whose purpose is to transport values between procedures. It sets a communication lane between two procedures in a single direction.

(a) Control: A path whose transported values are from the Function domain class. It connects a controller procedure to a variable executor procedure.

(b) Data: A path whose transported values are from the Info domain class. It is divided into two subclasses: Homogeneous and Hybrid.

i. Homogeneous: A data path that connects executor procedures among themselves.

ii. Hybrid: A data path that connects executor procedures and controller procedures.

3. Value: Entity that is exchanged between procedures via paths. It is divided into two domain classes: Info and Function.

(a) Info: A domain class of values that represents elementary pieces of information.

(b) Function: A domain class of values that represents a transformation, or a map, between values.

In order to define an entity as belonging to one of the realizable entity classes from the application ontology (green nodes from Figure 3.1), the notation used throughout this thesis considers the name of each hierarchical level of the ontology separated by dots, starting from the highest level, *e.g.* the entity G is a fixed executor procedure, then $G \in$ `Procedure.Executor.Fixed`.

For application models represented as directed graphs, Table 3.1 shows the defined standards of symbols adopted throughout this thesis to represent entities from different classes.

Table 3.1. Procedure and path model symbols and subclasses.

Class	Symbol	Subclass
`Procedure.Executor`		`Variable` and `Fixed`
`Procedure.Controller`		–
`Path.Data`		`Homogeneous` and `Hybrid`
`Path.Control`		–

Runtime reconfigurable processes belong to the `Procedure.Executor.Variable` application ontology class. As such, these processes have meta-behaviors, which can be seen either as multiple different behaviors defined by functions inputted via control paths, or as a single behavior defined by a *function application* function. The concept of function application function acting as a behavior will be further explored when defining semantics for the application model in the following sections.

Processes belonging to the `Procedure.Controller` application ontology class are also fundamental for RTR embedded systems. Such controller processes are responsible for managing the reconfiguration of all runtime reconfigurable processes. No restrictions on the number of controllers are imposed in the application model, however, due to contraints in the platform's technology, it might be necessary to impose a single controller architecture. This subject will be further explored during the platform model definition in the next chapter.

3.2 Application Semantics

The application model needs semantics to define the rules that will dictate the execution of its functional elements and how they communicate amongst themselves. The synchronous and the scenario-aware dataflow semantics are used for such purpose. The synchronous model of computation was chosen because it is the most elementary timed MoC there is and, as all timed MoCs, can be used to model real-time systems. On the other hand, the scenario-aware dataflow model was chosen as a representative of the untimed MoCs, which are ideal for streaming applications. It is important to note that other MoCs can be used to define semantics for runtime reconfigurable systems, however, they are out of scope of this thesis.

The SADF model of computation has kernel and detector processes, which naturally classifies them as entities from the `Procedure.Executor.Variable` and `Procedure.Controller` application ontology classes. Unfortunately, the synchronous MoC has no such thing and, therefore, processes from both these procedure classes must be defined. In the next sections both synchronous and SADF semantics for runtime reconfigurable applications are explored.

3.2.1 Synchronous Application Semantics

The synchronous model of computation defines rules of execution and communication between processes communicating with each other via signal paths in a network of processes. It does not constraint any process to a specific type of behavior, nor the kind of values that can be processed and exchanged between processes. As such, all realizable entities from the application ontology can be implemented using the synchronous semantics.

This section is focused on defining a general-purpose runtime reconfigurable process, from the `Procedure.Executor.Variable` application ontology class, using the synchronous semantics. First, a representation of a runtime reconfigurable process is defined in a high level of abstraction using the simplest form of meta-behavior: a *function application* function. Then, such process is further developed to include concepts of behavior and data storage, as well as reconfiguration time. All those concepts are present in current reconfigurable hardwares, such as FPGAs. The remaining classes of realizable entities from the application domain ontology are defined according to the specific application and will not be discussed in this section. An application example of a synchronous triple modular redundancy architecture, with runtime reconfiguration, is shown at the end of this section.

The simplest form of meta-behavior that can be used for entities that are instances of the `Procedure.Executor.Variable` application ontology class is a *function application* function. To understand its concept, it is important to think of functions as common values, *i.e.* belonging to the set of values V, as defined in the application domain ontology. As such, functions can be arguments to other functions as well as returned values. With that in mind, the operator `apply` : $(V^m \to V^n) \times V^m \to V^n$ defines a function application function for functions of m arguments and n return values. It applies a function $f : V^m \to V^n$ to arguments $x_i \in V_i \subset V$ such that

$$\texttt{apply}(f, x_1, \ldots, x_m) = f(x_1, \ldots, x_m) \tag{3.3}$$

Languages following the functional programming paradigm have, as a basic principle, the concept of functions as common values. They are derived from Lambda Calculus (Michaelson, 2011), a mathematical formalism that introduces the concept of lambda terms, which can be either values, functions or function applications. As such, all lambda terms can be used in the

same way. When John Backus coined the term *algebra of programs*, talking about functional languages in his Turin Award Lecture (Backus, 1978), he was referring to programs as functions, and functions as common data, which allows operations between programs. As an example, the functional language Haskell defines the infix operator $ as the function `apply` for functions with single argument and return values, *i.e.* $m = n = 1$. In Haskell, the operator $ is implemented as:

```
($) :: (a -> b) -> a -> b
f $ x = f x
```

A synchronous process whose behavior is defined by the `apply` function (3.3) is named *synchronous function placeholder* and defines a synchronous runtime reconfigurable process in a high level of abstraction. Such process is defined next.

Definition 3.1 (Synchronous function placeholder)**.** A synchronous process, belonging to the `Procedure.Executor.Variable` application ontology class, possessing an unique input port responsible for inputting functions of type $V^m \to V^n$ representing the processes' behavior at instant $k \in T$. Such input port is connected to a signal $\hat{\sigma} : T \times (V^m \to V^n)$. Figure 3.2 shows a representation of a function placeholder and equation (3.4) shows the functional behavior of the function placeholder.

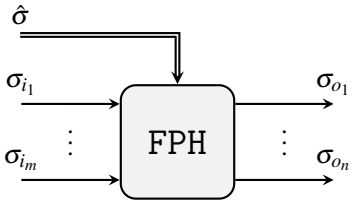


Figure 3.2. Function placeholder.

$$\begin{aligned} &\texttt{FPH}: \mathcal{S}\left(V^m \to V^n\right) \times \prod_{j=1}^{m} \mathcal{S}\left(V_{i_j}\right) \to \prod_{j=1}^{n} \mathcal{S}\left(V_{o_j}\right) \\ &\left(\sigma_{o_1}[k], \ldots, \sigma_{o_n}[k]\right) = \texttt{apply}(\hat{\sigma}[k], \sigma_{i_1}[k], \ldots, \sigma_{i_m}[k]) \end{aligned} \tag{3.4}$$

To define a synchronous function placeholder for m input data and n output data using the concept of process constructors, one needs a synchronous process constructor for $m+1$ inputs and n outputs. Using the process constructors of ForSyDe/Haskell, a synchronous single input, the single output function placeholder `fph11` is defined as follows:

```
fph11 :: Signal (a -> b) -> Signal a -> Signal b
fph11 = comb2SY ($)
```

The Example 3.1 shows an encoder/decoder system model using function placeholders and process constructors from ForSyDe/Haskell.

As previously mentioned, a synchronous function placeholder is an abstraction of a synchronous runtime reconfigurable process at a high level. In such level, parts like configuration memory, data memory and reconfiguration time are abstracted away. The configuration signal $\hat{\sigma}$ carries the behaviors to be executed by the process at every time instant k, meaning that if one wants to model a synchronous runtime reconfigurable process using a function placeholder,

Example 3.1

Figure 3.3 presents an encoder/decoder system model, adapted from (Sander et al., 2017), in which numeric keys flow through σ_{key} and are processed by both *genEnc* and *genDec*. Each of these processes outputs encoding/decoding functions $f(key,x)$ and $f^{-1}(key,x)$ respectively. These functions are used as behaviors for the processes ap_{Enc} and ap_{Dec}.

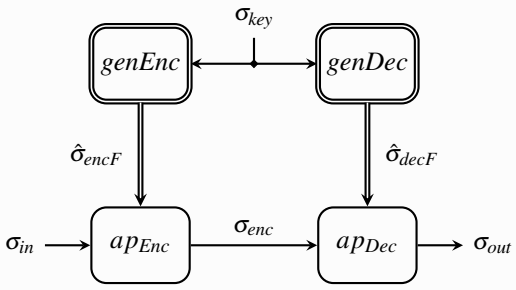


Figure 3.3. Encoder decoder system.

Using ForSyDe/Haskell, the encoder/decoder system model previously mentioned is implemented in the following code snippet:

```
1  -- Process Definitions
2  genEnc = comb2SY (\key x -> x+key)  -- f(key,x) = x + key
3  genDec = comb2SY (\key x -> x-key)  -- f^(-1)(key,x) = x - key
4  ap_enc = fph11    -- ap_enc is a function placeholder
5  ap_dec = fph11    -- ap_dec is a function placeholder
6  -- Encoder/Decoder System (eds) process network
7  eds s_key s_in = (s_enc, s_out)
8     where s_enc = ap_enc s_encF s_in
9           s_out = ap_dec s_decF s_enc
10          s_encF = genEnc s_key
11          s_decF = genDec s_key
```

for every time instant k, $\hat{\sigma}[k]$ must be the function to be executed at that time instant, even if it is the same function used in the previous time instant $k-1$.

Delayed feedback loops are used to include a configuration memory, to store the current behavior, a data memory, to store states or data, and reconfiguration time, to simulate the idle state during reconfiguration. Figure 3.4 shows a synchronous runtime reconfigurable process (RTRP) including three delayed feedback loops to represent a configuration memory, a data memory, and a reconfiguration countdown, used to simulate the time it takes to perform reconfiguration. An extra input $\bar{x}$ is used to define initial conditions for the data memory, represented by $x \in \mathbb{S}$, after a reconfiguration process.

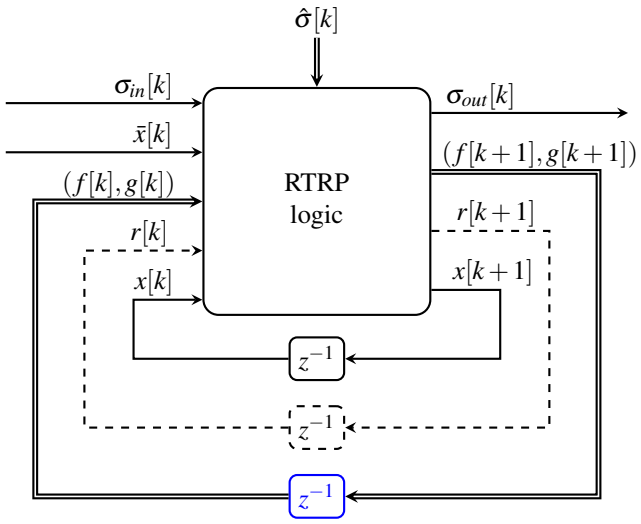


Figure 3.4. Synchronous runtime reconfigurable process with a configuration memory represented by the delay in blue, a data memory represented by the delay in black, and a reconfiguration countdown represented by the dashed delay in black.

The functional behavior of the RTRP, stored in the configuration memory, is represented by a pair of functions $f : \mathbb{S} \times V^m \to \mathbb{S}$ and $g : \mathbb{S} \times V^m \to V^n$, comprising a memory changing operation (or the next state function) and an output function respectively. The reconfiguration signal $\hat{\sigma}$ transmits such functions. In any time instant k, $\hat{\sigma}[k]$ can be either a 3-tuple $(\mathbf{f},\mathbf{g},\mathbf{r})$, representing a reconfiguration request, or an empty value $\perp$. When a reconfiguration request arrives at the reconfiguration input port, two things happen: a reconfiguration countdown lasting **r** cycles is started, and the functions **f** and **g** are stored in the configuration memory. Equations (3.5) and (3.6) represent such behavior.

$$r[k+1] = \begin{cases} \mathbf{r}-1 & \text{if } \hat{\sigma}[k] = (\mathbf{f},\mathbf{g},\mathbf{r}) \\ r[k]-1 & \text{if } \hat{\sigma}[k] = \perp \text{ and } r[k] > 0 \\ 0 & \text{if } \hat{\sigma}[k] = \perp \text{ and } r[k] = 0 \end{cases} \tag{3.5}$$

$$(f[k+1],g[k+1]) = \begin{cases} (\mathbf{f},\mathbf{g}) & \text{if } \hat{\sigma}[k] = (\mathbf{f},\mathbf{g},\mathbf{r}) \\ (f[k],g[k]) & \text{if } \hat{\sigma}[k] = \perp \end{cases} \tag{3.6}$$

Stored functions f and g define a state machine behavior for the RTRP when the reconfiguration is finished and no reconfiguration request arrives via control signal $\hat{\sigma}$. Such behavior is represented by equations (3.7) and (3.8).

$$x[k+1] = \begin{cases} \perp & \text{if } \hat{\sigma}[k] \neq \perp \text{ or } r[k] > 0 \\ \texttt{apply}(f[k],\bar{x}[k],\sigma_{in}[k]) & \text{else if } x[k] = \perp \\ \texttt{apply}(f[k],x[k],\sigma_{in}[k]) & \text{otherwise} \end{cases} \tag{3.7}$$

$$\sigma_{out}[k] = \begin{cases} \perp & \text{if } \hat{\sigma}[k] \neq \perp \text{ or } r[k] > 0 \\ \texttt{apply}(g[k], \bar{x}[k], \sigma_{in}[k]) & \text{else if } x[k] = \perp \\ \texttt{apply}(g[k], x[k], \sigma_{in}[k]) & \text{otherwise} \end{cases} \tag{3.8}$$

Note that in the event of a reconfiguration request ($\hat{\sigma}[k] \neq \perp$), the data memory represented by x is emptied, as described by the first line of equation (3.7). Whenever the reconfiguration finishes, *i.e.* $r[k] = 0$ after $r[k-1] > 0$, the input $\bar{x}$ is used as the current memory data, *i.e.* current state, as represented by the second line of equations (3.7) and (3.8). This allows the RTRP to start from any desired initial state and is particularly important for synchronization with other devices using a shared memory.

With equations (3.5), (3.6), (3.7) and (3.8), combined with the `apply` operator (3.3), one can formally define a synchronous runtime reconfigurable process' semantics. In the next section, an example of a triple modular architecture with runtime reconfiguration will be shown using multiple RTRP.

Synchronous Application Example

A triple modular redundancy with runtime reconfiguration from (Bonna et al., 2019a) is used as an example of the synchronous application semantics. This example is interesting for safety-critical systems, exploring concepts of fault detection and fault mask, as well as synchronization after reconfiguration. The example also presents how the model entities are classified within the proposed application ontology.

Figure 3.5 shows a triple modular redundancy architecture model with runtime reconfiguration. Such model comprises $j > 3$ runtime reconfigurable processors (RTRPs), a voting mechanism to detect possible faults, a control device that acts as a reconfiguration manager when the voter detects a fault in one of the RTRPs, and a current state shared memory (CSSM) that acts as a synchronization memory between different RTRPs after a reconfiguration event.

Although this architecture has a number $j > 3$ of RTRPs, only three of them are active simultaneously. The remaining RTRPs are inactive and are either in a idle state waiting to be reconfigured and assume an active duty, or performing different signal processing activities they are initially configured to perform. Active RTRPs have the same inputs, internal states and behavior, meaning they should produce the same results. In the event of a fault in one of the three active RTRPs, the voter, a process that acts as a voting mechanism, detects a mismatch in one of the outputs from the RTRPs. Figure 3.6 shows the voter and its input and output signals.

The voter is connected to all j RTRPs, however it is only processing the outputs of the three active RTRPs. The Control Device keeps track of the three active RTRPs and informs the voter

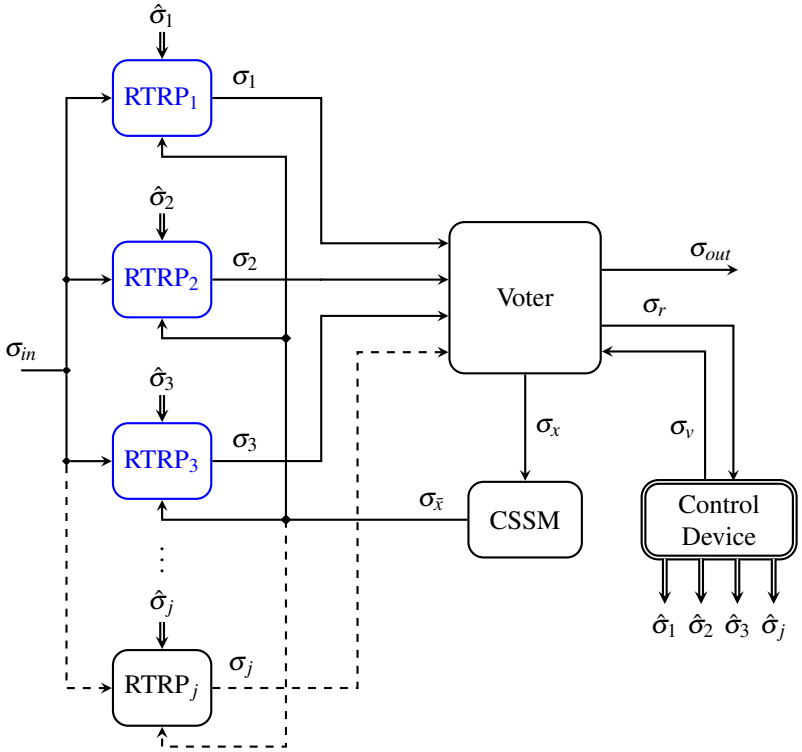


Figure 3.5. Triple modular redundancy example adapted from (Bonna et al., 2019a).

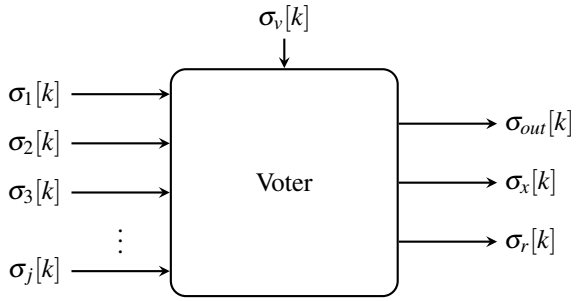


Figure 3.6. Voter input and output signals (Bonna et al., 2019a).

every cycle via signal σ_v. Equations (3.9) and (3.10) define the voter's input signals σ_v and σ_i.

$$\sigma_v[k] = (a,b,c), \quad a,b,c \in \{1,2,\ldots,j\} \tag{3.9}$$

$$\sigma_i[k] = (y_i[k], x_i[k]), \quad i \in \{1,2,\ldots,j\} \tag{3.10}$$

Note that each RTRP sends to the voter both its calculated output y_i and its internal state x_i via signal σ_i.

The output of the voter σ_{out} is the RTRP output with the highest occurrence between the three active RTRPs, and the output σ_x is the current state of the RTRP whose output is the one with the highest occurrence. Equation (3.11) defines the voter's behavior regarding outputs σ_{out}

and σ_x.

$$(\sigma_{out}[k], \sigma_x[k]) = \begin{cases} (y_a[k], x_a[k]) & \text{if } y_a[k] = y_b[k] \text{ or } y_a[k] = y_c[k] \\ (y_b[k], x_b[k]) & \text{if } y_b[k] = y_c[k] \end{cases} \tag{3.11}$$

In the event of a mismatch between one of the three active RTRPs, the voter informs the Control Device (Figure 3.7) which RTRP is faulty via signal σ_r. Equation (3.12) defines such behavior.

$$\sigma_r[k] = \begin{cases} \perp & \text{if } y_a[k] = y_b[k] = y_c[k] \\ a & \text{if } y_a[k] \neq y_b[k] = y_c[k] \\ b & \text{if } y_b[k] \neq y_a[k] = y_c[k] \\ c & \text{if } y_c[k] \neq y_a[k] = y_b[k] \end{cases} \tag{3.12}$$

Note that in the event of a triple mismatch in the voter caused by two or more faulty active RTRPs at the same time instant k, the voter fails to produce an output. A more robust voter logic could be used to avoid such situation, however this is only an illustrative example of an application model for an RTR system, and the quality of the voter's logic is of minor importance here.

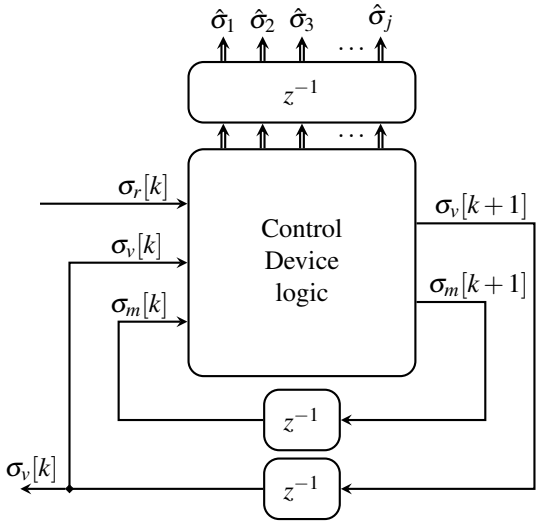


Figure 3.7. Control device internal structure (Bonna et al., 2019a).

The Control Device keeps track of the current three active RTRPs via σ_v, as well as tracks the reconfiguration timeout, when reconfiguring a new RTRP, via σ_m. If $\sigma_m[k] = 0$, then at instant k, the Control device is in the *Ready* state, meaning it is ready to perform a new reconfiguration when necessary. If $\sigma_m[k] > 0$, then at instant k, the Control Device is in the *Reconf* state, meaning it is reconfiguring some RTRP and, therefore, cannot start a new reconfiguration process until the current reconfiguration is finished.

When the Control Device receives from the voter a value different than $\perp$ via signal σ_r, and is in Ready state, it starts a new reconfiguration process. The state is switched to the Reconf

state and a reconfiguration timer is started lasting $\mathbf{r}$ cycles

$$\sigma_m[k+1] = \begin{cases} \mathbf{r} & \text{if } \sigma_r[k] \neq \perp \text{ and } \sigma_m[k] = 0 \\ \sigma_m[k]-1 & \text{if } \sigma_m[k] > 0 \\ 0 & \text{otherwise} \end{cases} \tag{3.13}$$

The output σ_v carries the current three active RTRPs as a 3-tuple, such as in (3.9). The behavior of the Control Device regarding the output σ_v is modeled as described in (3.14).

$$\sigma_v[k+1] = \begin{cases} \sigma_v[k] & \text{if } \sigma_r[k] = \perp \text{ or } \sigma_m[k] > 0 \\ h(\sigma_v[k], \sigma_r[k]) & \text{if } \sigma_r[k] \neq \perp \text{ and } \sigma_m[k] = 0 \end{cases} \tag{3.14}$$

with $h((a,b,c),x)$, a function that selects the next RTRP in line to be reconfigured, is given by

$$h((a,b,c),x) = \begin{cases} (\max(a,b,c)+1, b, c) & \text{if } x = a \\ (a, \max(a,b,c)+1, c) & \text{if } x = b \\ (a, b, \max(a,b,c)+1) & \text{if } x = c \end{cases} \tag{3.15}$$

The behavior of the Control Device, given by equations (3.13) and (3.14), is summarized in the state chart as illustrated in Figure 3.8.

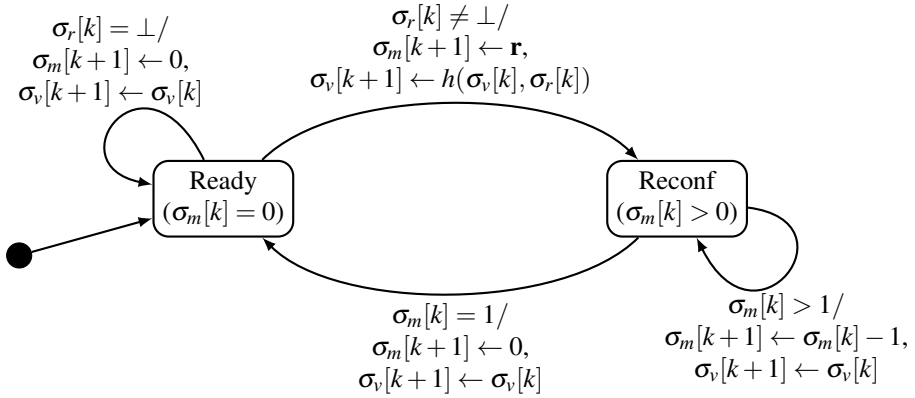


Figure 3.8. State chart that defines the behavior of the Control Device (Bonna et al., 2019a).

Finally, the control outputs $\hat{\sigma}_i$, with $i \in \{1, \ldots, j\}$, behave as follows. When the transition from Ready to Reconf is *taken*, *i.e.* $\sigma_r[k] \neq \perp$ and $\sigma_m[k] = 0$, the Control Device outputs a reconfiguration signal given by the 3-tuple $(\mathbf{f}, \mathbf{g}, \mathbf{r})$ to the output number $i = \max(\sigma_v[k]) + 1$ (the next available RTRP in the line), and outputs $\perp$ for the remaining RTRPs. In any other case, it

outputs $\perp$. Such behavior is given by

$$\hat{\sigma}_i[k+1] = \begin{cases} (\mathbf{f},\mathbf{g},\mathbf{r}) & \text{if } \sigma_r[k] \neq \perp \text{ and } \sigma_m[k] = 0 \text{ and } \max(\sigma_v[k]) + 1 = i \\ \perp & \text{otherwise} \end{cases} \tag{3.16}$$

The initial conditions to σ_m, σ_v and $\hat{\sigma}_i$ must be provided. As a general rule, these initial conditions are: $\sigma_m[0] = 0$, $\sigma_v[0] = (1,2,3)$ and $\hat{\sigma}_i[0] = \perp$. These initial conditions indicate the system starts with all RTRPs already configured and the Control Device in the Ready state.

The current state shared memory (CSSM) is implemented as a delay and receives as input the voter's output σ_x (the RTRP state whose output has higher occurrence) and outputs $\sigma_{\bar{x}}$, connected to all RTRPs, even the spare ones. When a reconfiguration is finished in one RTRP, the states stored in the CSSM are used as initial states for the newly reconfigured RTRP, as shown in equations (3.7) and (3.8).

The triple modular redundancy with runtime reconfiguration application model presented in this section is implemented using the process constructors of ForSyDe/Haskell. The complete Haskell code can be found in Section A.1. For simulation purposes, the RTRP's behavior is defined as an accumulator. The functions $\mathbf{f}$ and $\mathbf{g}$ that define such behavior are given by equations (3.17) and (3.18). To simulate a faulty behavior, function $\tilde{\mathbf{f}}$ (3.19) is implemented in a way that, for a specific state, the function performs a computation that differs from $\mathbf{f}$. The spare RTRPs are implemented with the negative accumulator function $\bar{\mathbf{f}}$ given by equation (3.20), representing a different task for the spare RTRPs.

$$\mathbf{f}(x,u) = x + u \tag{3.17}$$

$$\mathbf{g}(x,u) = x \tag{3.18}$$

$$\tilde{\mathbf{f}}(x,u) = \begin{cases} x - u & \text{if } x = 3 \\ x + u & \text{otherwise} \end{cases} \tag{3.19}$$

$$\bar{\mathbf{f}}(x,u) = x - u \tag{3.20}$$

The implemented application model comprises 5 RTRPs ($j = 5$) in which RTRPs 1 and 3 are initially configured with the pair $(\mathbf{f},\mathbf{g})$, RTRP 2 is initially configured with the pair $(\tilde{\mathbf{f}},\mathbf{g})$, while RTRPs 4 and 5 are initially configured with the pair $(\bar{\mathbf{f}},\mathbf{g})$. The reconfiguration time $\mathbf{r}$ is chosen to be 2 cycles and the input σ_{in} is a constant equals 1.

Table 3.2 shows the outputs of each RTRP y_i, as well as the input σ_{in}, the voted output σ_{out} and the mismatch signal σ_r, as a function of the cycle k. When $k = 4$, the voter detects a mismatch in RTRP 2, informing the Control Device via signal σ_r. During cycles 5 and 6, RTRP 4 is being reconfigured to assume the position of active RTRP. From cycle 7 onwards, the active RTRPs are 1, 3 and 4, producing the same results.

Table 3.3 shows the model entities and corresponding classes in the application domain

Table 3.2. Simulation results obtained from the ForSyDe/Haskell implementation.

k	0	1	2	3	4	5	6	7	8	9
σ_{in}	1	1	1	1	1	1	1	1	1	1
σ_{out}	0	1	2	3	4	5	6	7	8	9
y_1	0	1	2	3	4	5	6	7	8	9
y_2	0	1	2	3	2	3	2	3	2	3
y_3	0	1	2	3	4	5	6	7	8	9
y_4	0	-1	-2	-3	-4	$\perp$	$\perp$	7	8	9
y_5	0	-1	-2	-3	-4	-5	-6	-7	-8	-9
σ_r	$\perp$	$\perp$	$\perp$	$\perp$	2	4	4	$\perp$	$\perp$	$\perp$

ontology. Note that signals σ_{in} and σ_{out} are not in that table because in order to specify a class for a path, it is necessary to know the type of procedures they are connecting, although it is possible to define them as entities from the `Path.Data` class.

Table 3.3. Model entities and corresponding classes.

Entity	Application ontology class
RTRP_i	`Procedure.Executor.Variable`
Voter	`Procedure.Executor.Fixed`
CSSM	`Procedure.Executor.Fixed`
Control Device	`Procedure.Controller`
σ_i	`Path.Data.Homogeneous`
σ_x	`Path.Data.Homogeneous`
$\sigma_{\bar{x}}$	`Path.Data.Homogeneous`
σ_r	`Path.Data.Hybrid`
σ_v	`Path.Data.Hybrid`
$\hat{\sigma}_i$	`Path.Control`

3.2.2 SADF Application Semantics

In Section 2.5, the scenario-aware dataflow was introduced as an extension of the SDF MoC. That was made to cope with dynamic behavior while preserving most of the analysis tools and properties from the SDF. The SADF MoC was presented in terms of its extra functional characteristics, such as detector and kernel general behavior regarding token production and consumption. In the present section, the functional semantics is introduced to SADF, adding functional behaviors to kernels and detectors. The concepts presented in this section are based on the functional SADF model from the paper Bonna et al. (2019b), published at the ACM Transactions on Design Automation of Electronic Systems, and comprises the first major contribution of this doctoral research.

As specified in Section 2.5, the SADF MoC classifies processes into kernels and detectors. Kernels are processes whose token consumption and production are defined by a control token that is inputted through a unique control port every kernel possesses. Such control token carries a kernel scenario, defined as the pair of token consumption and production rates for each input and output port of the kernel. For the functional SADF model, besides consumption and production rates, the kernel scenario also provides a functional behavior for the kernel, *i.e.* a function that defines the token value transformation. In this sense, a kernel is defined next.

Definition 3.2 (Kernel)**.** A kernel k, belonging to the finite set of kernels $\mathcal{K}$, is defined by a 4-tuple $(\Psi_k, \theta_k, \mathcal{I}_k, \mathcal{O}_k)$ where:

- Ψ_k is the finite set of scenarios of k;
- θ_k is the control input port of k with token consumption rate 1;
- $\mathcal{I}_k$ is the set of input ports of k; and
- $\mathcal{O}_k$ is the set of output ports of k.

Before presenting the definition of kernel scenario, it is important to define the sequence set, denoted by the symbol Σ_a^n, where n is the length of the sequences in the set and a is the type of the elements in the sequences. Therefore, a sequence $s \in \Sigma_a^n$ represents an ordered sequence (or list) with n elements of type a. With the sequence set properly defined, the definition of kernel scenario is as follows.

Definition 3.3 (Kernel scenario)**.** A scenario ψ belonging to the finite set of scenarios Ψ_k of a kernel k with m inputs and n outputs is defined by a 3-tuple (c_ψ, p_ψ, f_ψ) where:

- $c_\psi = (c_{\psi 1}, \ldots, c_{\psi m}) \in \mathbb{N}^m$ denotes the consumption token rates for each input $i \in \mathcal{I}_k$;
- $p_\psi = (p_{\psi 1}, \ldots, p_{\psi n}) \in \mathbb{N}^n$ denotes the production token rates for each output $j \in \mathcal{O}_k$; and
- $f_\psi : \prod_{i=1}^{m} \Sigma_{\alpha_i}^{c_{\psi i}} \to \prod_{j=1}^{n} \Sigma_{\beta_j}^{p_{\psi j}}$ denotes the function for kernel k, where the length of each input sequence, with data type α_i, is determined by $c_{\psi i}$, and the length of each output sequence, with data type β_j, is determined by $p_{\psi j}$.

When a kernel k fires, it first consumes one control token from its control port. This control token sets the consumption c_ψ and production p_ψ rates as well as the function f_ψ for that execution. Then, it finishes firing by consuming the input tokens and producing the output tokens. Similar to the synchronous function placeholder, the kernel's behavior is defined by the `apply` function (3.3), receiving as inputs the function f_ψ and, for every input port $i \in \mathcal{I}_k$, sequences of values with length $c_{\psi i}$. In this sense, a kernel can be defined as a *dataflow function placeholder*.

Before formally defining a process constructor for kernels, it is important to define some operations associated with signals that are used in the process constructor's definition. Let $+\!\!+$

be a binary operator which concatenates two signals or sequences of the same base type, *e.g.* (3.21), provided that the first signal or sequence is finite; $\texttt{take}(n,\sigma)$ and $\texttt{drop}(n,\sigma)$ be two functions which take and drop n events from the beginning of a signal σ, *e.g.* (3.22) and (3.23); $|\sigma|$ be a set operation which returns the number of elements in σ; and $\varnothing$ denote an empty signal.

$$\{e_1,e_2\} +\!\!+ \{e_3,e_4,e_5,e_6\} = \{e_1,e_2,e_3,e_4,e_5,e_6\} \tag{3.21}$$

$$\texttt{take}(2,\{e_1,e_2,e_3,e_4,e_5\}) = \{e_1,e_2\} \tag{3.22}$$

$$\texttt{drop}(2,\{e_1,e_2,e_3,e_4,e_5\}) = \{e_3,e_4,e_5\} \tag{3.23}$$

Furthermore, the tuple notation $\left(a_i\right)_{i=1}^{m}$ denotes a tuple $(a_1,\ldots,a_m)$, and the operation $+\!\!+$ is performed element-wise for tuples of signals, *i.e.*

$$\left(\sigma_{1i}\right)_{i=1}^{m} +\!\!+ \left(\sigma_{2i}\right)_{i=1}^{m} = \left(\sigma_{1i} +\!\!+ \sigma_{2i}\right)_{i=1}^{m} \tag{3.24}$$

Note that both tuples must have the same size in order to perform the $+\!\!+$ operation.

Based on the definitions of kernel (Definition 3.2) and kernel scenario (Definition 3.3), a process constructor for a kernel with m inputs and n outputs is formally defined.

Definition 3.4 (Kernel process constructor (Bonna et al., 2019b))**.**

$$\texttt{kernel}_{mn} : \mathcal{S}\left(\mathbb{N}^m \times \mathbb{N}^n \times \prod_{i=1}^{m} \Sigma_{\alpha_i}^{c_{\psi i}} \to \prod_{j=1}^{n} \Sigma_{\beta_j}^{p_{\psi j}}\right) \times \prod_{i=1}^{m} \mathcal{S}\left(\alpha_i\right) \to \prod_{j=1}^{n} \mathcal{S}\left(\beta_j\right)$$

$$\texttt{kernel}_{mn}\left(\hat{\sigma}, \left(\sigma_i\right)_{i=1}^{m}\right) = \begin{cases} \left(\varnothing\right)_{j=1}^{n} & \textbf{if } \bigvee_{i=1}^{m} \left(|\sigma_i| < c_{\psi i}\right) \vee \left(|\sigma_\psi| = 0\right) \\ \texttt{apply}\left(f_\psi, \mathit{inps}\right) +\!\!+ \texttt{kernel}_{mn}\left(\overline{\hat{\sigma}}, \overline{\mathit{inps}}\right) & \textbf{otherwise} \end{cases}$$

$$\textbf{where} \quad \left(\left(c_{\psi i}\right)_{i=1}^{m}, \left(p_{\psi j}\right)_{j=1}^{n}, f_\psi\right) = \texttt{take}(1,\hat{\sigma})$$

$$\mathit{inps} = \left(\texttt{take}(c_{\psi i}, \sigma_i)\right)_{i=1}^{m}$$

$$\overline{\mathit{inps}} = \left(\texttt{drop}(c_{\psi i}, \sigma_i)\right)_{i=1}^{m}$$

$$\overline{\hat{\sigma}} = \texttt{drop}(1,\hat{\sigma})$$

As a result of the research (Bonna et al., 2019b), SADF was added as a MoC library in ForSyDe/Haskell. Such library possesses a collection of process constructors, including the family of process constructors `kernel`*mn*`SADF`, implemented following Definition 3.4, able to define kernels up to 5 inputs and 5 outputs ($m,n \in [1,5]$). Kernels defined using this family of process constructors possess a singular input signal of type $\mathcal{S}\left(c,p,f\right)$, where c and p are the token consumption and production rates for the data input and output ports respectively, and f is the function defining the behavior of the process in the current firing.

As an example, the process constructor `kernel11SADF` type signature in Haskell is defined as:

```
kernel11SADF :: Signal (Int, Int, [a] -> [b]) -> Signal a -> Signal b
```

Similarly to the SDF library, briefly introduced in Section 2.6, the token consumption and production rates are represented as integers and the function defining the behavior is of type `[a] -> [b]`. For multiple inputs and outputs, the token consumption and production rates are represented as tuples of integers. The complete SADF library of process constructors implemented in Haskell/ForSyDe can be found in Section A.2 and in the Git repository (ForSyDe, 2018).

Section 2.5 also introduced the concept of detectors, which are processes responsible for supplying the kernels with control tokens. Their functional behavior is dictated by a finite state machine (FSM) where each state defines a detector scenario. When there are enough tokens available in its input ports, the detector consumes these tokens and, based on their values, performs a state transition. Based on its new state of operation, a number of control tokens are supplied to the kernels connected to the detector via control channels. Differently from a kernel, a detector consumes a fixed amount of input tokens, but outputs a number of tokens that depends on is operational state. A functional detector is formally defined as follows:

Definition 3.5 (Detector)**.** A detector d, belonging to the finite set of detectors $\mathcal{D}$, is defined by a 6-tuple $(\Phi_d, \mathcal{I}_d, \mathcal{O}_d, c_d, \mathcal{F}_d, g_d)$ where:

- Φ_d is the finite set of scenarios of d;
- $\mathcal{I}_d$ is the set of input ports with $m = |\mathcal{I}_d|$ being the number of input ports;
- $\mathcal{O}_d$ is the set of output ports with $n = |\mathcal{O}_d|$ being the number of output ports;
- $c_d = (c_1, \ldots, c_m) \in \mathbb{N}^m$ denotes the consumption token rates for each input $i \in \mathcal{I}_d$;
- $\mathcal{F}_d$ is the 3-tuple $(\mathbb{S}, s^*, f_d)$ defining a deterministic FSM where $\mathbb{S}$ is the set of states, $s^* \in \mathbb{S}$ is the initial state, and $f_d : \mathbb{S} \times \prod_{i=1}^{m} \Sigma_{\alpha_i}^{c_i} \to \mathbb{S}$ is the state transition function; and
- $g_d : \mathbb{S} \to \Phi_d$ is the state to scenario function.

The concept of detector scenario differs from the concept of kernel scenario. For a kernel, a scenario is defined by the value inputted through its control port, *i.e.* the value carried by control tokens, which consists on the 3-tuple of Definition 3.3. For a detector, the concept of scenario can have multiple interpretations that are functionally equivalent between themselves. One interpretation would consist on an SDF subgraph composed by the detector and all kernels connected to it, each one operating on a specific kernel scenario defined by the control tokens supplied by the detector in a particular detector scenario. Another interpretation would be a set of sequences of control tokens that are outputted when the detector fires. This second interpretation is not only easier to formally define, but fits better for modeling runtime reconfigurable systems, which are the target systems of the design flow proposed in this thesis. Such interpretation is formally defined as:

Definition 3.6 (Detector's scenario)**.** A scenario ϕ belonging to the finite set of scenarios Φ_d of a detector d with n outputs is defined by a 2-tuple (p_ϕ, τ_ϕ) where:

- $p_\phi = (p_{d1}, \ldots, p_{dn}) \in \mathbb{N}^n$ denotes the production token rates for each control output $j \in \mathcal{O}_d$; and
- $\tau_\phi \in \prod_{j=1}^{n} \Sigma_{\Psi_{k_j}}^{p_{\phi j}}$ are the sequences of output control tokens for d, whose lengths are determined by $p_{\phi j}$ and each k_j is the kernel connected to output port number $j \in \mathcal{O}_d$.

Based on the definitions of detector (Definition 3.5) and detector scenario (Definition 3.6), a process constructor for a detector with m inputs and n outputs can be formally defined.

Definition 3.7 (Detector process constructor (Bonna et al., 2019b))**.**

$$\texttt{detector}_{mn} : \mathbb{N}^m \times \left(\mathbb{S} \times \prod_{i=1}^{m} \Sigma_{\alpha_i}^{c_i} \to \mathbb{S} \right) \times \left(\mathbb{S} \to \mathbb{N}^n \times \prod_{j=1}^{n} \Sigma_{\Psi_j}^{p_{\phi j}} \right) \times \mathbb{S} \times \prod_{i=1}^{m} \mathcal{S}(\alpha_i) \to \prod_{j=1}^{n} \mathcal{S}(\Psi_j)$$

$$\texttt{detector}_{mn} \left(\left(c_i \right)_{i=1}^{m}, f_d, g_d, s_d^*, \left(\sigma_i \right)_{i=1}^{m} \right) = output(g_d, \sigma_d)$$

$$\textbf{where} \quad \sigma_d = next\left(\texttt{delay}(s_d^*, \sigma_d), \left(\sigma_i \right)_{i=1}^{m} \right)$$

$$next\left(\sigma_t, \left(\sigma_i \right)_{i=1}^{m} \right) = \begin{cases} \varnothing & \textbf{if } \bigvee_{i=1}^{m} |\sigma_i| < c_i \\ f\left(\texttt{take}(1, \sigma_t), inps \right) +\!\!+ next\left(\texttt{drop}(1, \sigma_t), \overline{inps} \right) & \textbf{otherwise} \end{cases}$$

$$inps = \left(\texttt{take}(c_i, \sigma_i) \right)_{i=1}^{m}$$

$$\overline{inps} = \left(\texttt{drop}(c_i, \sigma_i) \right)_{i=1}^{m}$$

$$output(g, \sigma_t) = \begin{cases} \left(\varnothing \right)_{j=1}^{n} & \textbf{if } \sigma_t = \varnothing \\ g\left(\texttt{take}(1, \sigma_t) \right) +\!\!+ output\left(g, \texttt{drop}(1, \sigma_t) \right) & \textbf{otherwise} \end{cases}$$

The function `delay` used in Definition 3.7 is the equivalent of (2.8) to SADF MoC. The mathematical formulation is the same, however for SADF it is used to define initial tokens in channels.

In ForSyDe/Haskell, the family of process constructors `detector`*mn*`SADF` takes four arguments that are sufficient to fully define a detector's behavior: the token consumption rates from the input ports, the state transition function, the state to scenario function, and an initial state. As an example, the process constructor `detector11SADF` type signature in Haskell is defined as next.

```
detector11SADF :: Int -> (s -> [a] -> s) -> (s -> (Int, [y])) -> s
               -> Signal a -> Signal y
```

The output tokens of a detector are control tokens for the kernels that are connected to it. Therefore the type `y` seen in the type signature previously presented is actually a 3-tuple (c, p, f) representing token consumption and production rates, and a function.

SADF Application Example

To demonstrate the concepts presented in the last section, a comprehensive example of a reduced instruction set computer (RISC) processor is considered. This example is in Bonna et al. (2019b), but with emphasis to the model development using ForSyDe/Haskell. Here, the emphasis is given to the general functionality of the model, *i.e.* token rates and functions. The complete code developed in ForSyDe/Haskell can be found in Section A.3.

The RISC processor example is modeled with two kernels and one detector, as shown in Figure 3.9. The Instruction Fetch (IF) kernel stores the program memory in a self feedback loop, and is responsible for fetching an instruction, updating the program counter (also stored in the kernel with a self feedback loop), supplying the Decode (DEC) detector with the opcode of the current instruction, and the Execute (EXE) kernel with its arguments. The DEC detector decodes the opcode and provides both kernels, IF and EXE, with the scenario to be executed. The EXE kernel executes the instruction decoded by the detector with the arguments supplied by the IF kernel. It is also where the program memory and register bank are stored using self feedback loops. Note that, due to the balance of consumption and production token rates of every actor, the RISC SADF model is strongly consistent, and every actor fires once for any possible detector scenario.

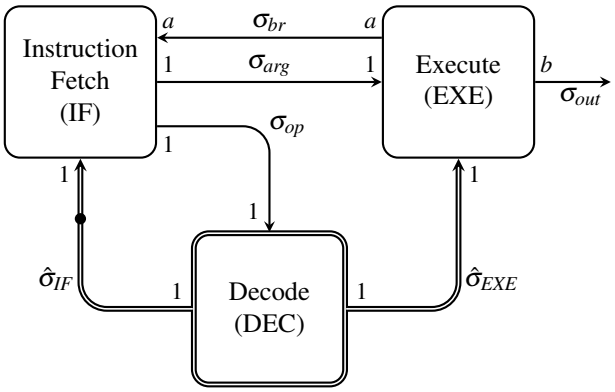


Figure 3.9. RISC processor modeled with SADF (Bonna et al., 2019b). An initial token must be provided to the control channel connected to the IF kernel (represented as a bullet •).

The instruction set implemented in this processor is shown in Table 3.4. The instructions are divided into 4 different classes: load and store (LS), arithmetic and logic (AL), branches (BR) and outputs (OP), totalizing 20 instructions.

Figure 3.10 shows the instruction fetch kernel internal signals. Self feedback signals are used to store the program memory and program counter. The IF kernel possesses two scenarios: branch and no-branch. In the no-branch scenario, it fetches in the program memory, represented as a single token with a list of strings (each string representing a line of instruction), the instruction at the program counter's location; sends the opcode to the DEC detector and the arguments to the EXE kernel; and finally, adds 1 (*i.e.* increments) to the program

Table 3.4. RISC processor complete instruction set.

n	Opcode	Args	Effect	Class
01	`ld`	`rd mn`	`rd` ← `mem(mn)`	LS
02	`ldr`	`rd rm`	`rd` ← `mem(rm)`	LS
03	`st`	`rs mn`	`mem(mn)` ← `rs`	LS
04	`str`	`rs rm`	`mem(rm)` ← `rs`	LS
05	`mov`	`rd rs`	`rd` ← `rs`	LS
06	`movi`	`rd i`	`rd` ← `i`	LS
07	`add`	`rd rs`	`rd` ← `rd + rs`	AL
08	`sub`	`rd rs`	`rd` ← `rd - rs`	AL
09	`mul`	`rd rs`	`rd` ← `rd` × `rs`	AL
10	`div`	`rd rs`	`rd` ← `rd` ÷ `rs`	AL
11	`and`	`rd rs`	`rd` ← `rd & rs`	AL
12	`or`	`rd rs`	`rd` ← `rd | rs`	AL
13	`xor`	`rd rs`	`rd` ← `rd` ∧ `rs`	AL
14	`bez`	`rs j`	`if rs == 0 then PC` ← `PC+j+1 else PC` ← `PC+1`	BR
15	`bnz`	`rs j`	`if rs` ≠ `0 then PC` ← `PC+j+1 else PC` ← `PC+1`	BR
16	`bgz`	`rs j`	`if rs > 0 then PC` ← `PC+j+1 else PC` ← `PC+1`	BR
17	`blz`	`rs j`	`if rs < 0 then PC` ← `PC+j+1 else PC` ← `PC+1`	BR
18	`jmp`	`j`	`PC` ← `PC+j+1`	BR
19	`outr`	`rs`	`print(rs)`	OP
20	`outm`	`mn`	`print(mem(mn))`	OP

counter. In the branch scenario, it fetches the instruction pointed by the program counter, added to a relative jump value inputted via σ_{br}, before performing the remainder of the steps just like the no-branch scenario.

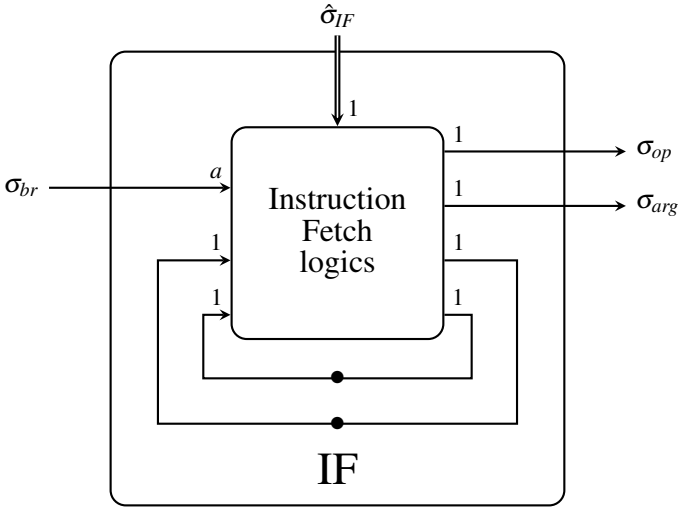


Figure 3.10. Instruction fetch (IF) kernel. Self feedback loops are used to store program memory and program counter. Initial tokens are represented as bullets •.

The branch scenario of the IF kernel comprises the instructions from the branch (BR) class (instructions 14 to 18), and it is characterized by having the token consumption a, of the input

port connected to σ_{br}, equals 1, as shown in Table 3.5. The remaining instructions from the instruction set are no-branch scenario for the IF kernel.

Figure 3.11 shows the EXE kernel internal signals. Self feedback signals are used to store the register bank and data memory, both as a list of integers with each element as a single register or memory slot. The EXE kernel possesses one scenario for each of the 20 instructions available. Table 3.5 shows the token rate values a, b, r and m for each scenario. Note that some token rate values are exclusive for a specific class of instruction, *e.g.* $a = 1$ is exclusive for branch instructions, $b = 1$ is exclusive for output instructions and $m = 1$ is exclusive for load and store instructions, as well as the `outm` (output directly from memory) instruction.

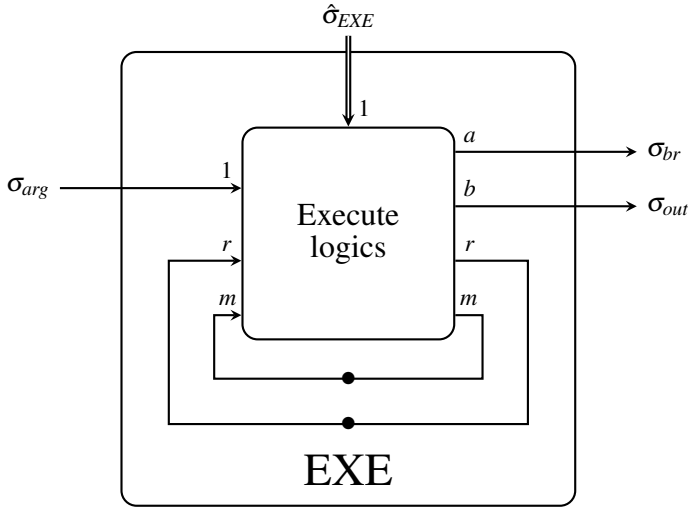


Figure 3.11. Execute (EXE) kernel. Self feedback loops are used to store data memory and register bank. Initial tokens are represented as bullets •.

Figure 3.12 shows the DEC detector. It possesses a state for each one of the instructions from the instruction set. The transition of states is performed based only on the opcode value inputted through σ_{op}, *i.e.* it does not depend on the current state to perform a state transition. Therefore, for each opcode inputted, a state transition is performed to its equivalent state, represented by the integer **n**. The scenario outputted for the EXE kernel, via signal $\hat{\sigma}_{EXE}$, sets the token rates according to Table 3.5, and the function according to the desired effect given by Table 3.4. The scenario outputted to the IF kernel, via signal $\hat{\sigma}_{IF}$, is either a branch scenario, if $14 \leq \mathbf{n} \leq 18$, or a no-branch scenario.

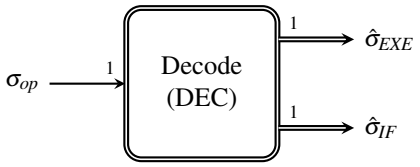


Figure 3.12. Processor decode (DEC) detector.

Table 3.6 shows how the RISC processor entities are classified within the application domain ontology. Signal σ_{out} is not classified in the table because it could be either an entity of the

Table 3.5. Token consumption and production for each scenario.

n	**Opcode**	a	b	r	m
01	`ld`	0	0	1	1
02	`ldr`	0	0	1	1
03	`st`	0	0	1	1
04	`str`	0	0	1	1
05	`mov`	0	0	1	0
06	`movi`	0	0	1	0
07	`add`	0	0	1	0
08	`sub`	0	0	1	0
09	`mul`	0	0	1	0
10	`div`	0	0	1	0
11	`and`	0	0	1	0
12	`or`	0	0	1	0
13	`xor`	0	0	1	0
14	`bez`	1	0	1	0
15	`bnz`	1	0	1	0
16	`bgz`	1	0	1	0
17	`blz`	1	0	1	0
18	`jmp`	1	0	0	0
19	`outr`	0	1	1	0
20	`outm`	0	1	0	1

`Path.Data.Homogeneous` class, or `Path.Data.Hybrid` class, depending on which procedure, besides EXE, it is connected to.

Table 3.6. RISC processor model entities and corresponding classes.

Entity	Application ontology class
IF	`Procedure.Executor.Variable`
EXE	`Procedure.Executor.Variable`
DEC	`Procedure.Controller`
σ_{br}	`Path.Data.Homogeneous`
σ_{arg}	`Path.Data.Homogeneous`
σ_{op}	`Path.Data.Hybrid`
$\hat{\sigma}_{IF}$	`Path.Control`
$\hat{\sigma}_{EXE}$	`Path.Control`

For the complete example regarding the RISC processor, refer to Bonna et al. (2019b) and ForSyDe/Haskell code in Section A.3.

4 PLATFORM MODEL

The *platform model* captures all platform related characteristics of the system, such as memory, communication buses and processing units.

This chapter presents a platform domain ontology, a classification system that provides the relations between different elements within a platform, and platform functional blocks, which describes, in detail, a variety of possible functional blocks for a runtime reconfigurable platform.

4.1 Platform Domain Ontology

The other branch of the proposed design flow (as illustrated in Figure 1.2) for runtime reconfigurable systems, comprises all platform related characteristics of the system.

This section presents a classification system, derived from a domain ontology, whose purpose is the same as the application domain ontology, *i.e.* to provide a foundation in which different platforms can be classified on top of it. The mapping rules defined in Chapter 5 also consider, as entry point, platforms that are in accordance with the platform domain ontology presented here.

The proposed platform domain ontology follows the concept of the *fixed plus variable* computer firstly presented by Estrin (1960). Such computer consists in a fixed general-purpose processor and a variable hardware area, which, in today's technology, would be composed of a microcontroller and an FPGA. Back in the '60s, the motivation to build such system was to solve problems not practicably computable, and to drastically increase the overall computation speed. Nowadays, the fixed plus variable computer concept is presented in system on chips (SoCs), which are widely used in industries such as automotive and aerospace, and provides the necessary flexibility to implement different solutions without the need to physically replace the hardware. Moreover, updates can be released for both software and hardware, decreasing the overall system obsolescence.

In the platform domain ontology, a platform is defined as the union of two subsystems: *fixed* and *variable*. Each subsystem is then composed of the three basic classes of entities presented in any computational machine: *processing unit*, *communication* and *memory*. For the variable subsystem, the existence of processing units, communication or memories depends on the application configured. Figure 4.1 shows the platform domain ontology represented as a

graph. The Greek letter χ is used to differentiate entities of the variable subsystem. Note that there can be entities belonging to both the fixed and the variable subsystems, which is the case of communication paths between the subsystems. In this case, such entities are grouped in the class *Mixed*. There are three possible relations between two different classes, represented by dashed, continuous and double arrows. The dashed arrow represents a relation of the type *is part of*, meaning that a class is composed by other classes. The continuous arrow represents a relation of the type *is a*, which indicates that a class is a sub-class of another class. Finally, the double arrow represents a relation of the type *associated with*, which indicates that two classes have entities that are paired with each other.

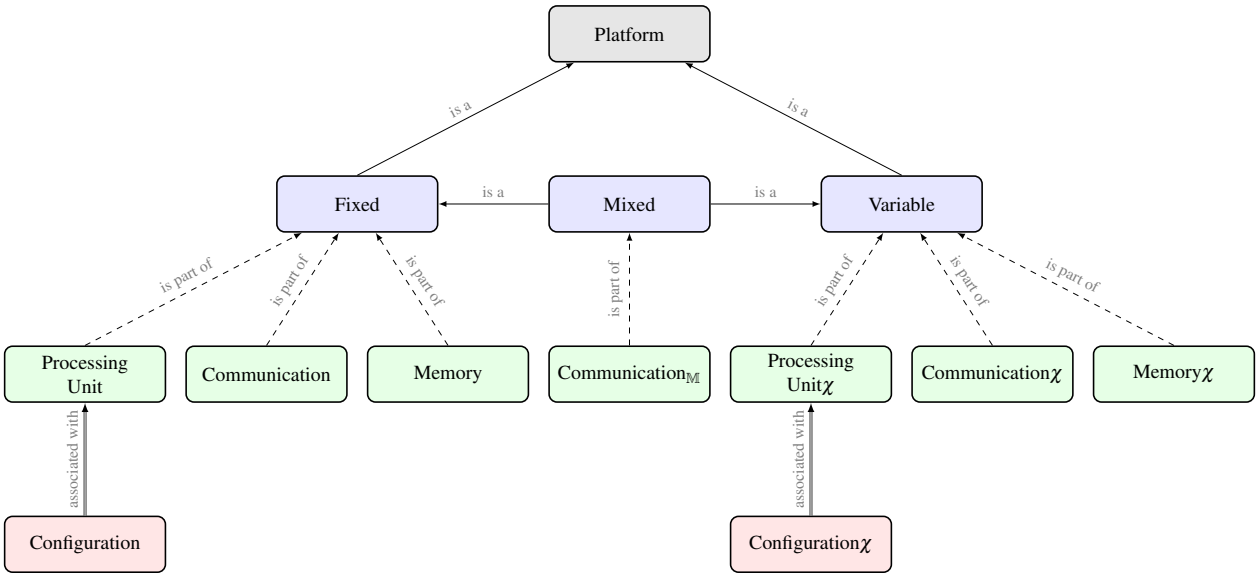


Figure 4.1. Platform domain ontology. Nodes in blue represent subsystems (mixed is considered a subsystem since it is the intersection of fixed and variable subsystems), nodes in green represents realizable entity classes and nodes in red are classes of entities that are associated with entities from another class.

As introduced in the application ontology, the relation *is a*, in terms of sets, expresses a subset relation. Therefore, the following relations are derived from the platform domain ontology:

$$\text{Fixed} \subset \text{Platform} \tag{4.1}$$

$$\text{Variable} \subset \text{Platform} \tag{4.2}$$

$$\text{Platform} = \text{Fixed} \cup \text{Variable} \tag{4.3}$$

$$\text{Mixed} = \text{Fixed} \cap \text{Variable} \tag{4.4}$$

Table 4.1 introduces the notation used to represent the realizable entity classes as sets. Using this notation, and knowing that the relation *is part of* represents a Cartesian product term, the

following relations are derived from the platform domain ontology:

$$\text{Fixed} = \texttt{PU} \times \texttt{Com} \times \texttt{Mem} \tag{4.5}$$

$$\text{Variable} = \texttt{PU}_{\chi} \times \texttt{Com}_{\chi} \times \texttt{Mem}_{\chi} \tag{4.6}$$

$$\text{Mixed} = \texttt{Com}_{\mathbb{M}} \tag{4.7}$$

Table 4.1. List of set symbols representing each realizable class.

Platform ontology class	Set symbol
Processing Unit	$\texttt{PU}$
Communication	$\texttt{Com}$
Memory	$\texttt{Mem}$
Configuration	$\texttt{Conf}$
Processing Unitχ	$\texttt{PU}_{\chi}$
Communicationχ	$\texttt{Com}_{\chi}$
Memoryχ	$\texttt{Mem}_{\chi}$
Configurationχ	$\texttt{Conf}_{\chi}$
Communication$_{\mathbb{M}}$	$\texttt{Com}_{\mathbb{M}}$

Entities from the three subsystems, *i.e.*, fixed, variable and mixed, as well as the realizable classes under the subsystems are described in details as follows:

1. Fixed: A subsystem composed of a general-purpose processor and peripherals. The hardware cannot be reconfigured, as in an FPGA, and the software needs to adapt to the hardware.

   (a) Processing Unit: The core of the general-purpose processor capable of executing sequences of instructions. A fixed subsystem can have more than one processing unit.

      i. Configuration: A functionality of a processing unit. Multiple configurations can be associated with the same processing unit. For the fixed subsystem, the configuration is a piece of software that defines what a programmable hardware does.

   (b) Communication: Any communication bus or signal that is part of the fixed subsystem.

   (c) Memory: general-purpose memory to store data and programs. The fixed subsystem can have more than one memory related to it.

2. Variable: A subsystem that can have its underlying hardware reconfigured to implement processing units, communication and memories. It also supports partial reconfiguration to enable runtime reconfiguration. The suffix χ is used to differentiate entities of the variable subsystem from the fixed subsystem.

(a) Processing Unitχ: Any processing node that transforms data, and it is part of the variable subsystem.

i. Configurationχ: A functionality of a processing unitχ. For the variable subsystem, it is a set of bitstreams that defines the functionality of a reconfigurable hardware area.

(b) Communicationχ: Any communication bus or signal that is part of the variable subsystem.

(c) Memoryχ: Any memory region that is part of the variable subsystem. Can be either configured in the reconfigurable area, or even a memory associated with the reconfigurable area.

3. Mixed: A subsystem composed by the intersection of fixed and variable subsystems. Entities belonging to the mixed class have part of its implementation in the fixed subsystem, and part in the variable subsystem.

(a) Communication$_{\mathbb{M}}$: Any communication bus or signal that is used to exchange data amongst fixed and variable subsystems, *i.e.* that connects subsystems together.

4.2 Platform Functional Blocks

The ontology presented in the previous section is general enough to represent different platform architectures that can be composed of a hard processor, microcontroller or graphics processing unit (GPU), representing the fixed subsystem, and an FPGA, representing the variable subsystem.

This section presents several platform architecture models, considering platforms with only a fixed subsystem, only a variable subsystem, and with both fixed and variable subsystems, *i.e.* heterogeneous. These platform architecture models are built up from the basic functional blocks *worker*, *steward*, *handler* and *memory region* defined next.

- Worker: A computational node, belonging to the set of processing units $\mathtt{PU} \cup \mathtt{PU}_{\chi}$, that performs data processing and can change its configuration, either by software or hardware reconfiguration. In hardware reconfiguration, it implements a function placeholder in the reconfigurable hardware, whereas in software reconfiguration, it is either a single function with multiple behaviors, or multiple functions with a single behavior. Workers that have only one configuration are called fixed workers.

- Handler: A computational node, belonging to the set of processing units $\mathtt{PU} \cup \mathtt{PU}_{\chi}$, whose task is to supervise and decide the current configuration of workers. It can be implemented

in either fixed or variable subsystem. The handler's configuration does not change.

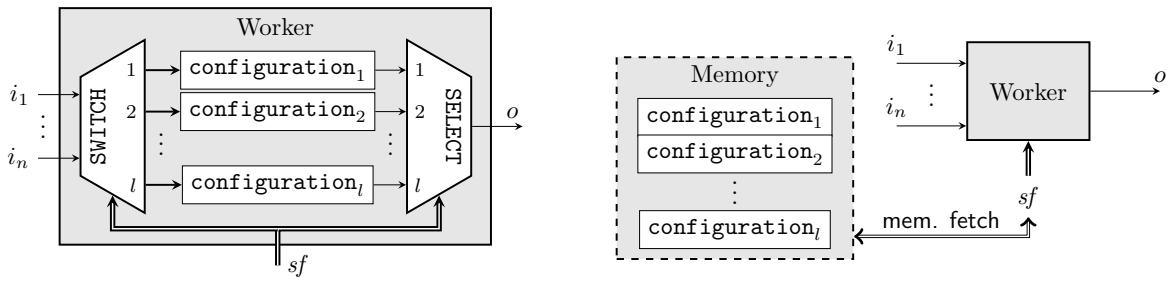
- Steward: A computational node, belonging to the set of processing units $\mathtt{PU} \cup \mathtt{PU}_\chi$, responsible for implementing the configuration replacement mechanism of a worker, in case of hardware reconfiguration. Such mechanism includes fetching the new configuration in a memory region responsible for storing the worker's configuration. Similar to the handler, the configuration of the steward does not change.

- Memory region: Any memory region, belonging to the set of memories $\mathtt{Mem} \cup \mathtt{Mem}_\chi$, presented in the platform either in the variable or fixed subsystem. For hardware reconfiguration, at least one memory region is necessary to store all configurations of the workers. Such memory region is called *configuration repository*.

The communication between functional blocks is done via *data* and *control buses*, which are elements from the communication set $\mathtt{Com} \cup \mathtt{Com}_\chi \cup \mathtt{Com}_\mathbb{M}$. The functional blocks are grouped into a *programmable device* (ProgDev) area, representing the fixed subsystem, and a *reconfigurable device* (ReconDev) area, representing the variable subsystem.

Two types of reconfiguration schemes are explored: *mode reconfiguration* (Fig. 4.2a) and *function reconfiguration* (Fig. 4.2b).

In the mode reconfiguration, all possible functional configurations (also denoted as modes) of a worker are already implemented inside it. To select the active configuration, a pair of *switch* and *select* blocks are used. This scheme has the advantage of having a low complexity and performing fast transitions between configurations, all at the cost of the worker's oversize, measured as the sum of the size of all configurations within it, as well as the switch and select blocks. Such scheme can be implemented entirely inside the ProgDev, or entirely inside the ReconDev.



(a) Mode reconfiguration. **(b)** Function reconfiguration.

Figure 4.2. Mode and function reconfiguration schemes considering a single worker with *l* possible configurations.

In the function reconfiguration, all configurations of a worker are stored in a memory called *configuration repository*. The worker acts as a function placeholder, having only the current configuration configured (*i.e.* implemented) in its area. As such, in function reconfiguration,

workers can only be implemented in the ReconDev area. To change between configurations, a memory fetch mechanism, implemented within a steward, is responsible for fetching the new configuration and sending it to the worker. The advantage of this scheme is the smaller size of the worker, measured as the size of its largest possible configuration, allowing a larger number of functionalities to be implemented in a smaller reconfiguration area. On the other hand, it possesses several disadvantages, including the necessity of an external memory to store all possible configurations, the complexity of both memory fetching and reconfiguration mechanisms, as well as the reconfiguration time required to change between different configurations.

Three possible platform architecture models for implementing runtime reconfiguration are enumerated as follows. One considers a fully fixed hardware, another considers a fully variable hardware, and the last one considers a heterogeneous hardware with fixed and variable subsystems.

1. *Fixed platform architecture model*: This platform model comprises only functional blocks from the fixed subsystem, meaning that, in this case, $\texttt{PU}_{\chi}$, $\texttt{Com}_{\chi}$, $\texttt{Mem}_{\chi}$ and $\texttt{Com}_{\mathbb{M}}$ are empty sets. Mode reconfiguration is the only possible scheme to perform runtime reconfiguration in this type of platform. As a consequence, stewards and configuration repositories are not required for this platform model. Figure 4.3 shows an example of such platform model comprising a single worker and a single handler.

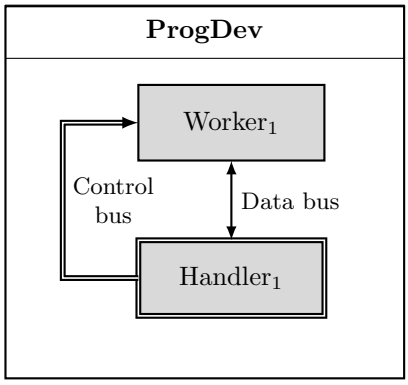


Figure 4.3. Platform composed of a fixed subsystem (ProgDev).

In this context, the same control bus defines the communication path used to set the configuration of the worker. For mode reconfiguration, the control bus does not send the configuration to the workers, but sends an integer value that selects on which mode the worker shall operate.

RTR is fully implemented in software in this type of platform model. The worker can be implemented either as a single function with a switch/case statement for selecting the current configuration, or as several different functions representing each configuration and using software function pointer technique to select them.

2. *Variable platform architecture model*: This platform model comprises only functional

blocks from the variable subsystem, meaning that, in this case, PU, Com, Mem and $\texttt{Com}_{\mathbb{M}}$ are empty sets. Both mode and function reconfiguration are possible schemes for this kind of platform.

To implement the mode reconfiguration, switch and select blocks must be implemented in hardware within each worker, as shown in Figure 4.2a. Similar to the fixed platform architecture model, mode reconfiguration does not require stewards and an implementation repository. Also, the reconfigurable platform does not need to support partial reconfiguration due to the fact that its hardware configuration will not change in runtime.

However, if the physical platform represented by the platform model supports partial reconfiguration, then function reconfiguration is a possible scheme to implement RTR. Figure 4.4 shows an example of such platform model for function reconfiguration with a single worker and handler.

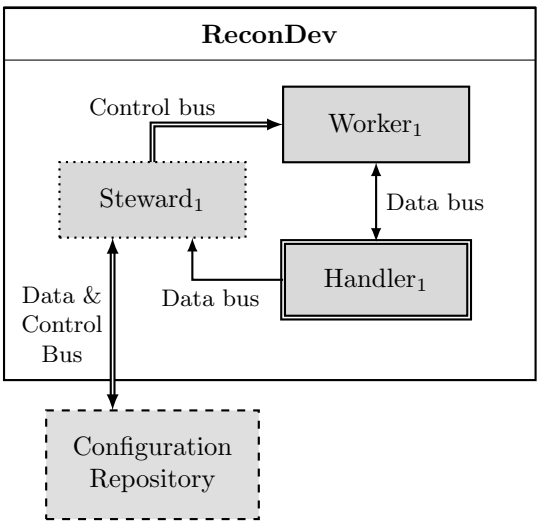


Figure 4.4. Platform composed of a variable subsystem (ReconDev).

The presence of a steward and a configuration repository is an unique characteristic of function reconfiguration. In this scheme, the handler decides which configuration is going to be used for each worker under its command, and when a reconfiguration is needed in one of such workers, the handler signals the steward with the desired configuration and target worker so that the steward may fetch the desired configuration in the configuration repository and perform the reconfiguration procedure in the target worker using the control bus. In this sense, only the target worker is reconfigured, hence the need for partial reconfiguration in the physical platform represented by the platform model.

Note that the configuration repository is depicted outside the ReconDev area. This occurs because the memory region that comprises the implementation repository does not need to lie inside the reconfigurable hardware, although it is possible to have the configuration repository defined inside the reconfigurable area. Given this, an external memory connected to ReconDev can be used to store the configurations of all workers.

3. *Heterogeneous platform architecture model*: Here, the name heterogeneous is referring to a platform composed of fixed and variable subsystems. Despite both mode and function reconfigurations are possible schemes to implement RTR in this platform model, the focus here will only be on function reconfiguration, considering the ReconDev supports partial reconfiguration.

   To take full advantage of both ProgDev and ReconDev, handlers and stewards, whose configurations are fixed, are implemented in the ProgDev, while workers are implemented in the ReconDev. Figure 4.5 shows an example of such architecture model.

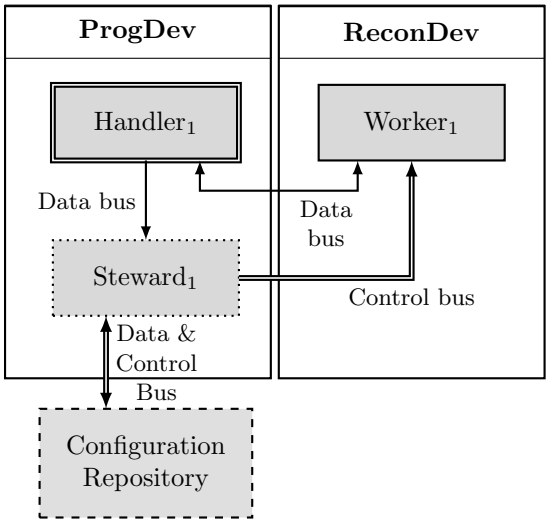


Figure 4.5. Heterogeneous platform composed of a fixed subsystem (ProgDev) and a variable subsystem (ReconDev).

   Note that data and control buses used to set communication paths between workers and handlers, and workers and stewards have one end in the ProgDev, and another in the ReconDev. As mentioned before, such entities are referred to as mixed entities, and belong to the set $\mathtt{Com}_{\mathbb{M}}$. These entities are only relevant in heterogeneous platform models, due to the presence of both fixed and variable subsystems. There can be as many entities in the $\mathtt{Com}_{\mathbb{M}}$ set as possible in the platform model, however a limitation on current physical platforms[1] constrains to a single control bus in $\mathtt{Com}_{\mathbb{M}}$.

To summarize, Table 4.2 shows the platform architecture types and the RTR schemes that are possible to implement in each one.

[1] Current physical platforms possess a single bus for writing in the configuration memory of the FPGA area.

Table 4.2. Summary of reconfiguration plataform model types and schemes.

Platform model type	RTR scheme
1. Fixed	mode
2. Variable	mode or function
3. Heterogeneous	mode or function

4.2.1 Example: Heterogeneous platform

This section presents an example of a heterogeneous platform architecture model that can be used to represent many commercial off-the-shelf (COTS) SoC devices with FPGAs, such as the Xilinx Zynq-7000 SoC (Xilinx, 2021) and the Intel-FPGA Cyclone V SoC (Intel, 2021). The platform model, as shown in Figure 4.6, is composed of a ProgDev and a ReconDev, following the heterogeneous platform model concept presented in the previous section.

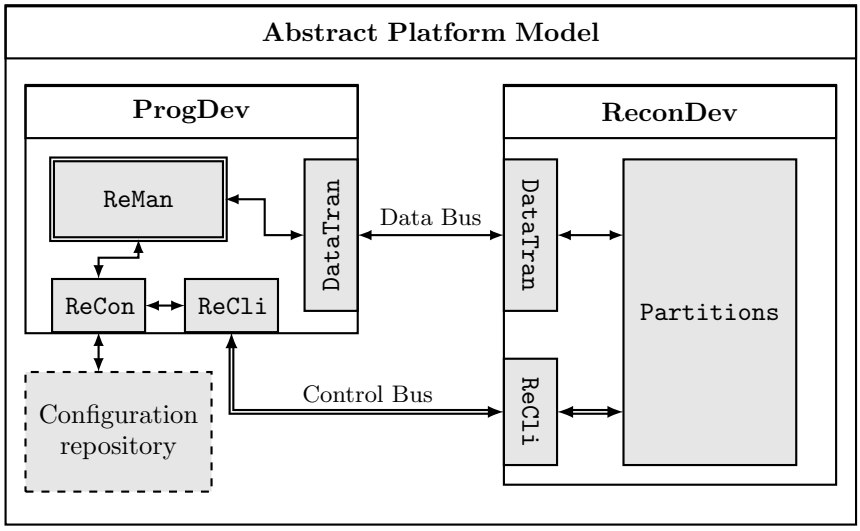


Figure 4.6. Heterogeneous platform model example.

The ProgDev is composed by four functional blocks enumerated in the sequence.

1. `ReMan`: Short for *reconfiguration manager*, it is a handler that chooses the functionalities, *i.e.* configurations, that are configured in the ReconDev. Although it is the only handler in this model, there is no limitation concerning the number of handlers a heterogeneous platform can have.

2. `ReCon`: The *reconfiguration controller* is the steward of this platform model. As so, it is responsible for fetching, in the configuration repository, the configuration selected by `ReMan` (in the form of bitstreams) and sending it to ReconDev via `ReCli`.

3. `ReCli`: The *reconfiguration client* is a mixed functional block belonging to $\texttt{Com}_{\mathbb{M}}$. It is responsible for implementing the communication protocols used to transfer bitstreams

between ProgDev and ReconDev. As a mixed entity, it has one part implemented in ProgDev, and another in ReconDev. For this platform model to be able to represent current physical heterogeneous platforms, the number of control buses, hence the number of reconfiguration clients, is constrained to the number of reconfiguration buses available in the physical platform modeled.

4. `DataTran`: The *data transceiver* is also a mixed functional block belonging to $\texttt{Com}_{\mathbb{M}}$. It is responsible for implementing the communication protocols used in the data exchange between ProgDev and ReconDev.

The ReconDev is composed of the `ReCli`, `DataTran` and `Partitions` area. The partitions area represents the area of the platform in which all workers, communication lanes among workers and possibly memories are implemented in. Therefore, entities implemented in the partitions area belong to the union of $\texttt{PU}_{\chi}$, $\texttt{Com}_{\chi}$ and $\texttt{Mem}_{\chi}$ sets. The number of workers, communication lanes and memories implemented inside the partitions area is defined by the application implemented, hence it is an information provided by the application model. Table 4.3 shows a summary of the functional blocks in the platform model. For simplicity, communication paths connecting `ReMan`, `Recon`, `ReCli`, `DataTran` and configuration repository are not shown in the table, but they belong to the `Com` platform ontology class.

Table 4.3. Summary of the functional blocks in the heterogeneous platform example.

Functional Block Name	Functional Block	Ontology Class
`ReMan`	Handler	`PU`
`ReCon`	Steward	`PU`
`ReCli`	Control Bus	$\texttt{Com}_{\mathbb{M}}$
`DataTran`	Data Bus	$\texttt{Com}_{\mathbb{M}}$
`Partitions`	Workers/memories/communication	$\texttt{PU}_{\chi} \cup \texttt{Mem}_{\chi} \cup \texttt{Com}_{\chi}$
Configuration repository	Memory region	`Mem`

5 MAPPING RULES

The previous chapters introduced two classification systems derived from domain ontologies for both application and platform models. Along with the classification systems, semantics for RTR using both synchronous and SADF MoCs were presented for the application model, as well as functional blocks applied to different platform architecture models and RTR schemes were presented for the platform model.

Up to this point, application and platform were treated separately, following the concept of orthogonalization of concerns (Keutzer et al., 2000). In this chapter the connection between application and platform models is defined in the form of *mapping rules*. The only requirement for the usage of the mapping rules defined in this chapter is to have both platform and application models classified according to the platform and application domain ontologies.

Mapping rules, represented by the letter $\mathfrak{m}$, are a set of functions that map elements from the application domain Λ to elements from the platform domain P, resulting in a feasible virtual implementation model I, as shown in Figure 5.1.

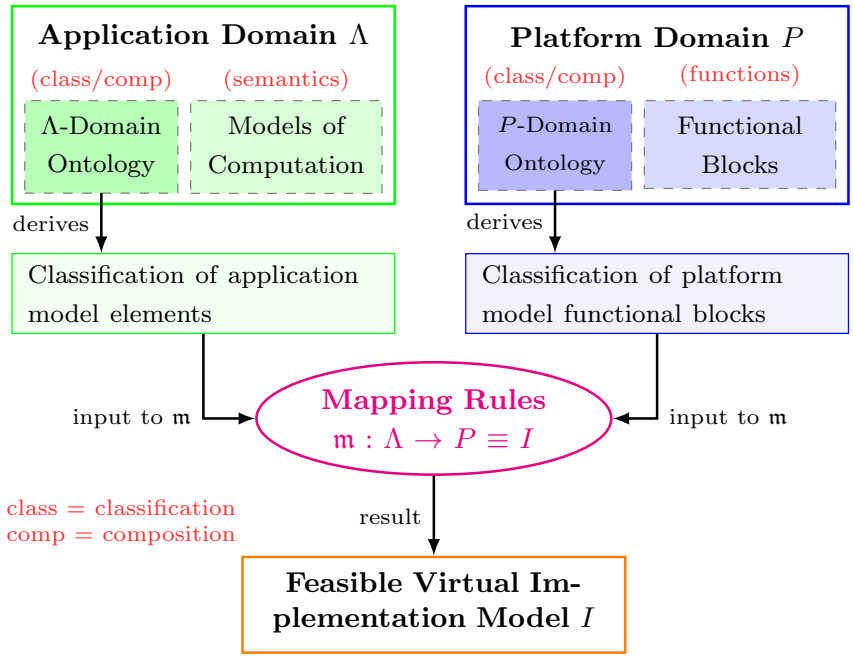


Figure 5.1. Mapping rules concept.

The application domain Λ is defined as the union of all realizable classes of the application domain ontology. On the other hand, the platform domain P is defined as the union of all sets of classes from the platform domain ontology.

In total, there are eight mapping rules that specify where entities from the application model are mapped to platform model's functional blocks. There is one rule for each application ontology realizable class, with exception of the `Value.Info` class, totalizing seven rules, and an 8^{th} rule for stewards and configuration repositories. The mapping rules are defined next.

1. `Procedure.Controller` rule:

   This mapping rule is about defining which functional blocks from the platform model are going to implement entities from the `Procedure.Controller` class. Such mapping rule is defined by the function $\mathfrak{m}_1$, whose type signature is given by:

   $$\mathfrak{m}_1 : \texttt{Procedure.Controller} \rightarrow \texttt{PU} \cup \texttt{PU}_\chi \tag{5.1}$$

   Entities from the `Procedure.Controller` class are responsible for outputting values from the `Value.Function` domain. Such functions define the behaviors of entities from the `Procedure.Executor.Variable` class.

   The mapping rule $\mathfrak{m}_1$ states the following:

   - Every entity $\alpha_i \in$ `Procedure.Controller` instantiates a handler functional block $\theta_i \in \texttt{PU} \cup \texttt{PU}_\chi$;
   - $\mathfrak{m}_1$ is reversible, *i.e.*, every α_i is mapped to a single θ_i, and no other α_j with $j \neq i$ is mapped to θ_i; and
   - Depending on the RTR scheme and the platform model, θ_i can belong to either `PU` or $\texttt{PU}_\chi$. Table 5.1 shows where handlers θ_i are located in the platform model depending on the RTR scheme and platform model type.

Table 5.1. Summary of the location of all handlers θ_i in the platform model for $\mathfrak{m}_1$.

	Mode	Function
Only Fixed	$\theta_i \in \texttt{PU}$	–
Only Variable	$\theta_i \in \texttt{PU}_\chi$	$\theta_i \in \texttt{PU}_\chi$
Heterogeneous	$\theta_i \in \texttt{PU} \cup \texttt{PU}_\chi$	$\theta_i \in \texttt{PU} \cup \texttt{PU}_\chi$

2. `Procedure.Executor.Variable` rule:

   This mapping rule, denoted by the function $\mathfrak{m}_2$, defines the functional blocks responsible for implementing entities from the `Procedure.Executor.Variable` class. $\mathfrak{m}_2$ has the following type signature:

   $$\mathfrak{m}_2 : \texttt{Procedure.Executor.Variable} \rightarrow \texttt{PU} \cup \texttt{PU}_\chi \tag{5.2}$$

   The mapping rule $\mathfrak{m}_2$ states the following:

- Every entity $\beta_i \in \texttt{Procedure.Executor.Variable}$ instantiates a worker functional block $\omega_i \in \texttt{PU} \cup \texttt{PU}_\chi$;
- $\mathfrak{m}_2$ is reversible, *i.e.*, every β_i is mapped to a single ω_i, and no other β_j with $j \neq i$ is mapped to ω_i;
- If the RTR scheme is function reconfiguration, then $\omega_i \in \texttt{PU}_\chi$ for all workers. If the RTR scheme is mode reconfiguration, then $\omega_i \in \texttt{PU} \cup \texttt{PU}_\chi$. This means that, for function reconfiguration, all workers ω_i must lie in the variable subsystem, whereas for mode reconfiguration, the workers are free to be either in the fixed or the variable subsystem, depending on the availability, *i.e.* if the platform is only fixed, only variable or heterogeneous. Table 5.2 summarizes where workers ω_i, implementing variable executors, are located in the platform model depending on the RTR scheme and platform model type.

Table 5.2. Summary of the location of workers ω_i implementing variable executors in the platform model for $\mathfrak{m}_2$.

	Mode	Function
Only Fixed	$\omega_i \in \texttt{PU}$	–
Only Variable	$\omega_i \in \texttt{PU}_\chi$	$\omega_i \in \texttt{PU}_\chi$
Heterogeneous	$\omega_i \in \texttt{PU} \cup \texttt{PU}_\chi$	$\omega_i \in \texttt{PU}_\chi$

3. `Procedure.Executor.Fixed` rule:

   This mapping rule defines the platform functional blocks responsible for the implementation of entities from the `Procedure.Executor.Fixed` application class. The mapping rule is defined by the function $\mathfrak{m}_3$, whose type signature is as follows:

$$\mathfrak{m}_3 : \texttt{Procedure.Executor.Fixed} \rightarrow \texttt{PU} \cup \texttt{PU}_\chi \tag{5.3}$$

   The mapping rule $\mathfrak{m}_3$ states that:

   - Every entity $\varepsilon_i \in \texttt{Procedure.Executor.Fixed}$ instantiates a worker functional block $\omega_i \in \texttt{PU} \cup \texttt{PU}_\chi$. Such worker has only one configuration, and it is not reconfigured at runtime, *i.e.* it is a fixed worker. For mode reconfiguration, there are no switch and select blocks in the worker;
   - $\mathfrak{m}_3$ is reversible, *i.e.*, every ε_i is mapped to a single ω_i, and no other ε_j with $j \neq i$ is mapped to ω_i. Since variable executors are also mapped to workers, any worker is either mapped to a fixed, or a variable executor, but not both; and
   - Workers implementing fixed executors can be implemented in either the fixed or variable subsystem, depending on the availability in the platform. Table 5.3 shows

where workers ω_i implementing fixed executors are located in the platform model depending on the RTR scheme and platform model type.

Table 5.3. Summary of the location of workers ω_i implementing fixed executors in the platform model for $\mathfrak{m}_3$.

	Mode	Function
Only Fixed	$\omega_i \in \mathtt{PU}$	–
Only Variable	$\omega_i \in \mathtt{PU}_\chi$	$\omega_i \in \mathtt{PU}_\chi$
Heterogeneous	$\omega_i \in \mathtt{PU} \cup \mathtt{PU}_\chi$	$\omega_i \in \mathtt{PU} \cup \mathtt{PU}_\chi$

4. `Path.Control` rule:

   This mapping rule defines the platform functional elements that are responsible for implementing entities from the `Path.Control` application class. The mapping rule is defined by the function $\mathfrak{m}_4$, whose type signature is as follows:

$$\mathfrak{m}_4 : \mathtt{Path.Control} \rightarrow \mathtt{Com} \cup \mathtt{Com}_\chi \cup \mathtt{Com}_\mathbb{M} \tag{5.4}$$

   The mapping rule $\mathfrak{m}_4$ states that:

   - Every entity $\hat{\sigma}_i \in \mathtt{Path.Control}$ is associated with a single control bus $\hat{\rho}_j \in \mathtt{Com} \cup \mathtt{Com}_\chi \cup \mathtt{Com}_\mathbb{M}$;
   - The number of control buses varies depending on the RTR scheme, which also affects the type of functional blocks connected by the buses, according to the following rules:
     - If the RTR scheme is function reconfiguration, then a control bus $\hat{\rho}_j$ connects a single steward to possibly multiple workers.
     - If the RTR scheme is mode reconfiguration, then a control bus $\hat{\rho}_j$ connects a handler to a single worker.
   - Depending on the RTR scheme, $\mathfrak{m}_4$ can be reversible or not. The following rules define when $\mathfrak{m}_4$ is reversible or not:
     - If the RTR scheme is function reconfiguration, then $\mathfrak{m}_4$ is not reversible.
     - If the RTR scheme is mode reconfiguration, then $\mathfrak{m}_4$ is reversible.

     The rationale behind these rules relies on the fact that function reconfiguration uses the platform bus responsible for transferring bitstreams to the reconfigurable area. Due to limitations on the number of such buses (normally, there is only one available for the entire platform), the same bus must be used to connect multiple workers. Note that such limitation is related only to the physical platform, however, it impacts in the existence of an inverse function for $\mathfrak{m}_4$.

- A control bus $\hat{\rho}_j$ may belong to different platform ontology classes, *i.e.* $\texttt{Com}$, $\texttt{Com}_\chi$ and $\texttt{Com}_\mathbb{M}$, depending on the RTR scheme and platform model type.

Table 5.4 summarizes the mapping rule $\mathfrak{m}_4$ in terms of the functional blocks connected by control buses, existence of a reverse function and the platform ontology class associated with the control buses.

Table 5.4. Summary of the mapping rule $\mathfrak{m}_4$ in terms of the functional blocks connected by control buses, existence of reverse function and the platform ontology class associated with the control buses.

	Mode	Function
Only Fixed	handler to worker $\mathfrak{m}_4$ is reversible $\hat{\rho}_j \in \texttt{Com}$	–
Only Variable	handler to worker $\mathfrak{m}_4$ is reversible $\hat{\rho}_j \in \texttt{Com}_\chi$	steward to workers $\mathfrak{m}_4$ is not reversible $\hat{\rho}_j \in \texttt{Com}_\chi$
Heterogeneous	handler to worker $\mathfrak{m}_4$ is reversible $\hat{\rho}_j \in \texttt{Com} \cup \texttt{Com}_\chi$	steward to workers $\mathfrak{m}_4$ is not reversible $\hat{\rho}_j \in \texttt{Com}_\mathbb{M}$

5. `Path.Data.Homogeneous` rule:

   This mapping rule defines the functional blocks responsible for implementing entities from the `Path.Data.Homogeneous` application class. The mapping rule is defined by the function $\mathfrak{m}_5$, whose type signature is as follows:

$$\mathfrak{m}_5 : \texttt{Path.Data.Homogeneous} \rightarrow \texttt{Com} \cup \texttt{Com}_\chi \tag{5.5}$$

   The mapping rule $\mathfrak{m}_5$ states that:

   - For every homogeneous data path σ_i in the application model, a unique data bus ρ_i is created to implement it;
   - $\mathfrak{m}_5$ is reversible, meaning that every data bus in the platform maps to a single homogeneous data path; and
   - Depending on the RTR scheme and platform model type, a given data bus ρ_i may belong to either $\texttt{Com}$ or $\texttt{Com}_\chi$. Table 5.5 presents a summary of the possible platform ontology classes in which ρ_i may belong to depending on the RTR scheme and platform model type.

Table 5.5. Summary of the location of data buses ρ_i implementing homogeneous data paths in the platform model for $\mathfrak{m}_5$.

	Mode	Function
Only Fixed	$\rho_i \in \texttt{Com}$	–
Only Variable	$\rho_i \in \texttt{Com}_\chi$	$\rho_i \in \texttt{Com}_\chi$
Heterogeneous	$\rho_i \in \texttt{Com} \cup \texttt{Com}_\chi$	$\rho_i \in \texttt{Com} \cup \texttt{Com}_\chi$

6. `Path.Data.Hybrid` rule:

   This mapping rule defines the functional blocks responsible for implementing entities from the `Path.Data.Hybrid` application class. The mapping rule is defined by the function $\mathfrak{m}_6$, whose type signature is as follows:

$$\mathfrak{m}_6 : \texttt{Path.Data.Hybrid} \rightarrow \texttt{Com} \cup \texttt{Com}_\chi \cup \texttt{Com}_\mathbb{M} \tag{5.6}$$

   The mapping rule $\mathfrak{m}_6$ states that:

   - Every hybrid data path σ_i in the application model is associated with a data bus ρ_j;
   - Depending on the RTR scheme and platform type, $\mathfrak{m}_6$ can be reversible or not. The following rules define when $\mathfrak{m}_6$ is reversible or not:
     - If the RTR scheme is function reconfiguration and the platform type is heterogeneous, then $\mathfrak{m}_6$ is not reversible. A single data bus implements multiple hybrid data paths.
     - For every other combination, $\mathfrak{m}_6$ is reversible.
   - Depending on the RTR scheme and platform model type, a given data bus ρ_i may belong to either $\texttt{Com}$, $\texttt{Com}_\chi$ or $\texttt{Com}_\mathbb{M}$. Table 5.6 presents a summary of the possible platform ontology classes in which ρ_j may belong to depending on the RTR scheme and platform model type.

Table 5.6. Summary of the location of data buses ρ_j implementing hybrid data paths in the platform model for $\mathfrak{m}_6$.

	Mode	Function
Only Fixed	$\rho_j \in \texttt{Com}$	–
Only Variable	$\rho_j \in \texttt{Com}_\chi$	$\rho_j \in \texttt{Com}_\chi$
Heterogeneous	$\rho_j \in \texttt{Com} \cup \texttt{Com}_\chi$	$\rho_j \in \texttt{Com}_\mathbb{M}$

7. `Value.Function` rule:

The mapping rule $\mathfrak{m}_7$ has the objective of mapping functions, representing behaviors of processes in the application model, to configurations for the platform model. The type signature of $\mathfrak{m}_7$ is as follows:

$$\mathfrak{m}_7 : \texttt{Value.Function} \rightarrow \texttt{Conf} \cup \texttt{Conf}_\chi \tag{5.7}$$

The mapping rule $\mathfrak{m}_7$ states that:

- Every function $f_i \in \texttt{Value.Function}$ is mapped to a configuration $\lambda_i \in \texttt{Conf} \cup \texttt{Conf}_\chi$;
- $\mathfrak{m}_7$ is reversible, meaning that each function f_i is mapped to a single configuration λ_i and vice-versa;
- Variable workers, *i.e.* workers mapped to variable executors using $\mathfrak{m}_2$, have multiple configurations associated to it. If the RTR scheme is mode reconfiguration, then such workers have all configurations already implemented on it, and a switch/select mechanism is used to select the active configuration. For function reconfiguration, such worker has a single configuration implemented on it at a time, meaning that if a new configuration is required, it overwrites the previous worker configuration; and
- Fixed workers, *i.e.* workers mapped to fixed executors using $\mathfrak{m}_3$, handlers and stewards have a single configuration which does not change in runtime.

8. Function reconfiguration exclusive rule:

   The 8^{th} rule is only applicable for function reconfiguration. It defines constraints on the number of stewards and configuration repositories. It is also used to map configurations to the configuration repositories storing them.

   Rule $\mathfrak{m}_8$, states the following:

   - Each handler has its own unique steward;
   - There is at least one configuration repository in the platform model;
   - Each steward can communicate with multiple configuration repositories; and
   - Depending on the platform model type, a configuration repository μ_i can belong to either `Mem` or $\texttt{Mem}_\chi$. Table 5.7 summarizes the platform ontology class each configuration repository belongs to, as a function of the platform model type.

Table 5.7. Summary of the location of memory regions μ_i acting as configuration repositories in the platform model for $\mathfrak{m}_8$.

	Mode	Function
Only Fixed	–	–
Only Variable	–	$\mu_i \in \texttt{Mem}_\chi$
Heterogeneous	–	$\mu_i \in \texttt{Mem}$

5.1 Mapping Rules Example

This section shows a comprehensive example involving the mapping rules presented in the previous section. The application model used for this example is the SADF RISC processor presented in Section 3.2.2, and the platform model is the heterogeneous platform example of Section 4.2.1. Both models are already labeled according to application and platform domain ontologies, fulfilling the pre-requisite to apply the mapping rules.

The first step is to map each element of the application model to a functional block from the platform model using rules $\mathfrak{m}_{1...6}$. The result is shown in Table 5.8.

Table 5.8. Summary of the mapping rules applied to each element from the application model.

application element	mapping rule	functional block	platform model location
IF	$\mathfrak{m}_2$	worker (variable)	`Partitions`
EXE	$\mathfrak{m}_2$	worker (variable)	`Partitions`
DEC	$\mathfrak{m}_1$	handler	`ReMan`
σ_{br}	$\mathfrak{m}_5$	data bus	`Partitions`
σ_{arg}	$\mathfrak{m}_5$	data bus	`Partitions`
σ_{op}	$\mathfrak{m}_6$	data bus	`DataTran`
$\hat{\sigma}_{IF}$	$\mathfrak{m}_4$	control bus	`ReCli`
$\hat{\sigma}_{EXE}$	$\mathfrak{m}_4$	control bus	`ReCli`

Every function defining the behaviors of each procedure in the application model is converted into a configuration using rule $\mathfrak{m}_7$. In total, there are 23 functions in the application model, 2 for IF, 20 for EXE and 1 for DEC. These 23 functions are converted into 23 configurations. Since DEC is a controller procedure, the single function associated with it is mapped to a configuration associated with the handler `ReMan`. Such configuration is implemented in software due to the fact that `ReMan` is part of the fixed subsystem. For the remaining 22 configurations, rule $\mathfrak{m}_8$ is used to define which configuration repository each one of these configurations will be stored as bitstreams. For the chosen platform model, all configurations are stored in the only configuration repository available.

Figure 5.2 illustrates in a diagram the SADF RISC processor application model mapped to the platform model.

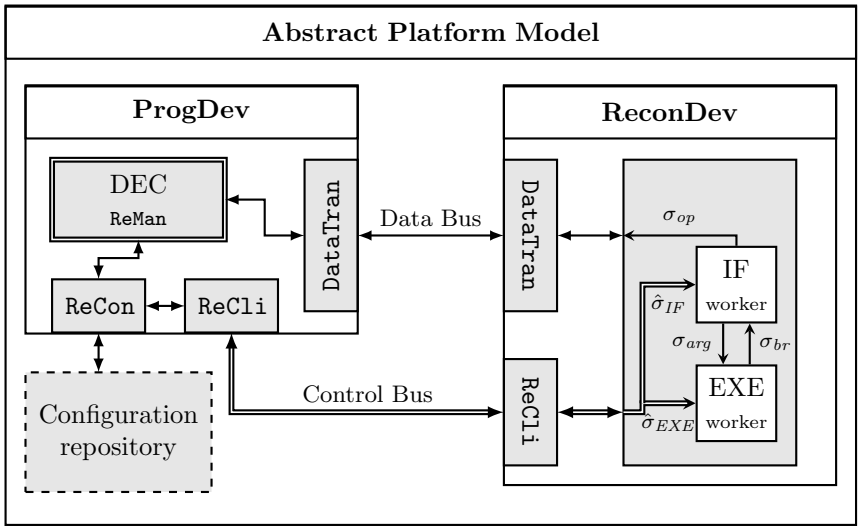


Figure 5.2. SADF RISC processor application mapped into platform model.

A few remarks are pointed out for this example:

- A single control bus (`ReCli`) is available in the platform. As a consequence, all variable workers are connected to this control bus. Also, there can be only one steward and, consequently, one configuration repository in the platform model. Many commercial off-the-shelf platforms have a single bus used for transporting bitstreams for reconfiguration purposes, hence, a single control bus. The platform model example used represents one of such platforms.

- The platform example represents a generic heterogeneous platform and considers the `Partitions` area as the entire variable subsystem. This means that, after the application of the mapping rules, the designer will have to perform another step which consists in defining where each worker and data bus mapped to the `Partitions` area is located. This step is not shown in the example because the mapping rules do not constraint the location of the workers and data buses inside the `Partitions` area, instead they constraint the functional blocks to the `Partitions` area. Design space exploration techniques can be used to provide an optimal location for these functional blocks inside the `Partitions` area, however such techniques are out of scope for this thesis.

6 CONCLUSION

This chapter concludes this thesis by wrapping up the concepts presented here as the foundations of a design flow for runtime reconfigurable systems. A summary of the publications related to the present research is also shown, as well as some directions for future works.

To understand the relevance of the work presented here, the importance of such concept in future embedded systems must be taken into consideration. RTR can provide several benefits for embedded systems, such as:

- Reduction of the logical area of a system (defined as the maximum area occupied by the logic circuitry of a system). Different tasks that are never executed simultaneously need not to have their logic circuitry implemented at the same time. Runtime reconfiguration allows for logic circuitry to be implemented in runtime only when needed, increasing usability and, as a consequence, reducing the cost of the system related to the hardware platform (platforms with smaller logic areas are generally cheaper). The concept of executing larger programs in smaller systems has been used in software for decades with the overlaying technique, and now it can be applied to hardware with runtime reconfiguration.

- Increased fault tolerance for safety-critical systems. In the event of a hardware failure, critical functionalities can be transferred to a new hardware location. Moreover, such new hardware location could be initially configured with less critical functionalities, and then reconfigured with critical functionalities when needed, eliminating the system's need to have dedicated unused spare hardware.

Up until today, the design of RTR systems has mostly been done via *ad hoc* procedures for small scale and experimental applications. Formal design methodologies are not used in the development of RTR systems in a systematic way. This thesis comprises the first steps towards a formal design methodology for RTR systems by providing the following concepts:

- Modeling RTR applications with formal models of computation, allowing simulation and formal property verification.

  In this thesis, RTR applications are modeled with either the synchronous or the SADF models of computation. The synchronous model is best suited for RTR applications in which time is an important factor, such as real-time applications. As such, the synchronous runtime reconfigurable processor presented in Section 3.2.1 takes reconfiguration time into consideration in its operational semantics. On the other hand, the SADF

model is best suited for RTR applications in which time is not important, such as streaming applications.

In this matter, it is important to emphasize two aspects:

1. Modeling RTR applications is not limited to synchronous and SADF models, as other models of computation can also be used; and
2. The design of embedded systems often includes the interaction of the designed system with the environment, as such interaction is relevant to the correct behavior of the overall system (embedded system + environment). Therefore, it may be necessary to include the model of the environment in the high level design phase. In the proposed design flow, the application model refers only to the model of the designed system (embedded system), *i.e.* the model of environment is not part of the application model.

- An application domain ontology for classification of RTR application model entities.

  The application domain ontology is used to group each entity in an application model into classes based on their common characteristics. Every class defined in the application domain ontology is derived from one of the three top classes: procedure, path and value. Such domain ontology is general enough to be used to classify entities of RTR applications modeled with many different models of computation, *i.e.* it is not limited to synchronous and SADF models.

- Modeling the reconfigurable platform with functional blocks.

  Specific functional blocks are defined to implement runtime reconfiguration functionalities. The functional block handler acts as the decision making mechanism, selecting the configuration of each worker. The steward is the functional block responsible for fetching configurations selected by the handler and sending them to each worker via control buses. Memory regions are various memory blocks that store data and configurations, in the case of the configuration repository. Control and data buses are also defined to set communication paths between functional blocks.

- A platform domain ontology for classification of reconfigurable platform functional blocks.

  The platform domain ontology follows the same principles as the application domain ontology, which is grouping platform functional blocks into classes based on their common characteristics. The ontology follows the concept of the *fixed plus variable* computer presented by Estrin (1960), and the classes defined in the ontology are derived from the top classes fixed, variable and mixed (mixed being the intersection of fixed and variable). The platform domain ontology is general enough to represent many different commercial off-the-shelf platforms.

- A set of mapping rules to map an application model into a platform model using as basis the classification mechanism provided by both application and platform domain ontologies.

  In order to combine application and platform models, a set of mapping rules is defined to map each application model entity to a platform functional block. The mapping rules are defined on top of both application and platform domain ontologies, in order to abstract away the specific model of computation and platform model used for applications and platforms.

6.1 Publications

Papers published by the author of this thesis as both leading and co-author are listed as follows:

- As leading author:
  - Modeling and Simulation of Dynamic Applications Using Scenario-Aware Dataflow (Bonna et al., 2019b).

    Since its conception, SADF has been treated as a structural model, allowing many properties and analysis techniques to be developed. Although implicit in the structural model, its operational semantics, what allows a model of computation to be executed, had not yet been defined. The paper formalizes the operational semantics for SADF by introducing a functional SADF model, and presents its implementation in ForSyDe, allowing simulation of SADF applications. Such operational semantics is used through Section 3.2.2 of this thesis.
  - Triple Modular Redundancy based on Runtime Reconfiguration and Formal Models of Computation (Bonna et al., 2019a).

    The paper presents a conceptual architecture for safety-critical systems using runtime reconfiguration and a triple modular redundancy scheme as the fault detection mechanism. Such architecture was used as example in Section 3.2.1 of this thesis.
- As co-author:
  - Analysis and Identification of Possible Automation Approaches for Embedded Systems Design Flows (Horita et al., 2020).

    The paper shows a method of identification of possible automation of steps in a design flow. The goal of the method is to assist in low-level implementation of automatic code generation.

– Analysis and Comparison of Frameworks Supporting Formal System Development based on Models of Computation (Horita et al., 2019a).

  The paper shows a comparison between ForSyDe and Ptolemy II, two frameworks for model-based development supporting models of computation, considering multiple aspects.

– Lempel-Ziv-Markov Chain Algorithm Modeling using Models of Computation and ForSyDe (Horita et al., 2019b).

  The paper shows a case study in which a famous data compression algorithm is modeled using SDF and ForSyDe.

6.2 Future Works

The scope of this thesis ends with the virtual implementation model. However, there are further steps to be developed in order to achieve a complete formal design flow for RTR systems, such as the one represented by Figure 6.1. The development of the following steps are proposed as future works:

- A design space exploration to select the optimal location to implement the functional blocks in the reconfigurable area using the virtual implementation model as constraint. As inputs to this design space exploration step, an optimization criteria must be provided by the designer, as well as other possible constraints, *e.g.*, performance and power consumption.

- Automatic code generation for both software and hardware using an implementation model complying with the virtual implementation model, *i.e.* a feasible implementation model considering the virtual implementation model as constraint. Such feasible implementation model could be the output of the design space exploration step previously mentioned. The work presented by Aronsson and Sheeran (2017) could be used as inspiration for the development of this step in the design process.

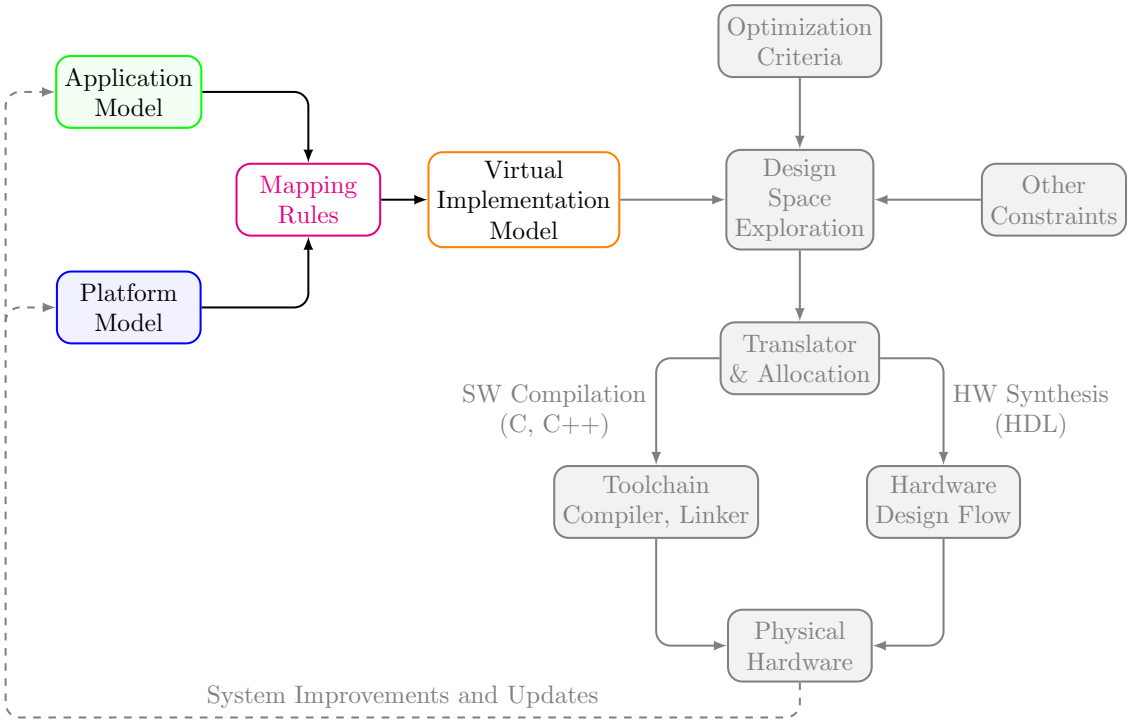


Figure 6.1. Complete design flow for RTR systems. Nodes in gray are out of scope of this thesis, and thus, considered future work.

Bibliography

Aronsson, M. and Sheeran, M. (2017). Hardware software co-design in haskell. In *Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell*. ACM.

Baaij, C., Kooijman, M., Kuper, J., Boeijink, A., and Gerards, M. (2010). CλaSH: Structural descriptions of synchronous hardware using haskell. In *2010 13th Euromicro Conference on Digital System Design: Architectures, Methods and Tools*. IEEE.

Backus, J. (1978). Can programming be liberated from the von neumann style?: a functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641.

Bieber, P., Boniol, F., Boyer, M., Noulard, E., and Pagetti, C. (2012). New Challenges for Future Avionic Architectures. *AerospaceLab*, (4):p. 1–10.

Bonna, R., Loubach, D. S., Sander, I., and Söderquist, I. (2019a). Triple modular redundancy based on runtime reconfiguration and formal models of computation. In *Proceedings of the 10th Aerospace Technology Congress, October 8-9, 2019, Stockholm, Sweden*. Linköping University Electronic Press.

Bonna, R., Loubach, D. S., Ungureanu, G., and Sander, I. (2019b). Modeling and simulation of dynamic applications using scenario-aware dataflow. *ACM Transactions on Design Automation of Electronic Systems*.

Bozzano, M. and Villafiorita, A. (2011). *Design and Safety Assessment of Critical Systems*. Auerbach Publications, Boca Raton, FL.

Chattopadhyay, A. (2013). Ingredients of adaptability: A survey of reconfigurable processors. *VLSI Design*, pages 1–18.

Dorflinger, A., Fiethe, B., Michalik, H., Fekete, S. P., Keldenich, P., and Scheffer, C. (2017). Resource-efficient dynamic partial reconfiguration on FPGAs for space instruments. In *2017 NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*. IEEE.

Edwards, S., Lavagno, L., Lee, E., and Sangiovanni-Vincentelli, A. (1997). Design of embedded systems: formal models, validation, and synthesis. *Proceedings of the IEEE*, 85(3):366–390.

Estrin, G. (1960). Organization of computer systems: the fixed plus variable structure computer. In *Papers presented at the May 3-5, 1960, western joint IRE-AIEE-ACM computer conference on - IRE-AIEE-ACM 60 (Western)*. ACM Press.

ForSyDe (2018). Forsyde-shallow. https://github.com/forsyde/forsyde-shallow.

Furht, B., Grostick, D., Gluch, D., Rabbat, G., Parker, J., and McRoberts, M. (1991). *Real-Time UNIX® Systems*. Springer.

Horita, A. Y., Bonna, R., and Loubach, D. S. (2019a). Analysis and comparison of frameworks supporting formal system development based on models of computation. In Latifi, S., editor, *16th International Conference on Information Technology-New Generations (ITNG 2019)*, pages 161–167, Cham. Springer International Publishing.

Horita, A. Y., Bonna, R., Loubach, D. S., Sander, I., and Söderquist, I. (2019b). Lempel-ziv-markov chain algorithm modeling using models of computation and ForSyDe. In *Proceedings of the 10th Aerospace Technology Congress, October 8-9, 2019, Stockholm, Sweden*. Linköping University Electronic Press.

Horita, A. Y., Loubach, D. S., and Bonna, R. (2020). Analysis and identification of possible automation approaches for embedded systems design flows. *Information*, 11(2):120.

Intel (2021). Cyclone v soc. https://www.intel.com/content/www/us/en/products/programmable/soc/cyclone-v.html.

Keutzer, K., Newton, A. R., Rabaey, J. M., and Sangiovanni-Vincentelli, A. (2000). System-level design: orthogonalization of concerns and platform-based design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 19(12):1523–1543.

Lee, E. and Sangiovanni-Vincentelli, A. (1998). A framework for comparing models of computation. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 17(12):1217–1229.

Lee, E. A. (2015). The past, present and future of cyber-physical systems: A focus on models. *Sensors*, 15(3):4837–4869.

Lee, E. A. and Messerschmitt, D. G. (1987a). Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Transactions on computers*, 100(1):24–35.

Lee, E. A. and Messerschmitt, D. G. (1987b). Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245.

Lee, E. A. and Seshia, S. A. (2015). *Introduction to Embedded Systems - A Cyber-Physical Systems Approach*. LeeSeshia.org, second edition.

Loubach, D. S. (2016). A runtime reconfiguration design targeting avionics systems. In *2016 IEEE/AIAA 35th Digital Avionics Systems Conference (DASC)*, pages 1–8. IEEE.

Michaelson, G. (2011). *An Introduction to Functional Programming Through Lambda Calculus*. DOVER PUBN INC.

Nguyen, M., Tamburo, R., Narasimhan, S., and Hoe, J. C. (2019). Quantifying the benefits of dynamic partial reconfiguration for embedded vision applications. In *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE.

Ptolemaeus, C., editor (2014). *System design, modeling, and simulation using Ptolemy II*. Ptolemy.org.

Radojevic, I. and Salcic, Z. (2011). *Embedded Systems Design Based on Formal Models of Computation*. Springer.

Sadeghi, A., Lighvan, M. Z., and Prinetto, P. (2020). Automatic and simultaneous floorplanning and placement in field-programmable gate arrays with dynamic partial reconfiguration based on genetic algorithm. *Canadian Journal of Electrical and Computer Engineering*, 43(4):224–234.

Sander, I. (2003). *System Modeling and Design Refinement in ForSyDe*. PhD thesis, Royal Institute of Technology KTH.

Sander, I., Jantsch, A., and Attarzadeh-Niaki, S.-H. (2017). ForSyDe: System design using a functional language and models of computation. In Ha, S. and Teich, J., editors, *Handbook of Hardware/Software Codesign*, pages 99–140. Springer Netherlands.

Sifakis, J. (2015). System design automation: Challenges and limitations. *Proceedings of the IEEE*, 103(11):2093–2103.

Stuijk, S., Geilen, M., and Basten, T. (2006). SDF^3: SDF for free. In *Sixth International Conference on Application of Concurrency to System Design (ACSD'06)*. IEEE.

Stuijk, S., Geilen, M., Theelen, B., and Basten, T. (2011). Scenario-aware dataflow: Modeling, analysis and implementation of dynamic applications. In *Embedded Computer Systems (SAMOS), 2011 International Conference on*, pages 404–411.

Teich, J. (2012). Hardware/software codesign: The past, the present, and predicting the future. *Proceedings of the IEEE*, 100(Special Centennial Issue):1411–1430.

Theelen, B., Geilen, M., Basten, T., Voeten, J., Gheorghita, S., and Stuijk, S. (2006). A scenario-aware data flow model for combined long-run average and worst-case performance analysis. In *Formal Methods and Models for Co-Design, 2006. MEMOCODE '06. Proceedings. Fourth ACM and IEEE International Conference on*, pages 185–194.

Thomas, N., Felder, A., and Bobda, C. (2015). Adaptive controller using runtime partial hardware reconfiguration for unmanned aerial vehicles (uavs). In *2015 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pages 1–7.

Tripakis, S. (2016). Compositionality in the science of system design. *Proceedings of the IEEE*, 104(5):960–972.

Ungureanu, G., Medeiros, J. E. G. D., Sundström, T., Söderquist, I., Åhlander, A., and Sander, I. (2021). ForSyDe-atom. *ACM Transactions on Embedded Computing Systems*, 20(2):1–27.

Xilinx (2021). Zynq-7000 soc. `https://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html`.

Appendix A SOURCE CODE

A.1 Synchronous Triple Modular Redundancy

```
1  ------------------------------------------------------------------------------
2  -- |
3  -- Module      :  TMR (Triple Modular Redundancy)
4  -- Copyright   :  (c) Ricardo Bonna
5  -- License     :  not available
6  --
7  -- Maintainer  :  ricardobonna@gmail.com
8  -- Stability   :  experimental
9  -- Portability :  portable
10 --
11 -- This is a prototype triple modular redundancy using runtime reconfiguration
12 --
13 ------------------------------------------------------------------------------
14
15 module TMR ( rtrp, voter, ctrlDev, rtrp1, tmrPN ) where
16
17 import ForSyDe.Shallow
18
19
20 -- Auxiliary functions
21 fst' :: (a,b,c) -> a
22 fst' (a,_,_) = a
23
24 snd' :: (a,b,c) -> b
25 snd' (_,b,_) = b
26
27 trd' :: (a,b,c) -> c
28 trd' (_,_,c) = c
29
30 tupleSel :: (a,a,a,a,a) -> Int -> a
31 tupleSel (a,_,_,_,_) 1 = a
32 tupleSel (_,a,_,_,_) 2 = a
33 tupleSel (_,_,a,_,_) 3 = a
34 tupleSel (_,_,_,a,_) 4 = a
35 tupleSel (_,_,_,_,a) 5 = a
36 tupleSel (_,_,_,_,_) _ = error "tupleSel: Outside tuple range"
37
38
39 -- Process Functionalities
40
41 rtrpFunc :: (Eq x) => AbstExt (AbstExt x -> AbstExt s_in -> AbstExt x, AbstExt x ->
       AbstExt s_in -> AbstExt y, Int) -> AbstExt s_in
42           -> AbstExt x -> (AbstExt x -> AbstExt s_in -> AbstExt x, AbstExt x ->
       AbstExt s_in -> AbstExt y, Int, AbstExt x)
43           -> ((AbstExt y, AbstExt x), (AbstExt x -> AbstExt s_in -> AbstExt x,
       AbstExt x -> AbstExt s_in -> AbstExt y, Int, AbstExt x))
44 rtrpFunc ctk s_ink x'k (fk,gk,mk,xk) = ((yk, xk1), (fk1,gk1,mk1,xk1))
45   where yk = ykFunc ctk s_ink x'k gk mk xk
46         (fk1,gk1,mk1,xk1) = nStateF ctk s_ink x'k (fk,gk,mk,xk)
47
48
```

```
49  nStateF :: (Eq x) => AbstExt (AbstExt x -> AbstExt s_in -> AbstExt x, AbstExt x ->
        AbstExt s_in -> AbstExt y, Int)
50          -> AbstExt s_in
51          -> AbstExt x
52          -> (AbstExt x -> AbstExt s_in -> AbstExt x, AbstExt x -> AbstExt s_in ->
        AbstExt y, Int, AbstExt x)
53          -> (AbstExt x -> AbstExt s_in -> AbstExt x, AbstExt x -> AbstExt s_in ->
        AbstExt y, Int, AbstExt x)
54  nStateF Abst s_ink x'k (fk,gk,mk,xk)
55      | mk > 0 = (fk, gk, mk - 1, Abst)
56      | xk == Abst = (fk, gk, 0, fk x'k s_ink)
57      | otherwise = (fk, gk, 0, fk xk s_ink)
58  nStateF (Prst ctk) _ _ _ = (fst' ctk, snd' ctk, trd' ctk - 1, Abst)
59
60
61  ykFunc :: (Eq x) => AbstExt (AbstExt x -> AbstExt s_in -> AbstExt x, AbstExt x ->
        AbstExt s_in -> AbstExt y, Int)
62         -> AbstExt s_in
63         -> AbstExt x
64         -> (AbstExt x -> AbstExt s_in -> AbstExt y)
65         -> Int
66         -> AbstExt x
67         -> AbstExt y
68  ykFunc Abst s_ink x'k gk mk xk
69      | mk > 0 = Abst
70      | xk == Abst = gk x'k s_ink
71      | otherwise = gk xk s_ink
72  ykFunc _ _ _ _ _ _ = Abst
73
74
75  voterFunc :: (Eq y) => (Int, Int, Int)
76            -> ((AbstExt y, AbstExt x), (AbstExt y, AbstExt x), (AbstExt y, AbstExt x)
        , (AbstExt y, AbstExt x), (AbstExt y, AbstExt x))
77            -> (AbstExt y, AbstExt x, AbstExt Int)
78  voterFunc (a,b,c) s_in
79      | ya == yb && ya == yc = (ya, xa, Abst)
80      | ya /= yb && yb == yc = (yb, xb, Prst a)
81      | yb /= ya && ya == yc = (ya, xa, Prst b)
82      | ya /= yc && yb == ya = (ya, xa, Prst c)
83      | otherwise = error "voterFunc: Three inputs are different"
84      where (ya, xa) = tupleSel s_in a
85            (yb, xb) = tupleSel s_in b
86            (yc, _) = tupleSel s_in c
87
88
89  ctrlDevLogic :: (AbstExt x -> AbstExt s_in -> AbstExt x, AbstExt x -> AbstExt s_in
        -> AbstExt y, Int)
90               -> AbstExt Int -> Int -> (Int, Int, Int) -> ((Int, Int, Int), Int,
91                (AbstExt (AbstExt x -> AbstExt s_in -> AbstExt x, AbstExt x -> AbstExt
        s_in -> AbstExt y, Int),
92                 AbstExt (AbstExt x -> AbstExt s_in -> AbstExt x, AbstExt x -> AbstExt
        s_in -> AbstExt y, Int),
93                 AbstExt (AbstExt x -> AbstExt s_in -> AbstExt x, AbstExt x -> AbstExt
        s_in -> AbstExt y, Int),
94                 AbstExt (AbstExt x -> AbstExt s_in -> AbstExt x, AbstExt x -> AbstExt
        s_in -> AbstExt y, Int),
95                 AbstExt (AbstExt x -> AbstExt s_in -> AbstExt x, AbstExt x -> AbstExt
        s_in -> AbstExt y, Int)))
96  ctrlDevLogic _ Abst m cv
97      | m > 0 = (cv, m-1, list4tuple (replicate n Abst))
98      | otherwise = (cv, 0, list4tuple (replicate n Abst))
99      where n = 5
100 ctrlDevLogic (f,g,m') (Prst r) m (a,b,c)
101     | m > 0 = ((a,b,c), m-1, list4tuple (replicate n Abst))
102     | r == a = ((d,b,c), m', list4tuple (func n (d-1) (Prst (f,g,m')) Abst))
103     | r == b = ((a,d,c), m', list4tuple (func n (d-1) (Prst (f,g,m')) Abst))
104     | r == c = ((a,b,d), m', list4tuple (func n (d-1) (Prst (f,g,m')) Abst))
105     | otherwise = error "ctrlDevLogic: Unmatched pattern"
106     where d = max (max a b) c + 1
107           n = 5
108
```

```
109
110 func :: Int -> Int -> a -> a -> [a]
111 func 0 _ _ _ = []
112 func n 0 a1 a2 = a1 : replicate (n-1) a2
113 func n k a1 a2 = a2 : func (n-1) (k-1) a1 a2
114
115
116 list4tuple :: [a] -> (a,a,a,a,a)
117 list4tuple [a,b,c,d,e] = (a,b,c,d,e)
118 list4tuple _ = error "list4tuple: Input list with wrong length"
119
120
121 -- RTRPs functionalities
122
123 prosopon1 :: (AbstExt Int -> AbstExt Int -> AbstExt Int, AbstExt Int -> AbstExt Int
        -> AbstExt Int, Int)
124 prosopon1 = (stateTran1, stateOut, 2)
125
126
127 stateTranBreak :: AbstExt Int -> AbstExt Int -> AbstExt Int
128 stateTranBreak Abst _ = Abst
129 stateTranBreak _ Abst = Abst
130 stateTranBreak (Prst x) (Prst s_in)
131     | x == 3 = Prst (x - s_in)
132     | otherwise = Prst (x + s_in)
133
134 stateTran1 :: AbstExt Int -> AbstExt Int -> AbstExt Int
135 stateTran1 (Prst x) (Prst s_in) = Prst (x + s_in)
136 stateTran1 Abst _ = Abst
137 stateTran1 _ Abst = Abst
138
139 stateTran2 :: AbstExt Int -> AbstExt Int -> AbstExt Int
140 stateTran2 (Prst x) (Prst s_in) = Prst (x - s_in)
141 stateTran2 Abst _ = Abst
142 stateTran2 _ Abst = Abst
143
144 stateOut :: AbstExt Int -> AbstExt Int -> AbstExt Int
145 stateOut a _ = a
146
147 -- Process Definitions
148
149 rtrp :: (Eq x, Eq y) => (AbstExt x -> AbstExt s_in -> AbstExt x, AbstExt x ->
        AbstExt s_in -> AbstExt y, Int, AbstExt x)  -- ^ Initial configuration
150       -> Signal (AbstExt (AbstExt x -> AbstExt s_in -> AbstExt x, AbstExt x ->
        AbstExt s_in -> AbstExt y, Int))  -- ^ Control input
151       -> Signal (AbstExt s_in)            -- ^ Signal input
152       -> Signal (AbstExt x)               -- ^ Initial state input
153       -> Signal (AbstExt y, AbstExt x)    -- ^ Output and state
154 rtrp (f0,g0,m0,x0) ct s_in x' = out
155     where (out, fb) = unzipSY $ comb4SY rtrpFunc ct s_in x' fb'
156           fb' = delaySY (f0,g0,m0,x0) fb
157
158
159 voter :: (Eq y) => Signal (Int, Int, Int)
160       -> Signal (AbstExt y, AbstExt x)
161       -> Signal (AbstExt y, AbstExt x)
162       -> Signal (AbstExt y, AbstExt x)
163       -> Signal (AbstExt y, AbstExt x)
164       -> Signal (AbstExt y, AbstExt x)
165       -> (Signal (AbstExt y), Signal (AbstExt x), Signal (AbstExt Int))
166 voter cv s1 s2 s3 s4 s5 = unzip3SY $ comb2SY voterFunc cv (zip5SY s1 s2 s3 s4 s5)
167
168
169 ctrlDev :: Signal (AbstExt Int) -> (Signal (Int, Int, Int),
170         (Signal (AbstExt (AbstExt Int -> AbstExt Int -> AbstExt Int, AbstExt Int ->
        AbstExt Int -> AbstExt Int, Int)),
171          Signal (AbstExt (AbstExt Int -> AbstExt Int -> AbstExt Int, AbstExt Int ->
        AbstExt Int -> AbstExt Int, Int)),
172          Signal (AbstExt (AbstExt Int -> AbstExt Int -> AbstExt Int, AbstExt Int ->
        AbstExt Int -> AbstExt Int, Int)),
```

```
173           Signal (AbstExt (AbstExt Int -> AbstExt Int -> AbstExt Int, AbstExt Int ->
      AbstExt Int -> AbstExt Int, Int)),
174           Signal (AbstExt (AbstExt Int -> AbstExt Int -> AbstExt Int, AbstExt Int ->
      AbstExt Int -> AbstExt Int, Int))))
175 ctrlDev r = (cv, cts)
176     where (cv, m, ct) = unzip3SY $ comb3SY (ctrlDevLogic prosopon1) r m' cv'
177           cts = unzip5SY ct
178           m' = delaySY 0 m
179           cv' = delaySY (1,2,3) cv
180
181
182 -- Process Network
183
184 rtrp1 = rtrp (stateTran1, stateOut, 0, Abst)
185 rtrp2 = rtrp (stateTranBreak, stateOut, 0, Abst)
186 rtrp3 = rtrp (stateTran1, stateOut, 0, Abst)
187 rtrp4 = rtrp (stateTran2, stateOut, 0, Abst)
188 rtrp5 = rtrp (stateTran2, stateOut, 0, Abst)
189
190 -- tripleMR :: Signal (AbstExt Int) -> (Signal (AbstExt Int), Signal (AbstExt Int))
191 tmrPN s_in = (r, s_out, out2, out4)
192     where out1 = rtrp1 ct1' s_in x'
193           out2 = rtrp2 ct2' s_in x'
194           out3 = rtrp3 ct3' s_in x'
195           out4 = rtrp4 ct4' s_in x'
196           out5 = rtrp5 ct5' s_in x'
197           (s_out, x, r) = voter cv' out1 out2 out3 out4 out5
198           x' = delaySY (Prst 0) x
199           (cv, (ct1,ct2,ct3,ct4,ct5)) = ctrlDev r
200           cv' = delaySY (1,2,3) cv
201           ct1' = delaySY Abst ct1
202           ct2' = delaySY Abst ct2
203           ct3' = delaySY Abst ct3
204           ct4' = delaySY Abst ct4
205           ct5' = delaySY Abst ct5
206
207
208 -- Test Input
209
210 signalIn :: Signal (AbstExt Int)
211 signalIn = signal $ replicate 11 (Prst 1)
```

A.2 ForSyDe/SADF

```
1  ------------------------------------------------------------------------------
2  -- |
3  -- Module  :  ForSyDe.Shallow.MoC.SADF
4  -- Copyright   :  (c) Ricardo Bonna, KTH/ICT/ES, ForSyDe-Group
5  -- License     :  BSD-style (see the file LICENSE)
6  --
7  -- Maintainer  :  ricardobonna@gmail.com
8  -- Stability   :  experimental
9  -- Portability :  portable
10 --
11 -- Experimental lib. Further test needed
12 --
13 ------------------------------------------------------------------------------
14
15 module ForSyDe.Shallow.MoC.SADF (
16   -- * Sequential Process Constructors
17   -- | Sequential process constructors are used for processes that
18   -- have a state. One of the input parameters is the initial state.
19   delaySADF,
```

```
20    -- * Kernels
21    -- | Based on the process constructors in the SADF-MoC, the
22    -- SADF-library provides SADF-kernels with single or multiple inputs
23    kernel11SADF, kernel12SADF, kernel13SADF, kernel14SADF, kernel15SADF,
24    kernel21SADF, kernel22SADF, kernel23SADF, kernel24SADF, kernel25SADF,
25    kernel31SADF, kernel32SADF, kernel33SADF, kernel34SADF, kernel35SADF,
26    kernel41SADF, kernel42SADF, kernel43SADF, kernel44SADF, kernel45SADF,
27    kernel51SADF, kernel52SADF, kernel53SADF, kernel54SADF, kernel55SADF,
28    -- * Detectors
29    -- | Based on the process constructors in the SADF-MoC, the
30    -- SADF-library provides SADF-detectors with single or multiple inputs
31    detector11SADF, detector12SADF, detector13SADF, detector14SADF, detector15SADF,
32    detector21SADF, detector22SADF, detector23SADF, detector24SADF, detector25SADF,
33    detector31SADF, detector32SADF, detector33SADF, detector34SADF, detector35SADF,
34    detector41SADF, detector42SADF, detector43SADF, detector44SADF, detector45SADF,
35    detector51SADF, detector52SADF, detector53SADF, detector54SADF, detector55SADF
36    ) where
37
38  import ForSyDe.Shallow.Core
39
40
41  ------------------------------------
42  --              --
43  -- SEQUENTIAL PROCESS CONSTRUCTORS --
44  --              --
45  ------------------------------------
46
47  -- | The process constructor 'delaynSADF' delays the signal n event
48  --   cycles by introducing n initial values at the beginning of the
49  --   output signal.
50  delaySADF :: [a] -> Signal a -> Signal a
51  delaySADF initial_tokens xs = signal initial_tokens +-+ xs
52
53
54  ---------------------------------------------------------------------------
55  --
56  -- SADF KERNELS
57  --
58  ---------------------------------------------------------------------------
59
60  -- > Kernels with one output
61
62  -- | The process constructor 'kernel11SADF' constructs an SADF kernel with
63  -- one data input and one data output signals. The scenario (token rates and
64  -- function) is determined by the control signal.
65  kernel11SADF :: Signal (Int, Int, [a] -> [b]) -- ^ Control signal
66               -> Signal a                      -- ^ Input
67               -> Signal b                      -- ^ Output
68  kernel11SADF = mapSADF
69
70  -- | The process constructor 'kernel21SADF' constructs an SADF kernel with
71  -- two data input and one data output signals. The scenario (token rates and
72  -- function) is determined by the control signal.
73  kernel21SADF :: Signal ((Int, Int), Int, [a] -> [b] -> [c])
74               -> Signal a -> Signal b
75               -> Signal c
76  kernel21SADF = zipWithSADF
77
78  -- | The process constructor 'kernel31SADF' constructs an SADF kernel with
79  -- three data input and one data output signals. The scenario (token rates and
80  -- function) is determined by the control signal.
81  kernel31SADF :: Signal ((Int, Int, Int), Int, [a] -> [b] -> [c] -> [d])
82               -> Signal a -> Signal b -> Signal c
83               -> Signal d
84  kernel31SADF = zipWith3SADF
85
86  -- | The process constructor 'kernel41SADF' constructs an SADF kernel with
87  -- four data input and one data output signals. The scenario (token rates and
88  -- function) is determined by the control signal.
89  kernel41SADF :: Signal ((Int, Int, Int, Int), Int, [a] -> [b] -> [c] -> [d] -> [e])
90               -> Signal a -> Signal b -> Signal c -> Signal d
```

```
91              -> Signal e
92  kernel41SADF = zipWith4SADF
93
94  -- | The process constructor 'kernel51SADF' constructs an SADF kernel with
95  -- five data input and one data output signals. The scenario (token rates and
96  -- function) is determined by the control signal.
97  kernel51SADF :: Signal ((Int, Int, Int, Int, Int), Int, [a] -> [b] -> [c] -> [d] ->
        [e] -> [f])
98              -> Signal a -> Signal b -> Signal c -> Signal d -> Signal e
99              -> Signal f
100 kernel51SADF = zipWith5SADF
101
102
103 -- > Kernels with two outputs
104
105 -- | The process constructor 'kernel12SADF' constructs an SADF kernel with
106 -- one data input and two data output signals. The scenario (token rates and
107 -- function) is determined by the control signal.
108 kernel12SADF :: Signal (Int, (Int, Int), [a] -> ([b], [c]))
109             -> Signal a
110             -> (Signal b, Signal c)
111 kernel12SADF ct xs = unzipSADF (get_prodToken ct) $ mapSADF (inpOut1n ct) xs
112
113 -- | The process constructor 'kernel22SADF' constructs an SADF kernel with
114 -- two data input and two data output signals. The scenario (token rates and
115 -- function) is determined by the control signal.
116 --
117 -- >>> let scen1 = ((1,1), (1,1), \[a] [b] -> ([2*a], [2*b]))
118 -- >>> let scen2 = ((2,2), (1,1), \[a,b] [c,d] -> ([a+b], [c+d]))
119 -- >>> let scen3 = ((1,2), (2,1), \[a] [b,c] -> ([b,c], [a]))
120 -- >>> let sc = signal [scen1, scen2, scen3]
121 -- >>> kernel22SADF sc (signal [1..20]) (signal [21 .. 40])
122 -- ({2,5,24,25},{42,45,4})
123 kernel22SADF :: Signal ((Int, Int), (Int, Int), [a] -> [b] -> ([c], [d]))
124             -> Signal a -> Signal b
125             -> (Signal c, Signal d)
126 kernel22SADF ct xs ys = unzipSADF (get_prodToken ct) $ zipWithSADF (inpOut2n ct) xs
        ys
127
128 -- | The process constructor 'kernel32SADF' constructs an SADF kernel with
129 -- three data input and two data output signals. The scenario (token rates and
130 -- function) is determined by the control signal.
131 kernel32SADF :: Signal ((Int, Int, Int), (Int, Int), [a] -> [b] -> [c] -> ([d], [e])
        )
132             -> Signal a -> Signal b -> Signal c
133             -> (Signal d, Signal e)
134 kernel32SADF ct as bs cs
135   = unzipSADF (get_prodToken ct) $ zipWith3SADF (inpOut3n ct) as bs cs
136
137 -- | The process constructor 'kernel42SADF' constructs an SADF kernel with
138 -- four data input and two data output signals. The scenario (token rates and
139 -- function) is determined by the control signal.
140 kernel42SADF :: Signal ((Int, Int, Int, Int), (Int, Int), [a] -> [b] -> [c] -> [d]
        -> ([e], [f]))
141             -> Signal a -> Signal b -> Signal c -> Signal d
142             -> (Signal e, Signal f)
143 kernel42SADF ct as bs cs ds
144   = unzipSADF (get_prodToken ct) $ zipWith4SADF (inpOut4n ct) as bs cs ds
145
146 -- | The process constructor 'kernel52SADF' constructs an SADF kernel with
147 -- five data input and two data output signals. The scenario (token rates and
148 -- function) is determined by the control signal.
149 kernel52SADF :: Signal ((Int, Int, Int, Int, Int), (Int, Int), [a]
150             -> [b] -> [c] -> [d] -> [e] -> ([f], [g]))
151             -> Signal a -> Signal b -> Signal c -> Signal d -> Signal e
152             -> (Signal f, Signal g)
153 kernel52SADF ct as bs cs ds es
154   = unzipSADF (get_prodToken ct) $ zipWith5SADF (inpOut5n ct) as bs cs ds es
155
156
157 -- > Kernels with three outputs
```

```
158
159 -- | The process constructor 'kernel13SADF' constructs an SADF kernel with
160 -- one data input and three data output signals. The scenario (token rates and
161 -- function) is determined by the control signal.
162 kernel13SADF :: Signal (Int, (Int, Int, Int), [a] -> ([b], [c], [d]))
163              -> Signal a
164              -> (Signal b, Signal c, Signal d)
165 kernel13SADF ct xs = unzip3SADF (get_prodToken ct) $ mapSADF (inpOut1n ct) xs
166
167 -- | The process constructor 'kernel23SADF' constructs an SADF kernel with
168 -- two data input and three data output signals. The scenario (token rates and
169 -- function) is determined by the control signal.
170 kernel23SADF :: Signal ((Int, Int), (Int, Int, Int), [a] -> [b] -> ([c], [d], [e]))
171              -> Signal a -> Signal b
172              -> (Signal c, Signal d, Signal e)
173 kernel23SADF ct xs ys = unzip3SADF (get_prodToken ct) $ zipWithSADF (inpOut2n ct) xs
      ys
174
175 -- | The process constructor 'kernel33SADF' constructs an SADF kernel with
176 -- three data input and three data output signals. The scenario (token rates and
177 -- function) is determined by the control signal.
178 kernel33SADF :: Signal ((Int, Int, Int), (Int, Int, Int), [a] -> [b] -> [c] -> ([d],
      [e], [f]))
179              -> Signal a -> Signal b -> Signal c
180              -> (Signal d, Signal e, Signal f)
181 kernel33SADF ct as bs cs
182   = unzip3SADF (get_prodToken ct) $ zipWith3SADF (inpOut3n ct) as bs cs
183
184 -- | The process constructor 'kernel43SADF' constructs an SADF kernel with
185 -- four data input and three data output signals. The scenario (token rates and
186 -- function) is determined by the control signal.
187 kernel43SADF :: Signal ((Int, Int, Int, Int), (Int, Int, Int),
188              [a] -> [b] -> [c] -> [d] -> ([e], [f], [g]))
189              -> Signal a -> Signal b -> Signal c -> Signal d
190              -> (Signal e, Signal f, Signal g)
191 kernel43SADF ct as bs cs ds
192   = unzip3SADF (get_prodToken ct) $ zipWith4SADF (inpOut4n ct) as bs cs ds
193
194 -- | The process constructor 'kernel53SADF' constructs an SADF kernel with
195 -- five data input and three data output signals. The scenario (token rates and
196 -- function) is determined by the control signal.
197 kernel53SADF :: Signal ((Int, Int, Int, Int, Int), (Int, Int, Int),
198              [a] -> [b] -> [c] -> [d] -> [e] -> ([f], [g], [h]))
199              -> Signal a -> Signal b -> Signal c -> Signal d -> Signal e
200              -> (Signal f, Signal g, Signal h)
201 kernel53SADF ct as bs cs ds es
202   = unzip3SADF (get_prodToken ct) $ zipWith5SADF (inpOut5n ct) as bs cs ds es
203
204
205 -- > Kernels with four outputs
206
207 -- | The process constructor 'kernel14SADF' constructs an SADF kernel with
208 -- one data input and four data output signals. The scenario (token rates and
209 -- function) is determined by the control signal.
210 kernel14SADF :: Signal (Int, (Int, Int, Int, Int), [a] -> ([b], [c], [d], [e]))
211              -> Signal a
212              -> (Signal b, Signal c, Signal d, Signal e)
213 kernel14SADF ct xs = unzip4SADF (get_prodToken ct) $ mapSADF (inpOut1n ct) xs
214
215 -- | The process constructor 'kernel24SADF' constructs an SADF kernel with
216 -- two data input and four data output signals. The scenario (token rates and
217 -- function) is determined by the control signal.
218 kernel24SADF :: Signal ((Int, Int), (Int, Int, Int, Int), [a] -> [b] -> ([c], [d], [
      e], [f]))
219              -> Signal a -> Signal b
220              -> (Signal c, Signal d, Signal e, Signal f)
221 kernel24SADF ct xs ys = unzip4SADF (get_prodToken ct) $ zipWithSADF (inpOut2n ct) xs
      ys
222
223 -- | The process constructor 'kernel34SADF' constructs an SADF kernel with
224 -- three data input and four data output signals. The scenario (token rates and
```

```
225 -- function) is determined by the control signal.
226 kernel34SADF :: Signal ((Int, Int, Int), (Int, Int, Int, Int),
227               [a] -> [b] -> [c] -> ([d], [e], [f], [g]))
228               -> Signal a -> Signal b -> Signal c
229               -> (Signal d, Signal e, Signal f, Signal g)
230 kernel34SADF ct as bs cs
231   = unzip4SADF (get_prodToken ct) $ zipWith3SADF (inpOut3n ct) as bs cs
232
233 -- | The process constructor 'kernel44SADF' constructs an SADF kernel with
234 -- four data input and four data output signals. The scenario (token rates and
235 -- function) is determined by the control signal.
236 kernel44SADF :: Signal ((Int, Int, Int, Int), (Int, Int, Int, Int),
237               [a] -> [b] -> [c] -> [d] -> ([e], [f], [g], [h]))
238               -> Signal a -> Signal b -> Signal c -> Signal d
239               -> (Signal e, Signal f, Signal g, Signal h)
240 kernel44SADF ct as bs cs ds
241   = unzip4SADF (get_prodToken ct) $ zipWith4SADF (inpOut4n ct) as bs cs ds
242
243 -- | The process constructor 'kernel54SADF' constructs an SADF kernel with
244 -- five data input and four data output signals. The scenario (token rates and
245 -- function) is determined by the control signal.
246 kernel54SADF :: Signal ((Int, Int, Int, Int, Int), (Int, Int, Int, Int),
247               [a] -> [b] -> [c] -> [d] -> [e] -> ([f], [g], [h], [i]))
248               -> Signal a -> Signal b -> Signal c -> Signal d -> Signal e
249               -> (Signal f, Signal g, Signal h, Signal i)
250 kernel54SADF ct as bs cs ds es
251   = unzip4SADF (get_prodToken ct) $ zipWith5SADF (inpOut5n ct) as bs cs ds es
252
253
254 -- > Kernels with five outputs
255
256 -- | The process constructor 'kernel15SADF' constructs an SADF kernel with
257 -- one data input and five data output signals. The scenario (token rates and
258 -- function) is determined by the control signal.
259 kernel15SADF :: Signal (Int, (Int, Int, Int, Int, Int), [a] -> ([b], [c], [d], [e],
        [f]))
260               -> Signal a
261               -> (Signal b, Signal c, Signal d, Signal e, Signal f)
262 kernel15SADF ct xs = unzip5SADF (get_prodToken ct) $ mapSADF (inpOut1n ct) xs
263
264 -- | The process constructor 'kernel25SADF' constructs an SADF kernel with
265 -- two data input and five data output signals. The scenario (token rates and
266 -- function) is determined by the control signal.
267 kernel25SADF :: Signal ((Int, Int), (Int, Int, Int, Int, Int), [a] -> [b] -> ([c], [
        d], [e], [f], [g]))
268               -> Signal a -> Signal b
269               -> (Signal c, Signal d, Signal e, Signal f, Signal g)
270 kernel25SADF ct xs ys = unzip5SADF (get_prodToken ct) $ zipWithSADF (inpOut2n ct) xs
         ys
271
272 -- | The process constructor 'kernel35SADF' constructs an SADF kernel with
273 -- three data input and five data output signals. The scenario (token rates and
274 -- function) is determined by the control signal.
275 kernel35SADF :: Signal ((Int, Int, Int), (Int, Int, Int, Int, Int),
276               [a] -> [b] -> [c] -> ([d], [e], [f], [g], [h]))
277               -> Signal a -> Signal b -> Signal c
278               -> (Signal d, Signal e, Signal f, Signal g, Signal h)
279 kernel35SADF ct as bs cs
280   = unzip5SADF (get_prodToken ct) $ zipWith3SADF (inpOut3n ct) as bs cs
281
282 -- | The process constructor 'kernel45SADF' constructs an SADF kernel with
283 -- four data input and five data output signals. The scenario (token rates and
284 -- function) is determined by the control signal.
285 kernel45SADF :: Signal ((Int, Int, Int, Int), (Int, Int, Int, Int, Int),
286               [a] -> [b] -> [c] -> [d] -> ([e], [f], [g], [h], [i]))
287               -> Signal a -> Signal b -> Signal c -> Signal d
288               -> (Signal e, Signal f, Signal g, Signal h, Signal i)
289 kernel45SADF ct as bs cs ds
290   = unzip5SADF (get_prodToken ct) $ zipWith4SADF (inpOut4n ct) as bs cs ds
291
292 -- | The process constructor 'kernel55SADF' constructs an SADF kernel with
```

```
293 -- five data input and five data output signals. The scenario (token rates and
294 -- function) is determined by the control signal.
295 kernel55SADF :: Signal ((Int, Int, Int, Int, Int), (Int, Int, Int, Int, Int),
296              [a] -> [b] -> [c] -> [d] -> [e] -> ([f], [g], [h], [i], [j]))
297             -> Signal a -> Signal b -> Signal c -> Signal d -> Signal e
298             -> (Signal f, Signal g, Signal h, Signal i, Signal j)
299 kernel55SADF ct as bs cs ds es
300   = unzip5SADF (get_prodToken ct) $ zipWith5SADF (inpOut5n ct) as bs cs ds es
301
302
303 ------------------------------------------------------------------------
304 --
305 -- SADF DETECTORS
306 --
307 ------------------------------------------------------------------------
308
309 -- > Detectors with one output
310
311 -- | The process constructor 'detector11SADF' takes the consumption token rate
312 -- (@c@), the state transition function (@f@), the scenario selection (@g@) and
313 -- the initial state (@s0@), and constructs an SADF detector with
314 -- a single data input and a single control output signals.
315 detector11SADF :: Int                  -- ^ consumption rates (@c@)
316                -> (s -> [a] -> s)      -- ^ next state function (@f@)
317                -> (s -> (Int, [y]))    -- ^ scenario selection (@g@)
318                -> s                    -- ^ initial state (@s0@)
319                -> Signal a             -- ^ Input
320                -> Signal y             -- ^ Output
321 detector11SADF c f g s0 as = outputFSM g next_state
322   where next_state = nextStateFSM c f current_state as
323         current_state = delaySADF [s0] next_state
324
325 -- | The process constructor 'detector21SADF' takes the consumption token rate
326 -- (@c@), the state transition function (@f@), the scenario selection (@g@) and
327 -- the initial state (@s0@), and constructs an SADF detector with two data input and
      a
328 -- single control output signals.
329 detector21SADF :: (Int, Int)
330                -> (s -> [a] -> [b] -> s)
331                -> (s -> (Int, [y]))
332                -> s
333                -> Signal a -> Signal b
334                -> Signal y
335 detector21SADF c f g s0 as bs = outputFSM g next_state
336   where next_state = nextStateFSM2 c f current_state as bs
337         current_state = delaySADF [s0] next_state
338
339 -- | The process constructor 'detector31SADF' takes the consumption token rate
340 -- (@c@), the state transition function (@f@), the scenario selection (@g@) and
341 -- the initial state (@s0@), and constructs an SADF detector with three data input
     and a
342 -- single control output signals.
343 detector31SADF :: (Int, Int, Int)
344                -> (s -> [a] -> [b] -> [c] -> s)
345                -> (s -> (Int, [y]))
346                -> s
347                -> Signal a -> Signal b -> Signal c
348                -> Signal y
349 detector31SADF c f g s0 as bs cs = outputFSM g next_state
350   where next_state = nextStateFSM3 c f current_state as bs cs
351         current_state = delaySADF [s0] next_state
352
353 -- | The process constructor 'detector41SADF' takes the consumption token rate
354 -- (@c@), the state transition function (@f@), the scenario selection (@g@) and
355 -- the initial state (@s0@), and constructs an SADF detector with four data input
     and a
356 -- single control output signals.
357 detector41SADF :: (Int, Int, Int, Int)
358                -> (s -> [a] -> [b] -> [c] -> [d] -> s)
359                -> (s -> (Int, [y]))
360                -> s
```

```
361                 -> Signal a -> Signal b -> Signal c -> Signal d
362                 -> Signal y
363 detector41SADF c f g s0 as bs cs ds = outputFSM g next_state
364   where next_state = nextStateFSM4 c f current_state as bs cs ds
365         current_state = delaySADF [s0] next_state
366
367 -- | The process constructor 'detector51SADF' takes the consumption token rate
368 -- (@c@), the state transition function (@f@), the scenario selection (@g@) and
369 -- the initial state (@s0@), and constructs an SADF detector with five data input
     and a
370 -- single control output signals.
371 detector51SADF :: (Int, Int, Int, Int, Int)
372                 -> (s -> [a] -> [b] -> [c] -> [d] -> [e] -> s)
373                 -> (s -> (Int, [y]))
374                 -> s
375                 -> Signal a -> Signal b -> Signal c -> Signal d -> Signal e
376                 -> Signal y
377 detector51SADF c f g s0 as bs cs ds es = outputFSM g next_state
378   where next_state = nextStateFSM5 c f current_state as bs cs ds es
379         current_state = delaySADF [s0] next_state
380
381
382 -- > Detectors with two output
383
384 -- | The process constructor 'detector12SADF' takes the consumption token rate
385 -- (@c@), the state transition function (@f@), the scenario selection (@g@) and
386 -- the initial state (@s0@), and constructs an SADF detector with a single data
     input and two
387 -- control output signals.
388 detector12SADF :: Int
389                 -> (s -> [a] -> s)
390                 -> (s -> ((Int, Int), ([y1], [y2])))
391                 -> s
392                 -> Signal a
393                 -> (Signal y1, Signal y2)
394 detector12SADF c f g s0 as = outputFSM2 g next_state
395   where next_state = nextStateFSM c f current_state as
396         current_state = delaySADF [s0] next_state
397
398 -- | The process constructor 'detector22SADF' takes the consumption token rate
399 -- (@c@), the state transition function (@f@), the scenario selection (@g@) and
400 -- the initial state (@s0@), and constructs an SADF detector with two data input and
      two
401 -- control output signals.
402 detector22SADF :: (Int, Int)
403                 -> (s -> [a] -> [b] -> s)
404                 -> (s -> ((Int, Int), ([y1], [y2])))
405                 -> s
406                 -> Signal a -> Signal b
407                 -> (Signal y1, Signal y2)
408 detector22SADF c f g s0 as bs = outputFSM2 g next_state
409   where next_state = nextStateFSM2 c f current_state as bs
410         current_state = delaySADF [s0] next_state
411
412 -- | The process constructor 'detector32SADF' takes the consumption token rate
413 -- (@c@), the state transition function (@f@), the scenario selection (@g@) and
414 -- the initial state (@s0@), and constructs an SADF detector with three data input
     and two
415 -- control output signals.
416 detector32SADF :: (Int, Int, Int)
417                 -> (s -> [a] -> [b] -> [c] -> s)
418                 -> (s -> ((Int, Int), ([y1], [y2])))
419                 -> s
420                 -> Signal a -> Signal b -> Signal c
421                 -> (Signal y1, Signal y2)
422 detector32SADF c f g s0 as bs cs = outputFSM2 g next_state
423   where next_state = nextStateFSM3 c f current_state as bs cs
424         current_state = delaySADF [s0] next_state
425
426 -- | The process constructor 'detector42SADF' takes the consumption token rate
427 -- (@c@), the state transition function (@f@), the scenario selection (@g@) and
```

```
428 -- the initial state (@s0@), and constructs an SADF detector with four data input
        and two
429 -- control output signals.
430 detector42SADF :: (Int, Int, Int, Int)
431                 -> (s -> [a] -> [b] -> [c] -> [d] -> s)
432                 -> (s -> ((Int, Int), ([y1], [y2])))
433                 -> s
434                 -> Signal a -> Signal b -> Signal c -> Signal d
435                 -> (Signal y1, Signal y2)
436 detector42SADF c f g s0 as bs cs ds = outputFSM2 g next_state
437   where next_state = nextStateFSM4 c f current_state as bs cs ds
438         current_state = delaySADF [s0] next_state
439
440 -- | The process constructor 'detector52SADF' takes the consumption token rate
441 -- (@c@), the state transition function (@f@), the scenario selection (@g@) and
442 -- the initial state (@s0@), and constructs an SADF detector with five data input
        and two
443 -- control output signals.
444 detector52SADF :: (Int, Int, Int, Int, Int)
445                 -> (s -> [a] -> [b] -> [c] -> [d] -> [e] -> s)
446                 -> (s -> ((Int, Int), ([y1], [y2])))
447                 -> s
448                 -> Signal a -> Signal b -> Signal c -> Signal d -> Signal e
449                 -> (Signal y1, Signal y2)
450 detector52SADF c f g s0 as bs cs ds es = outputFSM2 g next_state
451   where next_state = nextStateFSM5 c f current_state as bs cs ds es
452         current_state = delaySADF [s0] next_state
453
454
455 -- > Detectors with three output
456
457 -- | The process constructor 'detector13SADF' takes the consumption token rate
458 -- (@c@), the state transition function (@f@), the scenario selection (@g@) and
459 -- the initial state (@s0@), and constructs an SADF detector with a single data
        input and three
460 -- control output signals.
461 detector13SADF :: Int
462                 -> (s -> [a] -> s)
463                 -> (s -> ((Int, Int, Int), ([y1], [y2], [y3])))
464                 -> s
465                 -> Signal a
466                 -> (Signal y1, Signal y2, Signal y3)
467 detector13SADF c f g s0 as = outputFSM3 g next_state
468   where next_state = nextStateFSM c f current_state as
469         current_state = delaySADF [s0] next_state
470
471 -- | The process constructor 'detector23SADF' takes the consumption token rate
472 -- (@c@), the state transition function (@f@), the scenario selection (@g@) and
473 -- the initial state (@s0@), and constructs an SADF detector with two data input and
        three
474 -- control output signals.
475 detector23SADF :: (Int, Int)
476                 -> (s -> [a] -> [b] -> s)
477                 -> (s -> ((Int, Int, Int), ([y1], [y2], [y3])))
478                 -> s
479                 -> Signal a -> Signal b
480                 -> (Signal y1, Signal y2, Signal y3)
481 detector23SADF c f g s0 as bs = outputFSM3 g next_state
482   where next_state = nextStateFSM2 c f current_state as bs
483         current_state = delaySADF [s0] next_state
484
485 -- | The process constructor 'detector33SADF' takes the consumption token rate
486 -- (@c@), the state transition function (@f@), the scenario selection (@g@) and
487 -- the initial state (@s0@), and constructs an SADF detector with three data input
        and three
488 -- control output signals.
489 detector33SADF :: (Int, Int, Int)
490                 -> (s -> [a] -> [b] -> [c] -> s)
491                 -> (s -> ((Int, Int, Int), ([y1], [y2], [y3])))
492                 -> s
493                 -> Signal a -> Signal b -> Signal c
```

```
494                 -> (Signal y1, Signal y2, Signal y3)
495 detector33SADF c f g s0 as bs cs = outputFSM3 g next_state
496   where next_state = nextStateFSM3 c f current_state as bs cs
497         current_state = delaySADF [s0] next_state
498
499 -- | The process constructor 'detector43SADF' takes the consumption token rate
500 -- (@c@), the state transition function (@f@), the scenario selection (@g@) and
501 -- the initial state (@s0@), and constructs an SADF detector with four data input
      and three
502 -- control output signals.
503 detector43SADF :: (Int, Int, Int, Int)
504                 -> (s -> [a] -> [b] -> [c] -> [d] -> s)
505                 -> (s -> ((Int, Int, Int), ([y1], [y2], [y3])))
506                 -> s
507                 -> Signal a -> Signal b -> Signal c -> Signal d
508                 -> (Signal y1, Signal y2, Signal y3)
509 detector43SADF c f g s0 as bs cs ds = outputFSM3 g next_state
510   where next_state = nextStateFSM4 c f current_state as bs cs ds
511         current_state = delaySADF [s0] next_state
512
513 -- | The process constructor 'detector53SADF' takes the consumption token rate
514 -- (@c@), the state transition function (@f@), the scenario selection (@g@) and
515 -- the initial state (@s0@), and constructs an SADF detector with five data input
      and three
516 -- control output signals.
517 detector53SADF :: (Int, Int, Int, Int, Int)
518                 -> (s -> [a] -> [b] -> [c] -> [d] -> [e] -> s)
519                 -> (s -> ((Int, Int, Int), ([y1], [y2], [y3])))
520                 -> s
521                 -> Signal a -> Signal b -> Signal c -> Signal d -> Signal e
522                 -> (Signal y1, Signal y2, Signal y3)
523 detector53SADF c f g s0 as bs cs ds es = outputFSM3 g next_state
524   where next_state = nextStateFSM5 c f current_state as bs cs ds es
525         current_state = delaySADF [s0] next_state
526
527
528 -- > Detectors with four output
529
530 -- | The process constructor 'detector14SADF' takes the consumption token rate
531 -- (@c@), the state transition function (@f@), the scenario selection (@g@) and
532 -- the initial state (@s0@), and constructs an SADF detector with a single data
      input and four
533 -- control output signals.
534 detector14SADF :: Int
535                 -> (s -> [a] -> s)
536                 -> (s -> ((Int, Int, Int, Int), ([y1], [y2], [y3], [y4])))
537                 -> s
538                 -> Signal a
539                 -> (Signal y1, Signal y2, Signal y3, Signal y4)
540 detector14SADF c f g s0 as = outputFSM4 g next_state
541   where next_state = nextStateFSM c f current_state as
542         current_state = delaySADF [s0] next_state
543
544 -- | The process constructor 'detector24SADF' takes the consumption token rate
545 -- (@c@), the state transition function (@f@), the scenario selection (@g@) and
546 -- the initial state (@s0@), and constructs an SADF detector with two data input and
      four
547 -- control output signals.
548 detector24SADF :: (Int, Int)
549                 -> (s -> [a] -> [b] -> s)
550                 -> (s -> ((Int, Int, Int, Int), ([y1], [y2], [y3], [y4])))
551                 -> s
552                 -> Signal a -> Signal b
553                 -> (Signal y1, Signal y2, Signal y3, Signal y4)
554 detector24SADF c f g s0 as bs = outputFSM4 g next_state
555   where next_state = nextStateFSM2 c f current_state as bs
556         current_state = delaySADF [s0] next_state
557
558 -- | The process constructor 'detector34SADF' takes the consumption token rate
559 -- (@c@), the state transition function (@f@), the scenario selection (@g@) and
```

```
560 -- the initial state (@s0@), and constructs an SADF detector with three data input
        and four
561 -- control output signals.
562 detector34SADF :: (Int, Int, Int)
563                 -> (s -> [a] -> [b] -> [c] -> s)
564                 -> (s -> ((Int, Int, Int, Int), ([y1], [y2], [y3], [y4])))
565                 -> s
566                 -> Signal a -> Signal b -> Signal c
567                 -> (Signal y1, Signal y2, Signal y3, Signal y4)
568 detector34SADF c f g s0 as bs cs = outputFSM4 g next_state
569   where next_state = nextStateFSM3 c f current_state as bs cs
570         current_state = delaySADF [s0] next_state
571
572 -- | The process constructor 'detector44SADF' takes the consumption token rate
573 -- (@c@), the state transition function (@f@), the scenario selection (@g@) and
574 -- the initial state (@s0@), and constructs an SADF detector with four data input
        and four
575 -- control output signals.
576 detector44SADF :: (Int, Int, Int, Int)
577                 -> (s -> [a] -> [b] -> [c] -> [d] -> s)
578                 -> (s -> ((Int, Int, Int, Int), ([y1], [y2], [y3], [y4])))
579                 -> s
580                 -> Signal a -> Signal b -> Signal c -> Signal d
581                 -> (Signal y1, Signal y2, Signal y3, Signal y4)
582 detector44SADF c f g s0 as bs cs ds = outputFSM4 g next_state
583   where next_state = nextStateFSM4 c f current_state as bs cs ds
584         current_state = delaySADF [s0] next_state
585
586 -- | The process constructor 'detector54SADF' takes the consumption token rate
587 -- (@c@), the state transition function (@f@), the scenario selection (@g@) and
588 -- the initial state (@s0@), and constructs an SADF detector with five data input
        and four
589 -- control output signals.
590 detector54SADF :: (Int, Int, Int, Int, Int)
591                 -> (s -> [a] -> [b] -> [c] -> [d] -> [e] -> s)
592                 -> (s -> ((Int, Int, Int, Int), ([y1], [y2], [y3], [y4])))
593                 -> s
594                 -> Signal a -> Signal b -> Signal c -> Signal d -> Signal e
595                 -> (Signal y1, Signal y2, Signal y3, Signal y4)
596 detector54SADF c f g s0 as bs cs ds es = outputFSM4 g next_state
597   where next_state = nextStateFSM5 c f current_state as bs cs ds es
598         current_state = delaySADF [s0] next_state
599
600
601 -- > Detectors with five output
602
603 -- | The process constructor 'detector15SADF' takes the consumption token rate
604 -- (@c@), the state transition function (@f@), the scenario selection (@g@) and
605 -- the initial state (@s0@), and constructs an SADF detector with a single data
        input and five
606 -- control output signals.
607 detector15SADF :: Int
608                 -> (s -> [a] -> s)
609                 -> (s -> ((Int, Int, Int, Int, Int), ([y1], [y2], [y3], [y4], [y5])))
610                 -> s
611                 -> Signal a
612                 -> (Signal y1, Signal y2, Signal y3, Signal y4, Signal y5)
613 detector15SADF c f g s0 as = outputFSM5 g next_state
614   where next_state = nextStateFSM c f current_state as
615         current_state = delaySADF [s0] next_state
616
617 -- | The process constructor 'detector25SADF' takes the consumption token rate
618 -- (@c@), the state transition function (@f@), the scenario selection (@g@) and
619 -- the initial state (@s0@), and constructs an SADF detector with two data input and
        five
620 -- control output signals.
621 detector25SADF :: (Int, Int)
622                 -> (s -> [a] -> [b] -> s)
623                 -> (s -> ((Int, Int, Int, Int, Int), ([y1], [y2], [y3], [y4], [y5])))
624                 -> s
625                 -> Signal a -> Signal b
```

```
626                 -> (Signal y1, Signal y2, Signal y3, Signal y4, Signal y5)
627 detector25SADF c f g s0 as bs = outputFSM5 g next_state
628   where next_state = nextStateFSM2 c f current_state as bs
629         current_state = delaySADF [s0] next_state
630
631 -- | The process constructor 'detector35SADF' takes the consumption token rate
632 -- (@c@), the state transition function (@f@), the scenario selection (@g@) and
633 -- the initial state (@s0@), and constructs an SADF detector with three data input
      and five
634 -- control output signals.
635 detector35SADF :: (Int, Int, Int)
636                 -> (s -> [a] -> [b] -> [c] -> s)
637                 -> (s -> ((Int, Int, Int, Int, Int), ([y1], [y2], [y3], [y4], [y5])))
638                 -> s
639                 -> Signal a -> Signal b -> Signal c
640                 -> (Signal y1, Signal y2, Signal y3, Signal y4, Signal y5)
641 detector35SADF c f g s0 as bs cs = outputFSM5 g next_state
642   where next_state = nextStateFSM3 c f current_state as bs cs
643         current_state = delaySADF [s0] next_state
644
645 -- | The process constructor 'detector45SADF' takes the consumption token rate
646 -- (@c@), the state transition function (@f@), the scenario selection (@g@) and
647 -- the initial state (@s0@), and constructs an SADF detector with four data input
      and five
648 -- control output signals.
649 detector45SADF :: (Int, Int, Int, Int)
650                 -> (s -> [a] -> [b] -> [c] -> [d] -> s)
651                 -> (s -> ((Int, Int, Int, Int, Int), ([y1], [y2], [y3], [y4], [y5])))
652                 -> s
653                 -> Signal a -> Signal b -> Signal c -> Signal d
654                 -> (Signal y1, Signal y2, Signal y3, Signal y4, Signal y5)
655 detector45SADF c f g s0 as bs cs ds = outputFSM5 g next_state
656   where next_state = nextStateFSM4 c f current_state as bs cs ds
657         current_state = delaySADF [s0] next_state
658
659 -- | The process constructor 'detector55SADF' takes the consumption token rate
660 -- (@c@), the state transition function (@f@), the scenario selection (@g@) and
661 -- the initial state (@s0@), and constructs an SADF detector with five data input
      and five
662 -- control output signals.
663 detector55SADF :: (Int, Int, Int, Int, Int)
664                 -> (s -> [a] -> [b] -> [c] -> [d] -> [e] -> s)
665                 -> (s -> ((Int, Int, Int, Int, Int), ([y1], [y2], [y3], [y4], [y5])))
666                 -> s
667                 -> Signal a -> Signal b -> Signal c -> Signal d -> Signal e
668                 -> (Signal y1, Signal y2, Signal y3, Signal y4, Signal y5)
669 detector55SADF c f g s0 as bs cs ds es = outputFSM5 g next_state
670   where next_state = nextStateFSM5 c f current_state as bs cs ds es
671         current_state = delaySADF [s0] next_state
672
673
674 ------------------------------------------------------------------------
675 -- COMBINATIONAL PROCESS CONSTRUCTORS (not exported)
676 ------------------------------------------------------------------------
677
678 -- | The process constructor 'mapSADF' takes a signal of scenarios
679 -- (tuples with the consumed and produced tokens as well as a function operating
680 -- on lists), and results in an SADF-process that takes an input signal and results
681 -- in an output signal
682 mapSADF :: Signal (Int, Int, [a] -> [b]) -> Signal a -> Signal b
683 mapSADF NullS _ = NullS
684 mapSADF ct xs
685   | c < 0 = error "mapSADF: Number of consumed tokens must be a non-negative integer
      "
686   | not $ sufficient_tokens c xs  = NullS
687   | otherwise = if length produced_tokens == p then
688                   signal produced_tokens +-+ mapSADF (tailS ct) (dropS c xs)
689                 else
690                   error "mapSADF: Function does not produce correct number of tokens
      "
691   where (c, p, f) = headS ct
```

```
692         consumed_tokens = fromSignal $ takeS c xs
693         produced_tokens = f consumed_tokens
694
695
696 -- | The process constructor 'zipWithSADF' takes a signal of scenarios
697 -- (tuples with the consumed and produced tokens as well as a function operating
698 -- on lists), and results in an SADF-process that takes two input signals and
699 -- results in an output signal
700 zipWithSADF :: Signal ((Int, Int), Int, [a] -> [b] -> [c])
701             -> Signal a -> Signal b -> Signal c
702 zipWithSADF NullS _ _ = NullS
703 zipWithSADF ct as bs
704   | c1 < 0 || c2 < 0  = error "zipWithSADF: Number of consumed tokens must be a non-
      negative integer"
705   | (not $ sufficient_tokens c1 as)
706     || (not $ sufficient_tokens c2 bs) = NullS
707   | otherwise = if length produced_tokens == p then
708                   signal produced_tokens +-+ zipWithSADF (tailS ct) (dropS c1 as) (
      dropS c2 bs)
709                 else
710                   error "zipWithSADF: Function does not produce correct number of
      tokens"
711   where ((c1,c2), p, f) = headS ct
712         consumed_tokens_as = fromSignal $ takeS c1 as
713         consumed_tokens_bs = fromSignal $ takeS c2 bs
714         produced_tokens = f consumed_tokens_as consumed_tokens_bs
715
716
717 -- | The process constructor 'zipWith3SADF' takes a signal of scenarios
718 -- (tuples with the consumed and produced tokens as well as a function operating
719 -- on lists), and results in an SADF-process that takes three input signals and
720 -- results in an output signal
721 zipWith3SADF :: Signal ((Int, Int, Int), Int, [a] -> [b] -> [c] -> [d])
722              -> Signal a -> Signal b -> Signal c -> Signal d
723 zipWith3SADF NullS _ _ _ = NullS
724 zipWith3SADF ct as bs cs
725   | c1 < 0 || c2 < 0 || c3 < 0
726     = error "zipWith3SADF: Number of consumed tokens must be a non-negative integer"
727   | (not $ sufficient_tokens c1 as)
728     || (not $ sufficient_tokens c2 bs)
729     || (not $ sufficient_tokens c3 cs) = NullS
730   | otherwise = if length produced_tokens == p then
731                   signal produced_tokens +-+ zipWith3SADF (tailS ct) (dropS c1 as)
732                                                          (dropS c2 bs) (dropS c3 cs)
733                 else
734                   error "zipWith3SADF: Function does not produce correct number of
      tokens"
735   where ((c1, c2, c3), p, f) = headS ct
736         consumed_tokens_as = fromSignal $ takeS c1 as
737         consumed_tokens_bs = fromSignal $ takeS c2 bs
738         consumed_tokens_cs = fromSignal $ takeS c3 cs
739         produced_tokens = f consumed_tokens_as consumed_tokens_bs consumed_tokens_cs
740
741
742 -- | The process constructor 'zipWith4SADF' takes a signal of scenarios
743 -- (tuples with the consumed and produced tokens as well as a function operating
744 -- on lists), and results in an SADF-process that takes four input signals and
745 -- results in an output signal
746 zipWith4SADF :: Signal ((Int, Int, Int, Int), Int, [a] -> [b] -> [c] -> [d] -> [e])
747              -> Signal a -> Signal b -> Signal c -> Signal d -> Signal e
748 zipWith4SADF NullS _ _ _ _ = NullS
749 zipWith4SADF ct as bs cs ds
750   | c1 < 0 || c2 < 0 || c3 < 0 || c4 < 0
751     = error "zipWith4SADF: Number of consumed tokens must be a non-negative integer"
752   | (not $ sufficient_tokens c1 as)
753     || (not $ sufficient_tokens c2 bs)
754     || (not $ sufficient_tokens c3 cs)
755     || (not $ sufficient_tokens c4 ds) = NullS
756   | otherwise = if length produced_tokens == p then
757                   signal produced_tokens +-+ zipWith4SADF (tailS ct) (dropS c1 as)
```

```
758                                                 (dropS c2 bs) (dropS c3 cs) (dropS c4
      ds)
759                 else
760                   error "zipWith4SADF: Function does not produce correct number of
      tokens"
761     where ((c1, c2, c3, c4), p, f) = headS ct
762           consumed_tokens_as = fromSignal $ takeS c1 as
763           consumed_tokens_bs = fromSignal $ takeS c2 bs
764           consumed_tokens_cs = fromSignal $ takeS c3 cs
765           consumed_tokens_ds = fromSignal $ takeS c4 ds
766           produced_tokens = f consumed_tokens_as consumed_tokens_bs
767                               consumed_tokens_cs consumed_tokens_ds
768
769
770 -- | The process constructor 'zipWith5SADF' takes a signal of scenarios
771 -- (tuples with the consumed and produced tokens as well as a function operating
772 -- on lists), and results in an SADF-process that takes five input signals and
773 -- results in an output signal
774 zipWith5SADF :: Signal ((Int, Int, Int, Int, Int), Int, [a] -> [b] -> [c] -> [d] ->
      [e] -> [f])
775              -> Signal a -> Signal b -> Signal c -> Signal d -> Signal e -> Signal f
776 zipWith5SADF NullS _ _ _ _ _ = NullS
777 zipWith5SADF ct as bs cs ds es
778   | c1 < 0 || c2 < 0 || c3 < 0 || c4 < 0 || c5 < 0
779     = error "zipWith5SADF: Number of consumed tokens must be a non-negative integer"
780   | (not $ sufficient_tokens c1 as)
781     || (not $ sufficient_tokens c2 bs)
782     || (not $ sufficient_tokens c3 cs)
783     || (not $ sufficient_tokens c4 ds)
784     || (not $ sufficient_tokens c5 es) = NullS
785   | otherwise = if length produced_tokens == p then
786                   signal produced_tokens +-+ zipWith5SADF (tailS ct) (dropS c1 as)
787                                               (dropS c2 bs) (dropS c3 cs) (dropS c4
      ds) (dropS c5 es)
788                 else
789                   error "zipWith5SADF: Function does not produce correct number of
      tokens"
790     where ((c1, c2, c3, c4, c5), p, f) = headS ct
791           consumed_tokens_as = fromSignal $ takeS c1 as
792           consumed_tokens_bs = fromSignal $ takeS c2 bs
793           consumed_tokens_cs = fromSignal $ takeS c3 cs
794           consumed_tokens_ds = fromSignal $ takeS c4 ds
795           consumed_tokens_es = fromSignal $ takeS c5 es
796           produced_tokens = f consumed_tokens_as consumed_tokens_bs
797                               consumed_tokens_cs consumed_tokens_ds consumed_tokens_es
798
799
800 ------------------------------------------------------------------------
801 -- unzipSADF Processes (not exported)
802 ------------------------------------------------------------------------
803
804 unzipSADF :: [(Int, Int)] -> Signal ([a], [b]) -> (Signal a, Signal b)
805 unzipSADF [] _ = (NullS, NullS)
806 unzipSADF _ NullS = (NullS, NullS)
807 unzipSADF ((p1, p2) : ps) ((s1, s2) :- ss)
808   | length s1 /= p1 || length s2 /= p2 = error "unzipSADF: Process does not produce
      correct number of tokens"
809   | otherwise = (signal s1 +-+ sr1, signal s2 +-+ sr2)
810   where (sr1, sr2) = unzipSADF ps ss
811
812
813 unzip3SADF :: [(Int, Int, Int)] -> Signal ([a], [b], [c])
814            -> (Signal a, Signal b, Signal c)
815 unzip3SADF [] _ = (NullS, NullS, NullS)
816 unzip3SADF _ NullS = (NullS, NullS, NullS)
817 unzip3SADF ((p1, p2, p3) : ps) ((s1, s2, s3) :- ss)
818   | length s1 /= p1 || length s2 /= p2
819     || length s3 /= p3 = error "unzip3SADF: Process does not produce correct number
      of tokens"
820   | otherwise = (signal s1 +-+ sr1, signal s2 +-+ sr2, signal s3 +-+ sr3)
821   where (sr1, sr2, sr3) = unzip3SADF ps ss
```

```
822
823
824 unzip4SADF :: [(Int, Int, Int, Int)] -> Signal ([a], [b], [c], [d])
825             -> (Signal a, Signal b, Signal c, Signal d)
826 unzip4SADF [] _ = (NullS, NullS, NullS, NullS)
827 unzip4SADF _ NullS = (NullS, NullS, NullS, NullS)
828 unzip4SADF ((p1, p2, p3, p4) : ps) ((s1, s2, s3, s4) :- ss)
829   | length s1 /= p1 || length s2 /= p2
830     || length s3 /= p3 || length s4 /= p4 = error "unzip4SADF: Process does not
      produce correct number of tokens"
831   | otherwise = (signal s1 +-+ sr1, signal s2 +-+ sr2, signal s3 +-+ sr3, signal s4
      +-+ sr4)
832   where (sr1, sr2, sr3, sr4) = unzip4SADF ps ss
833
834
835 unzip5SADF :: [(Int, Int, Int, Int, Int)] -> Signal ([a], [b], [c], [d], [e])
836             -> (Signal a, Signal b, Signal c, Signal d, Signal e)
837 unzip5SADF [] _ = (NullS, NullS, NullS, NullS, NullS)
838 unzip5SADF _ NullS = (NullS, NullS, NullS, NullS, NullS)
839 unzip5SADF ((p1, p2, p3, p4, p5) : ps) ((s1, s2, s3, s4, s5) :- ss)
840   | length s1 /= p1 || length s2 /= p2
841     || length s3 /= p3 || length s4 /= p4
842     || length s5 /= p5 = error "unzip5SADF: Process does not produce correct number
      of tokens"
843   | otherwise = (signal s1 +-+ sr1, signal s2 +-+ sr2, signal s3 +-+ sr3,
844                  signal s4 +-+ sr4, signal s5 +-+ sr5)
845   where (sr1, sr2, sr3, sr4, sr5) = unzip5SADF ps ss
846
847 ------------------------------------------------------------------------
848 --
849 -- Helper functions (not exported!)
850 --
851 ------------------------------------------------------------------------
852
853 sufficient_tokens :: (Num a, Eq a, Ord a) => a -> Signal t -> Bool
854 sufficient_tokens 0 _      = True
855 sufficient_tokens _ NullS = False
856 sufficient_tokens n (_:-xs)
857  = if n < 0 then
858      error "sufficient_tokens: n must not be negative"
859    else
860      sufficient_tokens (n-1) xs
861
862
863 get_prodToken :: Signal (a,b,c) -> [b]
864 get_prodToken NullS = []
865 get_prodToken ((_, x, _):-xs) = x : get_prodToken xs
866
867
868 inpOut1n :: Signal (it, ot, [a] -> y) -> Signal (it, Int, [a] -> [y])
869 inpOut1n NullS = NullS
870 inpOut1n ((it, _, f):-xs) = (it, 1, \a -> [f a]) :- inpOut1n xs
871
872 inpOut2n :: Signal (it, ot, [a] -> [b] -> y) -> Signal (it, Int, [a] -> [b] -> [y])
873 inpOut2n NullS = NullS
874 inpOut2n ((it, _, f):-xs) = (it, 1, \a b -> [f a b]) :- inpOut2n xs
875
876 inpOut3n :: Signal (it, ot, [a] -> [b] -> [c] -> y)
877          -> Signal (it, Int, [a] -> [b] -> [c] -> [y])
878 inpOut3n NullS = NullS
879 inpOut3n ((it, _, f):-xs) = (it, 1, \a b c -> [f a b c]) :- inpOut3n xs
880
881 inpOut4n :: Signal (it, ot, [a] -> [b] -> [c] -> [d] -> y)
882          -> Signal (it, Int, [a] -> [b] -> [c] -> [d] -> [y])
883 inpOut4n NullS = NullS
884 inpOut4n ((it, _, f):-xs) = (it, 1, \a b c d -> [f a b c d]) :- inpOut4n xs
885
886 inpOut5n :: Signal (it, ot, [a] -> [b] -> [c] -> [d] -> [e] -> y)
887          -> Signal (it, Int, [a] -> [b] -> [c] -> [d] -> [e] -> [y])
888 inpOut5n NullS = NullS
889 inpOut5n ((it, _, f):-xs) = (it, 1, \a b c d e -> [f a b c d e]) :- inpOut5n xs
```

```
890
891 ------------------------------------------------------
892 -- Helper functios to the detector's FSM  (not exported)
893 ------------------------------------------------------
894
895 nextStateFSM :: Int -> (s -> [a] -> s)
896              -> Signal s -> Signal a -> Signal s
897 nextStateFSM _ _ NullS _ = NullS
898 nextStateFSM _ _ _ NullS = NullS
899 nextStateFSM c f ss as
900   | c <= 0 = error "nextStateFSM: Number of consumed tokens must be positive integer
      "
901   | not $ sufficient_tokens c as = NullS
902   | otherwise = signal [next_state] +-+ nextStateFSM c f (tailS ss) (dropS c as)
903   where consumed_tokens_as = fromSignal $ takeS c as
904         current_state = headS ss
905         next_state = f current_state consumed_tokens_as
906
907
908 nextStateFSM2 :: (Int, Int) -> (s -> [a] -> [b] -> s)
909               -> Signal s -> Signal a -> Signal b -> Signal s
910 nextStateFSM2 _ _ NullS _ _ = NullS
911 nextStateFSM2 _ _ _ NullS _ = NullS
912 nextStateFSM2 _ _ _ _ NullS = NullS
913 nextStateFSM2 (c1, c2) f ss as bs
914   | c1 <= 0 || c2 <= 0 = error "nextStateFSM2: Number of consumed tokens must be
      positive integer"
915   | (not $ sufficient_tokens c1 as)
916     || (not $ sufficient_tokens c2 bs) = NullS
917   | otherwise = signal [next_state] +-+ nextStateFSM2 (c1, c2) f (tailS ss) (dropS
      c1 as) (dropS c2 bs)
918   where consumed_tokens_as = fromSignal $ takeS c1 as
919         consumed_tokens_bs = fromSignal $ takeS c2 bs
920         current_state = headS ss
921         next_state = f current_state consumed_tokens_as consumed_tokens_bs
922
923
924 nextStateFSM3 :: (Int, Int, Int) -> (s -> [a] -> [b] -> [c] -> s)
925               -> Signal s -> Signal a -> Signal b -> Signal c -> Signal s
926 nextStateFSM3 _ _ NullS _ _ _ = NullS
927 nextStateFSM3 _ _ _ NullS _ _ = NullS
928 nextStateFSM3 _ _ _ _ NullS _ = NullS
929 nextStateFSM3 _ _ _ _ _ NullS = NullS
930 nextStateFSM3 (c1, c2, c3) f ss as bs cs
931   | c1 <= 0 || c2 <= 0 || c3 <= 0
932     = error "nextStateFSM3: Number of consumed tokens must be positive integer"
933   | (not $ sufficient_tokens c1 as)
934     || (not $ sufficient_tokens c2 bs)
935     || (not $ sufficient_tokens c3 cs) = NullS
936   | otherwise = signal [next_state] +-+ nextStateFSM3 (c1, c2, c3) f (tailS ss)
937                                         (dropS c1 as) (dropS c2 bs) (dropS c3 cs)
938   where consumed_tokens_as = fromSignal $ takeS c1 as
939         consumed_tokens_bs = fromSignal $ takeS c2 bs
940         consumed_tokens_cs = fromSignal $ takeS c3 cs
941         current_state = headS ss
942         next_state = f current_state consumed_tokens_as
943                        consumed_tokens_bs consumed_tokens_cs
944
945
946 nextStateFSM4 :: (Int, Int, Int, Int) -> (s -> [a] -> [b] -> [c] -> [d] -> s)
947               -> Signal s -> Signal a -> Signal b -> Signal c -> Signal d -> Signal
      s
948 nextStateFSM4 _ _ NullS _ _ _ _ = NullS
949 nextStateFSM4 _ _ _ NullS _ _ _ = NullS
950 nextStateFSM4 _ _ _ _ NullS _ _ = NullS
951 nextStateFSM4 _ _ _ _ _ NullS _ = NullS
952 nextStateFSM4 _ _ _ _ _ _ NullS = NullS
953 nextStateFSM4 (c1, c2, c3, c4) f ss as bs cs ds
954   | c1 <= 0 || c2 <= 0 || c3 <= 0 || c4 <= 0
955     = error "nextStateFSM4: Number of consumed tokens must be positive integer"
956   | (not $ sufficient_tokens c1 as)
```

```
957       || (not $ sufficient_tokens c2 bs)
958       || (not $ sufficient_tokens c3 cs)
959       || (not $ sufficient_tokens c4 ds) = NullS
960     | otherwise = signal [next_state] +-+ nextStateFSM4 (c1, c2, c3, c4) f (tailS ss)
961                                           (dropS c1 as) (dropS c2 bs) (dropS c3 cs) (
          dropS c4 ds)
962     where consumed_tokens_as = fromSignal $ takeS c1 as
963           consumed_tokens_bs = fromSignal $ takeS c2 bs
964           consumed_tokens_cs = fromSignal $ takeS c3 cs
965           consumed_tokens_ds = fromSignal $ takeS c4 ds
966           current_state = headS ss
967           next_state = f current_state consumed_tokens_as
968                         consumed_tokens_bs consumed_tokens_cs consumed_tokens_ds
969
970
971 nextStateFSM5 :: (Int, Int, Int, Int, Int) -> (s -> [a] -> [b] -> [c] -> [d] -> [e]
        -> s)
972               -> Signal s -> Signal a -> Signal b -> Signal c -> Signal d -> Signal
          e -> Signal s
973 nextStateFSM5 _ _ NullS _ _ _ _ _ = NullS
974 nextStateFSM5 _ _ _ NullS _ _ _ _ = NullS
975 nextStateFSM5 _ _ _ _ NullS _ _ _ = NullS
976 nextStateFSM5 _ _ _ _ _ NullS _ _ = NullS
977 nextStateFSM5 _ _ _ _ _ _ NullS _ = NullS
978 nextStateFSM5 _ _ _ _ _ _ _ NullS = NullS
979 nextStateFSM5 (c1, c2, c3, c4, c5) f ss as bs cs ds es
980   | c1 <= 0 || c2 <= 0 || c3 <= 0 || c4 <= 0 || c5 <= 0
981     = error "nextStateFSM4: Number of consumed tokens must be positive integer"
982   | (not $ sufficient_tokens c1 as)
983     || (not $ sufficient_tokens c2 bs)
984     || (not $ sufficient_tokens c3 cs)
985     || (not $ sufficient_tokens c4 ds)
986     || (not $ sufficient_tokens c5 es) = NullS
987   | otherwise = signal [next_state] +-+ nextStateFSM5 (c1, c2, c3, c4, c5) f (tailS
          ss)
988                                         (dropS c1 as) (dropS c2 bs) (dropS c3 cs)
989                                         (dropS c4 ds) (dropS c5 es)
990   where consumed_tokens_as = fromSignal $ takeS c1 as
991         consumed_tokens_bs = fromSignal $ takeS c2 bs
992         consumed_tokens_cs = fromSignal $ takeS c3 cs
993         consumed_tokens_ds = fromSignal $ takeS c4 ds
994         consumed_tokens_es = fromSignal $ takeS c5 es
995         current_state = headS ss
996         next_state = f current_state consumed_tokens_as
997                       consumed_tokens_bs consumed_tokens_cs
998                       consumed_tokens_ds consumed_tokens_es
999
1000
1001 outputFSM :: (s -> (Int, [a])) -> Signal s -> Signal a
1002 outputFSM _ NullS = NullS
1003 outputFSM g (s:-ss)
1004   | length y1 /= p = error "outputFSM: Incorrect number of produced tokens."
1005   | otherwise = signal y1 +-+ outputFSM g ss
1006   where (p, y1) = g s
1007
1008
1009 outputFSM2 :: (s -> ((Int, Int), ([a], [b]))) -> Signal s -> (Signal a, Signal b)
1010 outputFSM2 _ NullS = (NullS, NullS)
1011 outputFSM2 g (s:-ss)
1012   | length y1 /= p1 || length y2 /= p2 = error "outputFSM2: Incorrect number of
          produced tokens."
1013   | otherwise = (signal y1 +-+ yr1, signal y2 +-+ yr2)
1014   where ((p1, p2), (y1, y2)) = g s
1015         (yr1, yr2) = outputFSM2 g ss
1016
1017
1018 outputFSM3 :: (s -> ((Int, Int, Int), ([a], [b], [c])))
1019            -> Signal s -> (Signal a, Signal b, Signal c)
1020 outputFSM3 _ NullS = (NullS, NullS, NullS)
1021 outputFSM3 g (s:-ss)
1022   | length y1 /= p1
```

```
1023      || length y2 /= p2
1024      || length y3 /= p3 = error "outputFSM3: Incorrect number of produced tokens."
1025    | otherwise = (signal y1 +-+ yr1, signal y2 +-+ yr2, signal y3 +-+ yr3)
1026    where ((p1, p2, p3), (y1, y2, y3)) = g s
1027          (yr1, yr2, yr3) = outputFSM3 g ss
1028
1029
1030  outputFSM4 :: (s -> ((Int, Int, Int, Int), ([a], [b], [c], [d])))
1031             -> Signal s -> (Signal a, Signal b, Signal c, Signal d)
1032  outputFSM4 _ NullS = (NullS, NullS, NullS, NullS)
1033  outputFSM4 g (s:-ss)
1034    | length y1 /= p1
1035      || length y2 /= p2
1036      || length y3 /= p3
1037      || length y4 /= p4 = error "outputFSM4: Incorrect number of produced tokens."
1038    | otherwise = (signal y1 +-+ yr1, signal y2 +-+ yr2, signal y3 +-+ yr3, signal y4
          +-+ yr4)
1039    where ((p1, p2, p3, p4), (y1, y2, y3, y4)) = g s
1040          (yr1, yr2, yr3, yr4) = outputFSM4 g ss
1041
1042
1043  outputFSM5 :: (s -> ((Int, Int, Int, Int, Int), ([a], [b], [c], [d], [e])))
1044             -> Signal s -> (Signal a, Signal b, Signal c, Signal d, Signal e)
1045  outputFSM5 _ NullS = (NullS, NullS, NullS, NullS, NullS)
1046  outputFSM5 g (s:-ss)
1047    | length y1 /= p1
1048      || length y2 /= p2 || length y3 /= p3
1049      || length y4 /= p4 || length y5 /= p5 = error "outputFSM5: Incorrect number of
          produced tokens."
1050    | otherwise = (signal y1 +-+ yr1, signal y2 +-+ yr2, signal y3 +-+ yr3,
1051                   signal y4 +-+ yr4, signal y5 +-+ yr5)
1052    where ((p1, p2, p3, p4, p5),(y1, y2, y3, y4, y5)) = g s
1053          (yr1, yr2, yr3, yr4, yr5) = outputFSM5 g ss
```

A.3 RISC processor

```
1   ------------------------------------------------------------------------------
2   -- |
3   -- Module  :  ProSyDe_v2
4   -- Copyright   :  (c) Ricardo Bonna
5   -- License     :  still needs license
6   --
7   -- Maintainer  :  ricardobonna@gmail.com
8   -- Stability   :  experimental
9   -- Portability :  portable
10  --
11  -- This is the second version of the ProSyDe processor featuring a smaller
12  -- and more abstract concept. This makes easier to implement new operations.
13  --
14  ------------------------------------------------------------------------------
15
16  module ProSyDe_v2 (
17    procNet
18    ) where
19
20  import ForSyDe.Shallow
21  import Data.Bits((.&.), (.|.), xor)
22
23
24  ------------------------------------------------------
25  -- Instruction Fetch (IF) Kernel
26  ------------------------------------------------------
27
```

```
28 type ScenarioIF = ((Int, Int, Int), (Int, Int, Int, Int), [Int] -> [Int]
29                 -> [Vector String] -> ([[Int]], [String], [Int], [Vector String]))
30
31 -- | 'ifScenario' is the list of possible scenarios for the IF kernel
32 ifScenario :: Int -> ScenarioIF
33 ifScenario n
34   | n > 20 || n < 0   = error "ifScenario: Non existent scenario"
35   | n >= 14 && n < 19 = ((1,1,1), (1,1,1,1), \[a] [pc] [m] -> ([arg m (pc+a)], [op m
       (pc+a)], [pc+a+1], [m])) -- branch
36   | otherwise         = ((0,1,1), (1,1,1,1), \_ [pc] [m] -> ([arg m pc], [op m pc],
     [pc+1], [m]))             -- no branch
37
38 -- | 'slice' takes a string containing one assembly command and slice it into
39 -- one string with the opcode and a list of Ints containing the arguments
40 slice :: String -> (String, [Int])
41 slice a = (head $ words a, args)
42   where rest = tail $ words a
43         args
44           | null rest = []
45           | length rest == 1 = [read (head rest) :: Int]
46           | length rest == 2 = [read (head rest) :: Int, read (last rest) :: Int]
47           | otherwise = error "slice: Some instruction has more than 2 arguments"
48
49 arg :: (Eq a, Num a) => Vector String -> a -> [Int]
50 arg x y = snd $ slice (atV x y)
51
52 op :: (Eq a, Num a) => Vector String -> a -> String
53 op x y = fst $ slice (atV x y)
54
55 -- | 'ifKernel' is the Instruction Fetch (IF) kernel process
56 ifKernel :: Vector String -> Signal ScenarioIF -> Signal Int -> (Signal [Int],
       Signal String)
57 ifKernel progV ifCt sigBr = (sigArg, sigOp)
58   where (sigArg, sigOp, sigPc, sigPm) = kernel34SADF ifCt sigBr sigPc' sigPm'
59         sigPc' = delaySADF [0] sigPc
60         sigPm' = delaySADF [progV] sigPm
61
62
63 ------------------------------------------------------------
64 -- Execute (EXE) Kernel
65 ------------------------------------------------------------
66
67 type ScenarioEXE = ((Int, Int, Int), (Int, Int, Int, Int), [[Int]] -> [Vector Int]
68                  -> [Vector Int] -> ([Int], [Int], [Vector Int], [Vector Int]))
69
70 -- | 'exeScenario' is the list of possible scenarios for the EXE kernel
71 exeScenario :: Int -> ScenarioEXE
72 exeScenario 0  = ((1,0,0), (0,0,0,0), \_ _ _ -> ([], [], [], []))
                                          -- nop
73 exeScenario 1  = ((1,1,1), (0,0,1,1), \[[rd, mn]] [r] [m] -> ([], [], [replaceV r rd
       (atV m mn)], [m]))                 -- ld
74 exeScenario 2  = ((1,1,1), (0,0,1,1), \[[rd, rm]] [r] [m] -> ([], [], [replaceV r rd
       (atV m (atV r rm))], [m]))         -- ldr
75 exeScenario 3  = ((1,1,1), (0,0,1,1), \[[rs, mn]] [r] [m] -> ([], [], [r], [replaceV
       m mn (atV r rs)]))                 -- st
76 exeScenario 4  = ((1,1,1), (0,0,1,1), \[[rs, rm]] [r] [m] -> ([], [], [r], [replaceV
       m (atV r rm) (atV r rs)]))         -- str
77 exeScenario 5  = ((1,1,0), (0,0,1,0), \[[rd, rs]] [r] _ -> ([], [], [replaceV r rd (
      atV r rs)], []))                    -- mov
78 exeScenario 6  = ((1,1,0), (0,0,1,0), \[[rd, i]] [r] _ -> ([], [], [replaceV r rd i
      ], []))                              -- movi
79 exeScenario 7  = ((1,1,0), (0,0,1,0), \[[rd, rs]] [r] _ -> ([], [], [replaceV r rd
      ((atV r rd) + (atV r rs))], []))     -- add
80 exeScenario 8  = ((1,1,0), (0,0,1,0), \[[rd, rs]] [r] _ -> ([], [], [replaceV r rd
      ((atV r rd) - (atV r rs))], []))     -- sub
81 exeScenario 9  = ((1,1,0), (0,0,1,0), \[[rd, rs]] [r] _ -> ([], [], [replaceV r rd
      ((atV r rd) * (atV r rs))], []))     -- mul
82 exeScenario 10 = ((1,1,0), (0,0,1,0), \[[rd, rs]] [r] _ -> ([], [], [replaceV r rd (
      div (atV r rd) (atV r rs))], [])) -- div
83 exeScenario 11 = ((1,1,0), (0,0,1,0), \[[rd, rs]] [r] _ -> ([], [], [replaceV r rd
      ((atV r rd) .&. (atV r rs))], [])) -- and
```

```
84  exeScenario 12 = ((1,1,0), (0,0,1,0), \[[rd, rs]] [r] _ -> ([], [], [replaceV r rd
        ((atV r rd) .|. (atV r rs))], [])) -- or
85  exeScenario 13 = ((1,1,0), (0,0,1,0), \[[rd, rs]] [r] _ -> ([], [], [replaceV r rd (
        xor (atV r rd) (atV r rs))], [])) -- xor
86  exeScenario 14 = ((1,1,0), (1,0,1,0), \[[rs, v]] [r] _ -> ([if atV r rs == 0 then v
        else 0], [], [r], []))            -- bez
87  exeScenario 15 = ((1,1,0), (1,0,1,0), \[[rs, v]] [r] _ -> ([if atV r rs /= 0 then v
        else 0], [], [r], []))            -- bnz
88  exeScenario 16 = ((1,1,0), (1,0,1,0), \[[rs, v]] [r] _ -> ([if atV r rs > 0 then v
        else 0], [], [r], []))             -- bgz
89  exeScenario 17 = ((1,1,0), (1,0,1,0), \[[rs, v]] [r] _ -> ([if atV r rs < 0 then v
        else 0], [], [r], []))             -- blz
90  exeScenario 18 = ((1,0,0), (1,0,0,0), \[[v]] _ _ -> ([v], [], [], []))
                                           -- jmp
91  exeScenario 19 = ((1,1,0), (0,1,1,0), \[[rs]] [r] _ -> ([], [atV r rs], [r], []))
                                           -- outr
92  exeScenario 20 = ((1,0,1), (0,1,0,1), \[[mn]] _ [m] -> ([], [atV m mn], [], [m]))
                                           -- outm
93  exeScenario _  = error "exeScenario: Non existent scenario"
94
95  -- | 'exeKernel' is the Execution (EXE) kernel process
96  exeKernel :: Signal ScenarioEXE -> Signal [Int] -> (Signal Int, Signal Int)
97  exeKernel exeCt sigArg = (sigBr, sigDmp)
98    where (sigBr, sigDmp, sigReg, sigDm) = kernel34SADF exeCt sigArg sigReg' sigDm'
99          sigReg' = delaySADF [regV] sigReg
100         sigDm' = delaySADF [memV] sigDm
101
102 regV = vector $ replicate 32 0
103 memV = vector $ replicate 1024 0
104
105 ------------------------------------------------------------
106 -- Decode Detector
107 ------------------------------------------------------------
108
109 type ScenarioDEC = ((Int,Int), ([ScenarioIF], [ScenarioEXE]))
110
111 type StateDEC = Int
112
113 -- | 'detectorScenario' is th output function of the detector. It converts a
114 -- state into an equivalent scenario
115 decScenario :: StateDEC -> ScenarioDEC
116 decScenario n = ((1,1), ([ifScenario n], [exeScenario n]))
117
118 -- | 'decSwitchState' is the state transition function of the detector
119 decSwitchState :: StateDEC -> [String] -> StateDEC
120 decSwitchState _ ["nop"]  = 0
121 decSwitchState _ ["ld"]   = 1
122 decSwitchState _ ["ldr"]  = 2
123 decSwitchState _ ["st"]   = 3
124 decSwitchState _ ["str"]  = 4
125 decSwitchState _ ["mov"]  = 5
126 decSwitchState _ ["movi"] = 6
127 decSwitchState _ ["add"]  = 7
128 decSwitchState _ ["sub"]  = 8
129 decSwitchState _ ["mul"]  = 9
130 decSwitchState _ ["div"]  = 10
131 decSwitchState _ ["and"]  = 11
132 decSwitchState _ ["or"]   = 12
133 decSwitchState _ ["xor"]  = 13
134 decSwitchState _ ["bez"]  = 14
135 decSwitchState _ ["bnz"]  = 15
136 decSwitchState _ ["bgz"]  = 16
137 decSwitchState _ ["blz"]  = 17
138 decSwitchState _ ["jmp"]  = 18
139 decSwitchState _ ["outr"] = 19
140 decSwitchState _ ["outm"] = 20
141 decSwitchState _ _ = error "decSwitchState: Input not recognized"
142
143 -- | 'decodeDetector' is the detector of ProSyDe
144 decDetector :: Signal String -> (Signal ScenarioIF, Signal ScenarioEXE)
145 decDetector = detector12SADF 1 decSwitchState decScenario 0
```

```
146
147
148 ------------------------------------------------------------
149 -- ProSyDe compact
150 ------------------------------------------------------------
151
152 -- | 'prosyde' is the processor built with the ForSyDe SADF library. It outputs
153 -- whatever it is in register @rs@ during the execution of instruction @outr rs@
154 procNet :: Vector String -> Signal Int
155 procNet progV = sigDmp
156   where (sigBr, sigDmp) = exeKernel exeCt sigArg
157         (sigArg, sigOp) = ifKernel progV ifCt' sigBr
158         (ifCt, exeCt) = decDetector sigOp
159         ifCt' = delaySADF [ifScenario 0] ifCt
160
161
162 ------------------------------------------------------------
163 -- Program Code
164 ------------------------------------------------------------
165
166 progV :: Vector String
167 progV = vector [
168   "movi 0 1",    -- reg 0 = 1
169   "movi 1 1",    -- reg 1 = 1
170   "movi 2 100", -- reg 2 = 100
171   "add 0 1",     -- reg 0 = reg 0 + reg 1
172   "sub 2 0",     -- reg 2 = reg 2 - reg 0
173   "bez 2 1",     -- skip next instruction if reg 2 == 0
174   "jmp -5",      -- jump back to instruction 3: "movi 2 100"
175   "str 0 0",     -- mem(reg 0) = reg 0
176   "ldr 10 0",    -- reg 10 = mem(reg 0)
177   "outr 10",     -- output reg 10
178   "outr 1",      -- output reg 1
179   "outm 100",    -- output mem(100)
180   "movi 10 7",   -- reg 10 = 7
181   "st 10 42",    -- mem(42) = reg 10
182   "outm 42",     -- output mem(42)
183   "jmp -1"       -- end of program
184   ]
```