UNICAMP

UNIVERSIDADE ESTADUAL DE CAMPINAS

FACULDADE DE ENGENHARIA MECÂNICA

Augusto Yoshio Horita

Automation Approaches for Embedded Systems Design Flows Based on Formal Models of Computation

Estratégias de Automação para Desenvolvimento de Projetos de Sistemas Embarcados Baseados em Modelos Formais de Computação

CAMPINAS

2019

Augusto Yoshio Horita

Automation Approaches for Embedded Systems Design Flows Based on Formal Models of Computation

Estratégias de Automação para Desenvolvimento de Projetos de Sistemas Embarcados Baseados em Modelos Formais de Computação

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

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

Orientador: Prof. Dr. Denis Silva Loubach

CAMPINAS

2019

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

H782a

Horita, Augusto Yoshio, 1987-
Automation approaches for embedded systems design flows based on formal models of computation / Augusto Yoshio Horita. – Campinas, SP : [s.n.], 2019.

Orientador: Denis SIlva Loubach.
Dissertação (mestrado) – Universidade Estadual de Campinas, Faculdade de Engenharia Mecânica.

1. Sistemas embarcados (Computadores). 2. Computabilidade e modelos de computação. I. Loubach, Denis Silva, 1982-. II. Universidade Estadual de Campinas. Faculdade de Engenharia Mecânica. III. Título.

Informações para Biblioteca Digital

Título em outro idioma: Estratégias de automação para desenvolvimento de projetos de sistemas embarcados baseados em modelos formais de computação
Palavras-chave em inglês:
Embedded systems (Computers)
Computability and models of computation
Área de concentração: Mecatrônica
Titulação: Mestre em Engenharia Mecânica
Banca examinadora:
Denis Silva Loubach
Eurípedes Guilherme de Oliveira Nóbrega
Romis Ribeiro de Faissol Attux
Data de defesa: 17-10-2019
Programa de Pós-Graduação: Engenharia Mecânica

Identificação e informações acadêmicas do(a) aluno(a)
- ORCID do autor: https://orcid.org/0000-0003-2805-8530
- Currículo Lattes do autor: http://lattes.cnpq.br/1738569966447609

UNIVERSIDADE ESTADUAL DE CAMPINAS
FACULDADE DE ENGENHARIA MECÂNICA
COMISSÃO DE PÓS-GRADUAÇÃO EM ENGENHARIA MECÂNICA
DEPARTAMENTO DE MECÂNICA COMPUTACIONAL

DISSERTAÇÃO DE MESTRADO ACADÊMICO

Automation Approaches for Embedded Systems Design Flows Based on Formal Models of Computation

Estratégias de Automação para Desenvolvimento de Projetos de Sistemas Embarcados Baseados em Modelos Formais de Computação

Autor: Augusto Yoshio Horita
Orientador: Prof. Dr. Denis Silva Loubach

A banca examinadora composta pelos membros abaixo aprovou esta dissertação:

Prof. Dr. Denis Silva Loubach, Presidente
Departamento de Sistemas de Computação/ITA

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

Prof. Dr. Romis Ribeiro de Faissol Attux
FEEC/Unicamp

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, 17 de Outubro de 2019

This work is dedicated to my wife and kids, who have inspired and supported me.

Acknowledgements

First, I thank God for all the blessings I have got, including wonderful people around me, besides the opportunities and virtues I was given to which made this work possible.

I thank Prof. Dr. Denis Silva Loubach, who has always been attentive and patient with my difficulties, giving me all the support and sharing his expertise that led me to finish this work. Besides, I thank Ricardo Bonna for promptly sharing his knowledge whenever I needed.

I thank Unicamp for affording me the opportunity to complete my MSc. study here.
I thank my parents, brother and sister, who are part of my being, teaching me how to be better and never give up.

I deeply thank my wife, for always giving me the necessary support and being by my side.
I thank my children for giving me the inspiration to be a better person.

I humbly extend my thanks to all my friends and persons who co-operated in any manner with me in this work.

Resumo

Sistemas embarcados de alta performance estão presentes em cada vez mais áreas de aplicação. Com o aumento da complexidade, se torna mais difícil atender ao requisito de se projetar o sistema mais otimizado utilizando menos recursos. Nesse contexto, os métodos de projeto de sistemas embarcados baseados em modelos formais têm sido estudados para tornar esse processo mais robusto e escalável. O uso de modelos de computação (MoC), que consistem na modelagem de uma aplicação utilizando um alto nível de abstração com base formal, possibilita uma análise sistemática do sistema antes de sua implementação. Ferramentas e frameworks têm sido desenvolvidos para a modelagem baseada em MoCs. Algumas dessas ferramentas suportam a simulação dos modelos, possibilitando a verificação das funcionalidades do sistema antes das próximas fases do projeto. O aumento do nível de abstração, proporcionado pelo uso dos MoCs, dificulta a fase de implementação pela falta de detalhes nos modelos de alto nível de abstração. Nesse sentido, esta pesquisa tem como objetivo identificar possíveis estratégias de automação para o desenvolvimento de sistemas embarcados baseado em modelos formais de computação.

Palavras-chave: Sistemas embarcados, Modelos de computação, Projeto baseado em modelos formais, Modelagem, Simulação, Programação funcional.

Abstract

Sophisticated and high performance embedded systems are present in an increasing number of application domains. As the complexity grows, it gets harder to satisfy the requirement of getting the most optimized system using less development resources. In this context, formal-based design methods have been studied to make the development process robust and scalable, using the correct-by-construction approach. Models of computation (MoC), which consists on modeling an application at a high abstraction level by using a formal base, enables a systematic application analysis before its implementation. Different tools and frameworks have been developed supporting MoCs. Some of them can simulate the models and also verify its functionality and feasibility before the next design steps. As MoC elevates the abstraction level, the implementation steps get more complex, creating an abstraction gap. In view of this, the present research aims to identify possible automation approaches for embedded systems design flows.

Keywords: Embedded systems, Models of computation, Formal-based design flow, Modeling, Simulation, Functional programming.

List of Figures

List of Tables

List of Symbols and Abbreviations

AIPAA	Analysis and Identification of Possible Automation Approaches
ALU	Arithmetic Logic Unit
ARM	Advanced RISC Machine
CPS	Cyber-Physical Systems
CSDF	Cycle-Static Dataflow
CSP	Communicationg Sequential Processes
CT	Continuous Time
DSL	Domain Specific Language
EDS	Encoder-Decoder System
EDSL	Embedded DSL
ForSyDe	Formal System Design
FPGA	Field Programmable Gate Array
FPU	Floating Point Unit
FSM	Finite State Machine
GHC	Glasgow Haskell Compiler
GHCi	GHC interactive environment
GPU	Graphics Processing Unit
GUI	Graphical User Interface
HDL	Hardware Description Language
iCyPhy	Industrial Cyber-Physical Center
IEEE	Institute of Electrical and Electronics Engineers
MoC	Model of Computation
MoML	Modeling Markup Language
NASA	National Aeronautics and Space Administration

OSMC	Open Source Modelica Consortium
PASS	Periodic Admissible Sequential Schedule
R2D2C	Requirements to Design Code
RISC	Reduced Instruction Set Computer
SADF	Scenario-Aware Dataflow
SDK	Software Development Kit
SDF	Synchronous Dataflow
SDF3	Synchronous Dataflow For Free
SOL	Sequential Object Language
SR	Synchronout Reactive
SY	Synchronous
TSM	Tagged Signal Model
XML	eXtensible Markup Language

Contents

1 INTRODUCTION

Embedded systems are present in a growing number of different application areas, which includes a wide complexity range, from simple wearable gadgets to aerospace and biomedical. Power consumption, performance, and cost usually figure as key constraints to these systems. *Real-time embedded systems*, specifically, have as a critical requirement predictable and deterministic response time. A design error in these cases can cost whole projects or even lives (Buttazzo, 2011).

Besides the growing number of embedded systems applications, their integration and connectivity, aiming to improve control and monitoring methods, have created the concept of *cyber-physical systems* (CPS), which represents the integration of computation and physical processes controlled by embedded computers and its networks, generally, by using feedback loops. Therefore, computation and physical systems affect one another (Lee, 2010).

As embedded systems and CPS complexities grow, it gets harder and harder to specify, model and simulate them, therefore making the system implementation phase more complex. In this sense, *formal-based design methods* have been developed to make this process reliable, robust and scalable together with *design space exploration* (DSE).

Towards the *correct-by-construction* development, optimizing the available resources, Edwards et al. (1997) argue that systems should be implemented at a high abstraction level, using formal models. This implementation consists on an executable model which makes no references to implementation code or platforms. In this context, *models of computation* (MoC) are presented as a key approach to formal-based system modeling and simulation. There is a range of existing MoCs, some illustrated in Fig. 2.10, where each one represents and captures different aspects and semantics of system's functionalities. Therefore, one should carefully choose the MoC to use based on the type of application being modeled, besides the modeling and simulation methods (Jantsch and Sander, 2005).

Considering the design phase, a range of frameworks were developed aiming to aid the modeling and simulation of systems on a formal base. Examples are Ptolemy II (Ptolemaeus, 2014), ForSyDe (Sander et al., 2016), SDF3 (Stuijk et al., 2006), and Simulink (MathWorks, 2019a). Each framework focuses on different design methodologies aspects and has its own profits and drawbacks when compared to the others.

Although formal models with a high abstraction level have advantages, making possible the early detection of inconsistencies and ambiguities in the specification model step, they have a counter-part known as *abstraction gap*, caused by its absence of implementation details. This leads to a wider *design space* possibilities, defined as the range of implementations options related to the initial model. Fig. 1.1 illustrates the synthesis process and that abstraction gap (Sander, 2003).

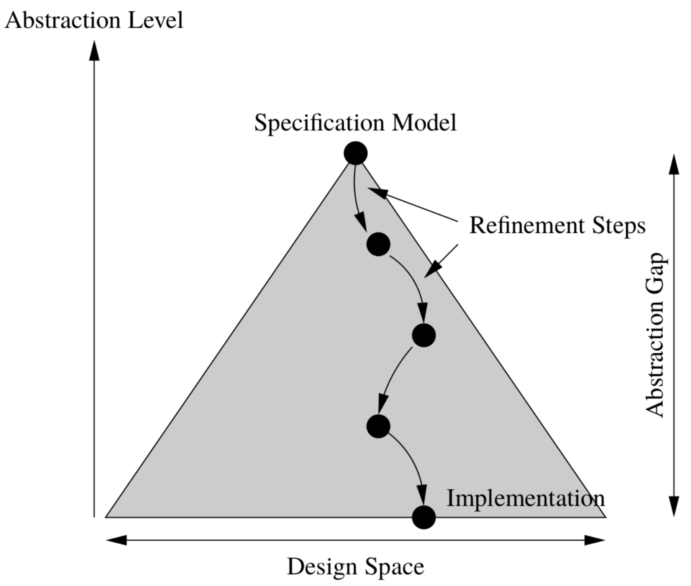


Figure 1.1. Synthesis process from specification model to system implementation (Sander, 2003).

Research works are continuously presenting methods to optimize and overcome the abstraction gap, composing the *design space exploration* concept (Sinaei and Fatemi, 2016; Li et al., 2017).

1.1 Research Objective

This research work objective is to **identify possible automation approaches for design flows based on formal models of computation**, aiming to assist in a *future implementation* of automatic code generation and also a trustable, robust and scalable embedded systems design flow.

1.2 Research Scope

The present research considers both the *design* and the *high level implementation*, *i.e.,* in the specification domain, of an embedded system case study, following formal design methods to model and simulate the system.

Basically, two types of MoCs with different timing abstraction, as discussed in (Horita et al., 2019a), are used here. One is the *synchronous* (SY) MoC, representing the timed classification of MoCs, and the other is the *synchronous dataflow* (SDF), representing the untimed. A third MoC, named *scenario-aware dataflow* (SADF) is also addressed in the present work. SADF is a generalization of SDF to model dynamic systems.

Based on the defined high level abstraction models and their simulation and high level implementations, it should be possible to identify which are the main intermediate steps candidate to be automated.

1.3 Research Requirements

This research work addresses the following requirements:

- R_0 Literature review with respect to the main concepts and theory on MoC and high level modeling;
- R_1 Complete specification of embedded systems case studies;
- R_2 A formal-based modeling framework selection criteria and list of candidates tools;
- R_3 Embedded system case study modeling and high level implementation, based on formal design methods; and
- R_4 An analysis identifying implementation steps to be automated.

1.4 Expected Results

The expected results are the formal-based modeling, simulation and high level implementation of case studies. Besides that, this research aims to present an analysis of refinement steps taken during the design space exploration, pointing out possible automation approaches in the design flow, that can be implemented as a future works.

The validation of this research will be based on the verification of the correct manual system implementation, following the selected MoC semantics.

1.5 Document Structure

The remainder of this document is organized as follows. Chapter 2 presents a background on the main concepts used in this work. In Chapter 3, it is described a proposed method for possible automation identification together with an illustrative example. Next, Chapter 4 shows the implementation of a case study based on the introduced method, demonstrating its potential and applicability. Chapter 5 introduces a discussion regarding the case study development and results. Finally, Chapter 6 summarizes the research conclusions, contributions and possible future works.

2 BACKGROUND

This chapter presents the main concepts involved in this research, including embedded and cyber-physical systems, models of computation, functional programming paradigm and automatic code generation, together with related works.

2.1 Embedded and Cyber-Physical Systems

Embedded systems can be defined as computing systems designed to perform a dedicated function, in which hardware and software are tightly coupled (Li and Yao, 2003). One of the main components of these systems is the processing unit, *i.e.,* microcontrollers or microprocessors, in which application instructions are executed. As the technology advances, different processor's architectures are developed, optimizing their performance by increasing their capabilities, *e.g.,* runtime reconfiguration, or resources, *e.g.,* multiple processing cores.

Despite those performance improvements, a range of complex applications requires embedded system's networks to monitor and control physical variables or processes based on logical computational algorithms, usually using feedback loops. This intersection of cyber and physical components is named *cyber-physical systems* (CPS). It combines engineering models and methods from mechanical, electrical, and chemical engineering with models and methods from computer science (Lee, 2015). CPS applications include modern transportation systems, security systems, and distributed robotics, for instance.

A possible CPS structure is illustrated in Fig. 2.1, representing a network of three platforms composed by actuators, sensors and embedded computers, as *computation* blocks. As an example, (Lee, 2010) describes it as an automation application, in which controllers for high-speed printing presses are modeled as the actuators and sensors represent disruptions detection. The control algorithms, modeled as computation blocks, handle rapid shutdown modes to prevent damage to the equipment in case of paper jams.

As the applications complexity grows, the number of components included in the CPS network and the computational algorithms can exponentially increase. In a scalable perspective, the greater and more complex the CPS gets, the harder it is to specify, simulate and implement these systems. In view of this, formal-based design methodologies are adopted aiming the correct-by-construction development.

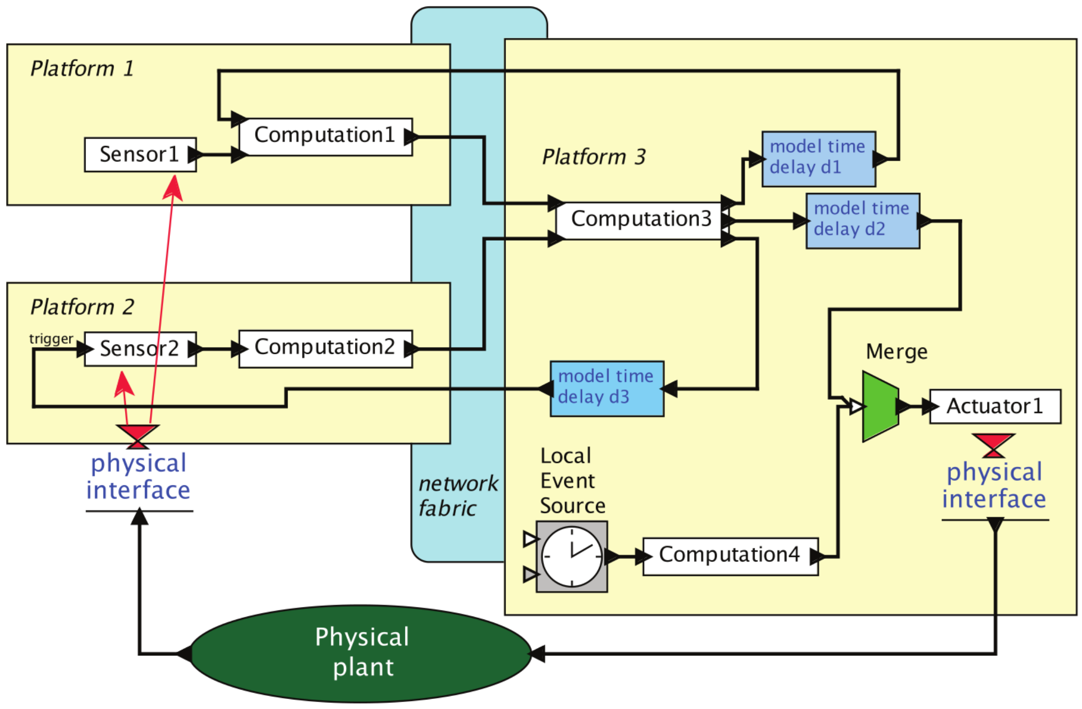


Figure 2.1. A possible CPS structure (Lee, 2010).

2.2 Models of Computation (MoC)

According to Jantsch (2003), a *model* is a simplification of another entity, which can be a physical entity or even another model. It describes only the characteristics that are relevant for a given task.

In the present research context, formal models of computation (MoCs) are used to model embedded systems. These MoCs elevate the abstraction level of the modeled systems, making no reference to implementation platforms or implementation languages, but capturing functional behaviors regarding communication, synchronization and processes interactions (Fernández, 2009; Jantsch, 2005). Essentially, MoCs are collections of *abstract rules* that dictate the semantics of execution and concurrency in heterogeneous computational systems.

2.2.1 Tagged Signal Model (TSM)

As MoCs abstract the system functionality, which are relevant for a specific task, a range of different models have been developed. As a consequence, the selection of a MoC is not a trivial design step. For that reason, many researches have presented different comparison methods and MoC's classification (University of California, Berkeley-online, 2018; Lee and Sangiovanni-

Vincentelli, 1998; Paul and Thomas, 2005).

In this sense, Lee and Sangiovanni-Vincentelli (1998) presented a meta-model/framework named *tagged signal model* (TSM) for reasoning about MoCs definitions and properties. A range of MoCs, studies and frameworks share the main concepts contained in that research work (University of California, Berkeley-online, 2018; Sander, 2002; Jantsch, 2003). Within TSM, systems are regarded as a collection of processes. A process communicates through signals, composed by events, having the following definitions.

Definition 1 (Event) *An event* $\mathtt{e}$*, is an elementary unit of information composed by a tag* $\mathtt{t_i} \in \mathtt{T}$ *and a value* $\mathtt{v_i} \in \mathtt{V}$.

Definition 2 (Signal) *A signal* $\mathtt{s}$*, belonging to the set of signals* $\mathtt{S}$*, is a set of events* $\mathtt{e_i} = (\mathtt{t_i}, \mathtt{v_i})$*, responsible for processes communication.*

Definition 3 (Process) *A process* $\mathtt{P}$ *is a set of possible behaviors, and can be viewed as relations between input signals* $\mathtt{S^I}$ *and output signals* $\mathtt{S^O}$*. The set of output signals is given by the intersection between the input signals and the process* $\mathtt{S^O} = \mathtt{S^I} \cap \mathtt{P}$*. A process is functional when there is a single value mapping* $\mathtt{f} : \mathtt{S^I} \to \mathtt{S^O}$ *which describes it. Therefore, a functional process has either one behavior or no behavior at all.*

TSM classifies the MoCs within two categories, *timed* and *untimed*, which are described as follows.

2.2.2 Timed Models of Computation

In a *timed* MoC, the set of tags T is *totally ordered*. For that reason, the tags are also considered the *timestamp* of each event. As a consequence, it is possible to order every event in every signal of the MoC based on its tag. As examples of timed MoCs, TSM presents continuous time, discrete-event, synchronous and sequential systems (Jantsch, 2003).

The synchronous MoC was one of the first to be defined and also is a widely used timed MoC. For those reasons, it is object of study in the present research.

Synchronous (SY) MoC

The synchronous MoC is based on the *perfect synchrony hypothesis*, which states that neither computation nor communication takes time. The time is abstracted by dividing its axis into slots. Everything that happens inside a specific slot occurs synchronized by a global time clock

(Jantsch, 2003). For a system to attend the perfect synchrony hypothesis, the time slot must be selected in a manner that all the model's process are able to respond *fast enough*, *i.e.,* inside the same time slot.

Fig. 2.2 represents a SY process, composed by two sub-process. The signals are composed by the events u_x and v_x, where x is the tag of these events, which in turn are inputs to the `Process P`. The `Process P1` sub-process consumes u_x and v_x, producing the intermediate signals composed by u'_x and v'_x. The last are consumed by `Process P2`, generating the `Process P` output signals composed by the events u''_x and v''_x. According to synchrony hypothesis, v'_1 and v''_1 are outputted at the same instant, *i.e.,* the first time slot or first execution cycle of the system. The next events of each signal also follow the same behavior.

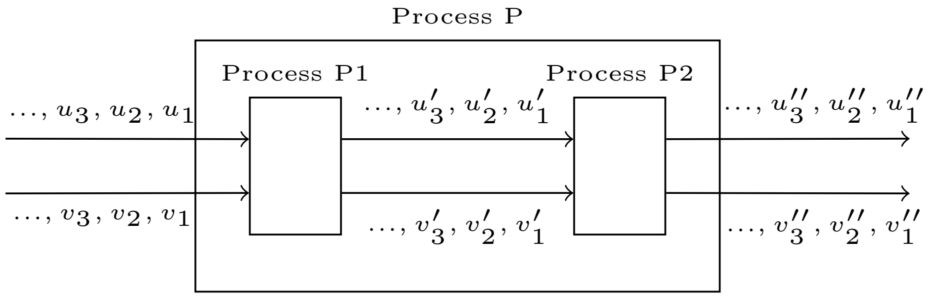

Process P
Process P1
Process P2
$\ldots, u_3, u_2, u_1$
$\ldots, v_3, v_2, v_1$
$\ldots, u'_3, u'_2, u'_1$
$\ldots, v'_3, v'_2, v'_1$
$\ldots, u''_3, u''_2, u''_1$
$\ldots, v''_3, v''_2, v''_1$

Figure 2.2. SY process representation, adapted from (Jantsch, 2003).

A wide range of systems can be modeled using synchronous MoC due to its behavior, in which the processes read inputs, compute outputs and communicate with other processes, *e.g.,* reactive systems and CPS composed by sensors and actuators (Lee and Sangiovanni-Vincentelli, 1998).

2.2.3 Untimed Models of Computation

In an *untimed* MoC, the set of tags T are *partially ordered*, denoting causality or synchronization, as a consequence, only local groups of events can be ordered based on their tags, rather than all set T. Some examples of untimed MoCs are Kahn process networks (KPN), dataflows and Petri Nets (Lee and Sangiovanni-Vincentelli, 1998).

The dataflows can be divided into a range of MoCs. They are represented by directed graphs, where each *node* represents a process and the *arcs* represent the communication paths. A process can only execute, *i.e.,* fire, if it has the necessary events, *i.e., tokens*, available in all of its input ports.

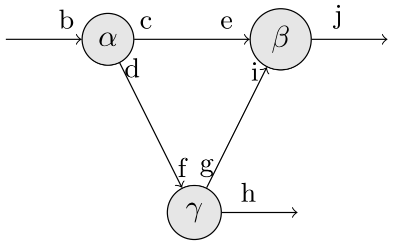
One special case of dataflow is the synchronous dataflow (SDF). It was first presented in 1987 (Lee and Messerschmitt, 1987) and is widely used due to its performance characteristics, as the inputs and outputs rates are defined at compile time, and its modeling simplicity compared to others MoCs from the dataflows family.

Synchronous Dataflow (SDF) MoC

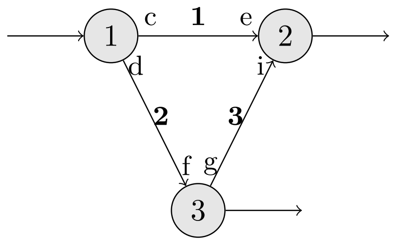
SDF MoC was first presented as dataflows graphs composed by synchronous nodes (Lee and Messerschmitt, 1987). A node is defined as *synchronous* if the number of tokens consumed by the input ports and produced by the output ports are constant and possible to be defined at compile time.

The SDF graph must be non-terminating, which means that the model must be able to run without any deadlock. To check if a model is consistent, it is possible to perform a formal analysis based on the *periodic admissible sequential schedule* (PASS).

Fig. 2.3a illustrates how a system may be modeled using the SDF MoC graph. To prove the existence of the system PASS, it is necessary to assemble its topology matrix, in which the (x,y) entry represents the amount of data produced by the node y on the arc x. Fig. 2.3b identifies the arcs and nodes of the example system and the Eq. (2.1) represents its topology matrix.



(a) Regular representation of SDF graph.



(b) SDF with identified nodes and paths.

Figure 2.3. SDF example, adapted from (Lee and Messerschmitt, 1987).

$$\Gamma = \begin{bmatrix} c & -e & 0 \\ d & 0 & -f \\ 0 & i & -g \end{bmatrix} \tag{2.1}$$

A set of lemmas and equations (Lee and Messerschmitt, 1987) demonstrates a necessary condition for a system model containing s nodes to hold the PASS:

$$rank(\Gamma) = s - 1 \tag{2.2}$$

The SDF MoC is suitable to model streaming processing due to the possibility to previously define the periodic processing cycle and the token rate of the input and output processes ports.

2.2.4 Hybrid Models

As the CPS complexity grows, it comprehends more processes and variables that do not follow the same semantics when compared to each other, composing the concept of heterogeneous systems. As a consequence, it is not possible to model the whole system using only one MoC, as each MoC represents only one aspect of the entire system (Eker et al., 2003). In this sense, frameworks and modeling domain specific language (DSL) included tools to model these systems by enabling the use of multiple MoCs in one single model, generating the *hybrid models* concept.

A typical example of hybrid models application is the *modal model*, in which the top level represents a finite state machine (FSM) and each state is refined into models that can be driven by different MoCs. An example is illustrated in Fig. 2.4, in which the states are represented by *stt* nodes, the sub-models by *MoC* nodes, and environment inputs by *In*. This type of model can be used with systems that dynamically changes it behavior depending on user or environment inputs, such as sensor data (Lee and Tripakis, 2010).

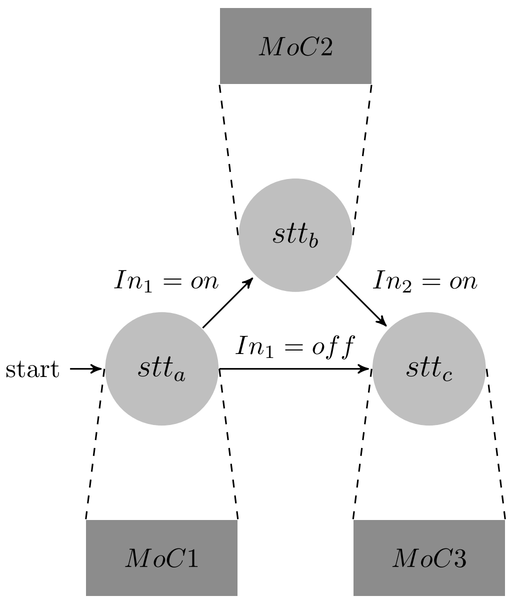


Figure 2.4. Modal model example.

2.3 Frameworks supporting formal MoCs

There is a wide range of formal-based development frameworks that differs on a series of aspects, such as available MoCs, user interface, functionality, underlying programming paradigm and language, and license styles.

Simulink is considered one of the most powerful and complete available framework, being a block diagram environment for simulation and model-based design that is integrated to Matlab (MathWorks, 2019a). Simulink includes tools for modeling, simulating and automatic code generation, among other features. It also checks the model compatibility with industry standards such as DO-178C, DO-331 and ISO 26262. Its main disadvantage, if one can point that, is the commercial license style, besides being a proprietary code framework.

Regarding open source frameworks, *Modelica* is presented as a modeling language alternative targeting CPS, which was formalized in 1997, and is supported by a global and non-profit association (Association, 2019). It also provides a modeling and simulation environment, the *OpenModelica*, also maintained by a non-profit association, the OSMC (Open Source Modelica Consortium (OSMC), 2019).

SDF^3 (Stuijk et al., 2006), read as "SDF For Free", is presented as a tool that aims the generation, analysis and visualization of SDF graphs, as illustrated in Fig. 2.5. Its analysis computes parameters of the SDFG, such as the *repetition vector*, which represents the number of times each actor should be fired to bring the system back to beginning state. In this context, an actor is the representation of an SDF node. The source code of this framework is accessible under the SDF^3 proprietary license conditions. One disadvantage of this framework is the lack of a system simulation tool.

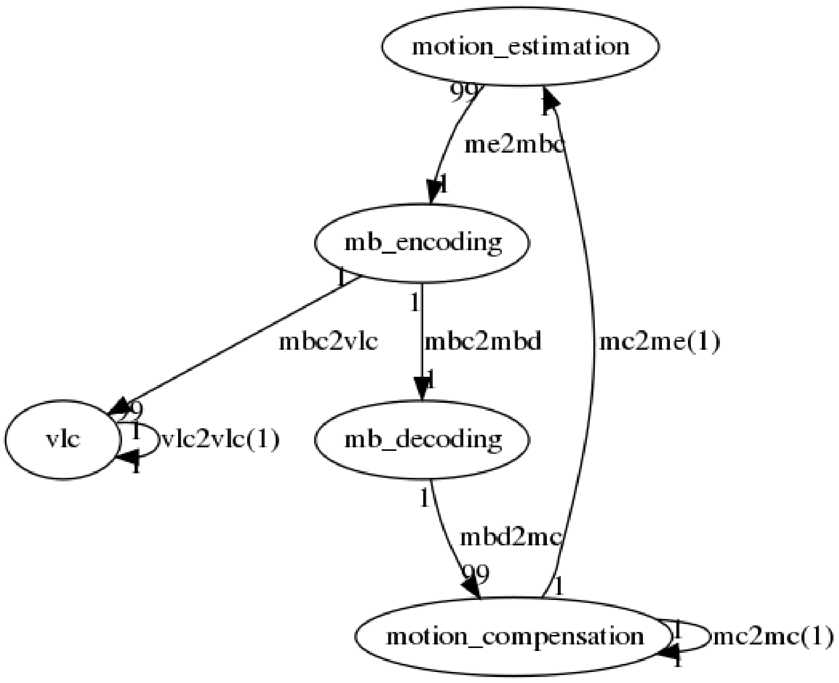


Figure 2.5. H.263 encoder graph generated and visualized with SDF^3 (Stuijk, 2018).

A variety of other frameworks can be found in the literature, *e.g.,* EWD framework (Mathaikutty et al., 2008) and the SystemC modeling framework presented in (Herrera and Villar, 2007).

To reduce the multitude of frameworks and select a couple to work with, the next criteria were proposed and followed in the present work. A framework must support both modeling and simulation of embedded systems, besides the open source code license style (Horita et al., 2019a). Based on this, the framework used in this research case study were ForSyDe and PtolemyII, which are presented in Sections 2.3.1 and 2.3.2.

2.3.1 Formal System Design (ForSyDe)

Aiming the elevation of model abstraction level, and still based on formal design methods, formal systems design (ForSyDe) was first presented in 1999 (Sander and Jantsch, 1999) as a methodology based on a purely functional language, *Haskell*, and on the perfect synchrony hypothesis, thus supporting only synchronous MoC at first. Its main modeling and simulation tool is the ForSyDe-Shallow, implemented as a Haskell embedded domain specific language (EDSL). Nowadays, ForSyDe methodology framework has evolved, including new MoCs, such as continuous time, SDF, and scenario-aware dataflow (SADF) (Bonna et al., 2019), besides other branches and frameworks, *e.g.,* ForSyDe-SystemC, a modeling framework based on the IEEE standard language SystemC (Sander et al., 2016).

The ForSyDe methodology is illustrated in Fig. 2.6. Its synthesis process is divided into two phases: the *refinement* of the high abstraction level specification model into the implementation model, and the *mapping* of the implementation model into a system architecture.

The refinement phase is performed in the functional domain through the application of formal-based design transformations. As a consequence, the model semantics is prevailed and, the formal verification and validation methods applied to these models can be the same.

The mapping phase is allocated in the implementation domain, in which the model processes are mapped to allocation of resources, platform partitioning and code generation.

The main purpose of the refinement phase is to refine the high abstraction specification model, including implementation information, in order to optimize the mapping phase.

The ForSyDe formal based modeling representation and classification is presented in Section 2.4.1.

2.3.2 PtolemyII

The PtolemyII framework is part of the Ptolemy Project, that has been developed at the University of California at Berkeley, starting in the 80's. It aims the formal modeling and simulation of heterogeneous cyber-physical systems and is based on the imperative paradigm and object-

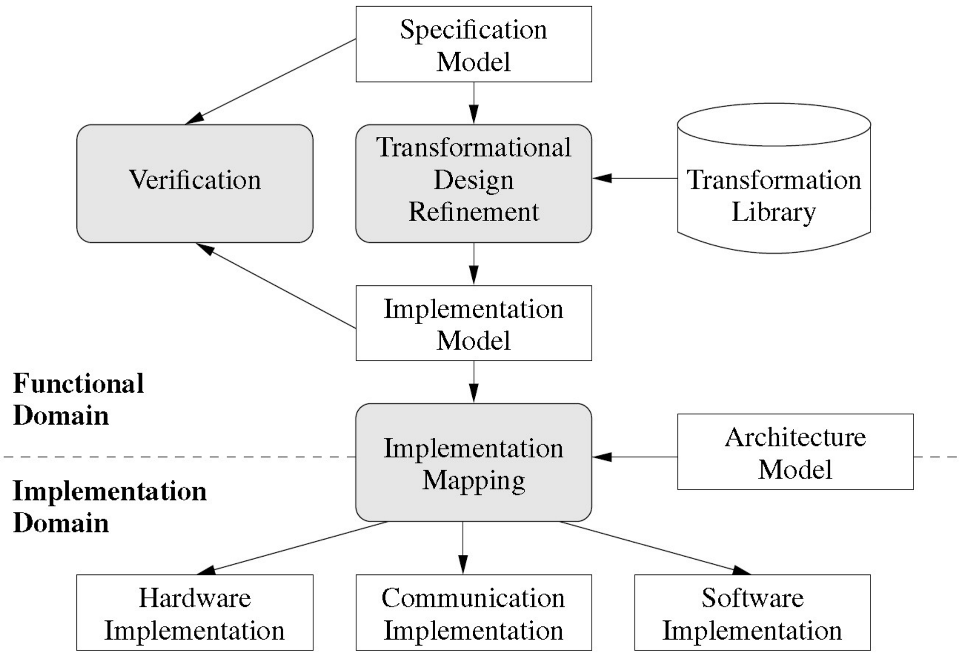


Figure 2.6. ForSyDe Design Process (Sander, 2003).

oriented design, using Java as base language. This provides multi-threading and graphical user interface (Ptolemaeus, 2014).

Since PtolemyII is based on a strongly typed and objected-oriented language, *i.e.,* Java, its architecture has a well-defined package structure and packages functionality, as illustrated in Fig. 2.7.

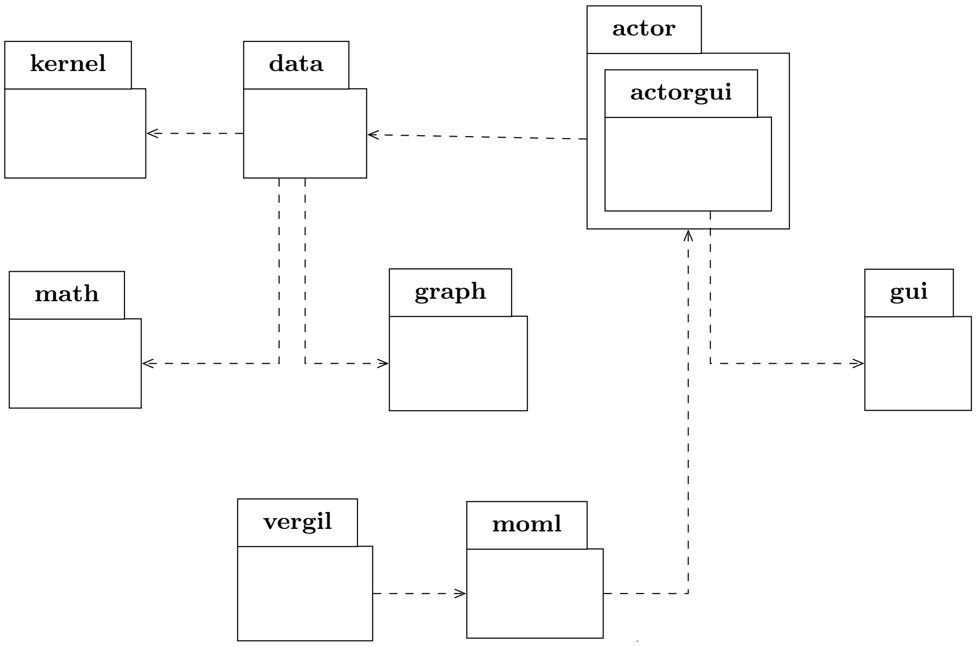


Figure 2.7. PtolemyII architecture based on (Ptolemaeus, 2014).

PtolemyII architecture:

- The `kernel` package defines the structure of MoCs and the relationship among components and domains, besides their hierarchy;
- `Data` package includes classes responsible for data transfer among models. The main class in this package is the `Token`, which represents the base for all units of data exchanged among components;
- `Math` package treats operations with matrices and vectors;
- `Graph` package provides support to plot, analyze and manipulate mathematical graphs;
- The `actor` package implements I/O ports and actors, which are executable entities that exchange data through ports. This package contains the Director class, which are customized to drive the model semantics, *i.e.,* it is the MoC representation inside the application;
- Graphical user interface - `GUI` package offers user interface methods to parameterize and customize the model components. It also provides user interface sub-packages, *i.e.,* vergil package which implements Vergil, the GUI for PtolemyII; and
- `moml` package provides a parser for modeling markup language (MoML) files, which is the XML schema used to store models.

Nowadays, Ptolemy project has grown and includes different branches based on PtolemyII, some sponsored by commercial partners, having as main purpose the application of formal based design methodologies. This set of branches composes the *Industrial Cyber-Physical Systems Center* (iCyPhy) (University of California, 2018).

2.4 MoCs Perspective under PtolemyII and ForSyDe

As the MoCs variety and complexity grows, some concepts and implementation methods can slightly differ from one framework to another, each one having its benefits and drawbacks. This section presents how PtolemyII and ForSyDe classify models of computation and how the MoCs are represented in their system model.

2.4.1 ForSyDe Overview

ForSyDe implements its models of computation based on the TSM. The signals are modeled as *list of events*, where the tags can either be implicitly given by the event position in the list or explicitly specified by a list of tuples, depending on which MoC is used. Two events from different signals with the same tag do not necessarily happen at the same time, their tags only represent the order of events in their specific signal.

In ForSyDe, *infinite signals* can be modeled thanks to Haskell *lazy evaluation* mechanism, as shown in Listing 2.1. That evaluates the necessary number of events by using the `takeS` function applied to the signal $\{5,5,5\}$ (Sander et al., 2016).

Listing 2.1. Haskell Lazy Evaluation.

```
constS  x = x  :-  constS  x
takeS  3  (constS  5)
```

The ForSyDe processes modeling methodology is mainly based on the concept of process constructors, illustrated in Fig. 2.8. This are basically *higher-order functions* that take side effect-free functions and values as arguments to create processes. Each MoC implemented in ForSyDe is essentially a collection of process constructors that enforce the semantics of that specific MoC.

Process constructors are classified as follows.

1. combinational – process that has no state;
2. delay – process delays input; and
3. sequential – process that has an internal state and contains a delay process.

The implementation of heterogeneous systems can be done by using process constructors from different MoC libraries.

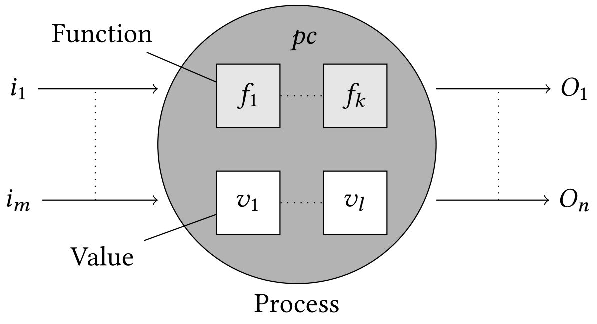


Figure 2.8. Process built by a Process Constructor (Sander et al., 2016).

ForSyDe MoCs classification is derived from the TSM framework, which means they can be divided into timed and untimed MoCs, as shown in Fig. 2.9.

ForSyDe defines the untimed MoCs by sets of process constructors and combinators, characterized by the way its processes communicate and synchronize with each other, and in particular, by the absence of timing information available to and used by processes. It operates on the

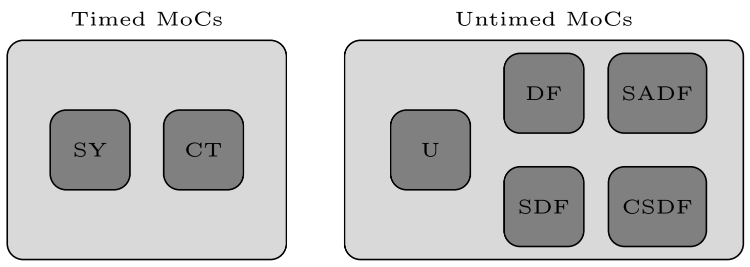


Figure 2.9. ForSyDe models of computation.

causality abstraction of time. Only the order of events and cause and effect of events are relevant (Jantsch, 2003). ForSyDe includes the following MoCs in this category: dataflow (DF); synchronous dataflow (SDF); cycle-static dataflow (CSDF); scenario-aware dataflow (SADF); and the untimed MoC (U) (ForSyDe Group, 2019).

In the timed MoCs, on the other hand, timing information is conveyed on the signals by *absent events* transmitted in regular time intervals, allowing the processes to know when a particular event has occurred and when no event has occurred. The sole sources of processes information are input signals, without the need of access to a global state variable. ForSyDe includes two timed MoCs, the synchronous (SY) and the continuous time (CT) (ForSyDe Group, 2019).

2.4.2 PtolemyII Overview

The main way to model and simulate systems using PtolemyII is using its GUI, called *Vergil*. The models are represented as a set of actors, that communicates through interconnected ports. The model semantics are driven by an special actor called Director, which is the graphical representation of the selected MoC.

The actors represent the system processes and can be classified into two groups: opaque and transparent. The opaque actor has its intern logic invisible to the outside model, *i.e.,* the sub-model inside it can be driven by a different Director, composing a hierarchical and heterogeneous system model. The transparent actor semantics are driven by the Director included in the model they belong to. All the actors have its source code available and are customizable.

PtolemyII includes a wider set of MoCs when compared to ForSyDe. Fig. 2.10 illustrates the most commonly used and their relationship, considering their behavior and heritage, according to (University of California, Berkeley-online, 2018).

In PtolemyII, synchronous-reactive (SR) MoCs execution follows ticks of a global clock. At each tick, each variable, represented visually in Vergil by the wires that connect the actors, may or may not have a value. Its value, or absence of value, is given by an actor output port connected to the wire. The actor maps the values at its input ports to the values at its output

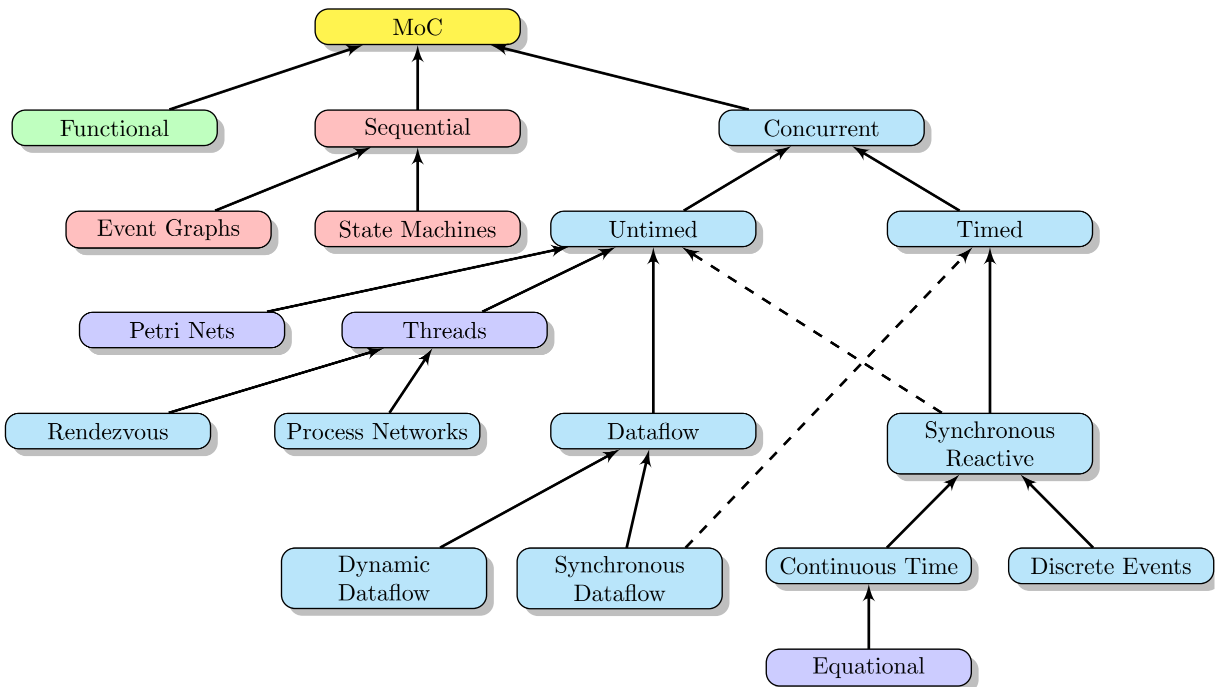


Figure 2.10. MoCs Chart, according to (University of California, Berkeley-online, 2018).

ports, using a given function. The function can vary from tick to tick.

The PtolemyII SR MoC is by default untimed, but it can optionally be configured as timed, in case there is a fixed time interval between ticks.

In PtolemyII dataflow MoCs, the execution of an actor consists of a sequence of firings, where each firing occurs as a reaction to the availability of input data. A firing is a computation that consumes the input data and produces output data.

Considering the SDF domain, when an actor is executed, it consumes a fixed amount of data from each input port, and produces a fixed amount of data to each output port. As a consequence, the potential for deadlock and boundedness can be statically checked, and its schedules can be statically computed. PtolemyII SDF MoC can be timed or untimed, though it is usually untimed.

An example of non-concurrent MoC included in PtolemyII is the FSM, which is considered as a sequential MoC. In that case, the graphical components are not actors, but states, and their relationships represent transitions between those states. Transitions have guards that determine when state transitions can occur.

An FSM can be used to define the behavior of an actor used in other domains. When that actor executes, the FSM reads the inputs, evaluates the guards to determine which transition to take, and produces outputs as specified on the selected transition.

FSM is also used to create a class of hierarchical models, the modal models. In this case, the states of an FSM contain submodels that process inputs and produce outputs. Each state of the FSM represents a mode of execution, that can be driven by different MoCs. When a submodel is not active, its local time does not advance.

2.5 Functional Programming Paradigm

Considering that systems should be first modeled at a high abstraction level during the specification phase of a project (Edwards et al., 1997), the present work focus is on functional programming paradigm-based framework, *i.e.,* ForSyDe.

The reason is that functional paradigm features elevate the model abstraction level. It is possible to list higher-order and side-effect free functions, data abstraction, lazy evaluation and pattern-matching as useful features.

Functional programming base was first introduced in 1936, when Alonzo Church *et al.* presented the main concepts of *Lambda Calculus* (Turner, 2013). In that paradigm, all the computation functions of a program are mathematical expressions to be evaluated.

Some languages were then created based on that paradigm, including Lisp, Miranda, Scheme and *Haskell.* ForSyDe has as its main programming language Haskell.

2.5.1 Haskell Programming Language

Haskell had its first official report issued in 1990, having an update in 1998, named *Haskell 98 Report* (Hudak et al., 2007). During the first years of existence, Haskell was mainly used in the academic area. During the 2000's, with the advents of multi-core processors and logical parallelism needs, it gained visibility and developers began to contribute with several libraries, having a full language revision in 2010.

Haskell is a pure functional language, which means it is exclusively based on the concepts of this paradigm. Its main characteristics can be briefly described as follows.

- *Laziness*: all the function and variables evaluation are done only when needed. As a consequence, it is possible to declare infinite lists in Haskell, taking the necessary values only when they require to be used;

- *Statically Typed*: all variables and functions types are known at compile time. Besides, types can also be inferred by the compiler. If it cannot infer or identify a variable or function type, an error is raised at compile time;

- *Referential transparency*: functions always return the same value considering they have the same inputs. As a consequence, all functions can be replaced by their returning value;

- *Immutable state*: functions do not implicitly modify variables or states, and cannot affect other functions, *i.e.,* they are side-effects free;

- *High level*: complex algorithms can be implemented through simpler syntax when compared to other paradigms, *e.g.,* imperative or object-oriented.

As Haskell becomes more popular, more tools and packages are developed. Its main compiler is the *Glasgow Haskell Compiler* (GHC), commonly used with its interactive environment, the *GHCi*, which supports different operational systems and platforms. Besides, the environment has an useful system used for building and packages libraries called *cabal*. It builds the applications in a portable way. The main tools can be found in the Haskell home page (Haskell, 2018).

2.6 Imperative Programming Paradigm

Although the imperative programming paradigm has not the same abstraction level as the functional one, it has characteristics that can contribute in model-based design, *e.g.,* concurrency programming or classes inheritance. The imperative paradigm representation is directly derived from the way digital hardware works, by signals changing state over time. The machine code works in an imperative way, without abstractions. This paradigm can be explained as a sequential execution of commands over time, producing a program that behaves as a state function of time (Scott, 2009). Some widely used pure imperative languages are C, Pascal, Basic and Fortran.

Imperative programming paradigm can be divided into some subsets, depending on the research, this classification can differ. Scott (2009) includes into these subsets the *von Neumann*, *scripting* and *object-oriented* paradigms.

2.6.1 Java Language

Java, the PtolemyII base language, is classified as an object-oriented programming language. The main feature of this paradigm is the modularity and encapsulation, improving the scalability when compared to other languages, like C.

The Java attributes allows the modeling of concurrent or parallel process of a system, using threads. This facilitates the creation of new classes, due to its inheritance feature, and allows the development of graphical user interfaces.

2.7 Straightforward Paradigms Comparison

Although imperative paradigm has its advantages, allowing a low level software tuning, the functional paradigm enforces safer programming and higher abstraction level, attending one of the model-based development methodology targets. Moraes and Loubach (2017) briefly lists benefits and drawbacks of these paradigms when programming real-time systems, as shown in Table 2.1.

Table 2.1. Functional and Imperative Paradigms Comparison (Moraes and Loubach, 2017).

	Imperative	**Functional**
Memory Management	Allocation and deallocation are done as the developer implements.	Functional languages are interpreted/-compiled and high-level, the compiler itself implements garbage collection.
Determinism	To ensure determinism, all state mutations must be covered, which is complex in imperative programs.	Functional programs have referential transparency and force explicit notation on code that mutates state, making validation on functional determinism much easier.
Time Measure	Due to their proximity with machine language, imperative languages are straightforward regarding time measuring.	Due to garbage collection's non determinism, it can be hard to measure time for specific functions.
Concurrency	Concurrency adds yet another level of complexity regarding state mutations, since the number of possible interactions between threads is exponential on the number of processors.	Immutable state allows for concurrent independent computation without affecting the functional determinism.

2.8 Properties Verification

Working with formal-based models and targeting the correct-by-construction design requires the model to be simulated and tested in different procedure steps. One of these steps consists on the verification whether the model holds all the required system properties it should, or not.

The exhaustive test of each property, comprehending all the possible errors, is a hard and labor intensive task, costing a considerable portion of the software development. In this sense, a range of tools have been presented towards the automatic generation of test cases for model properties verification (Claessen and Hughes, 2000; Naylor and Runciman, 2007; Runciman et al., 2008; Castagna and Gordon, 2017).

Quickcheck (Claessen and Hughes, 2000) was first presented in 2001 as a DSL, implemented in Haskell, targeting the test of program properties and was used as base for other similar tools having the same purpose, *e.g.,* SmallCheck (Runciman et al., 2008), Reach (Fowler and Huttom, 2016), and Luck (Castagna and Gordon, 2017). Besides, it has also been emulated in many other programming languages, including Scala, F# and Google Go (Horváth et al., 2010).

To accomplish the properties verification, Quickcheck uses pre-defined checkable formal system specifications, called *properties*, as inputs and automatically generates random values test cases for each one, indicating if the model holds the desired property or not.

2.9 Automatic Code Generation

With the elevation of the model abstraction level, aiming the verification and simulation of the system functionality, there is a lack of implementation details. Towards a more robust, scalable and formal-based design, researches have been presenting alternatives for automatic code generation based on formal models of computation.

A synchronous-based code generator for explicit hybrid systems languages, which includes discrete and continuous time semantics, was presented by Bourke et al. (2015). A compiler was created to generate statically scheduled sequential code having as inputs a formal model based on Scade 6 synchronous languages using ordinary differential equations. Fig. 2.11 illustrates the presented compiler flowchart. It passes through some consistency analysis and optimization. The code is then translated into a sequential object language (SOL), having its output sliced into functions and again optimized by removing dead-code from the source. Finally, the SOL is translated into C language.

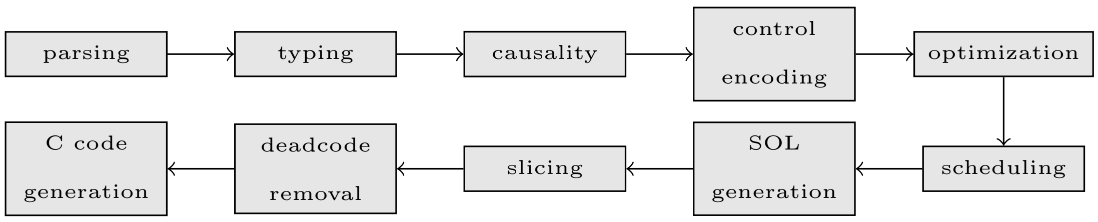


Figure 2.11. Compiler architecture based on (Bourke et al., 2015).

Another automatic code generation tool was presented by Grabmüeller and Kleeblatt (2007), called Harpy. It is a domain-specific language for runtime code generation targeting x86 architectures. Its main base language is Haskell, taking advantage of the its features for needed abstractions and language extensions, *e.g.,* meta-programming. Harpy was added to the Haskell community's central package archive, which is called Hackage (Community, 2019).

2.10 Main Related Works

In the context of higher abstraction level models and abstraction gap, frameworks and methodologies have been developed towards possible automation approaches for formal model-based design flows. Those tools differ in aspects such as input modeling language, output implementation language and target design step.

Simulink is one of the most powerful and commercially used tools for model-based design of systems. One of its design flow automation tool is the Embedded Coder, which generates C and C++ code for embedded processors in mass production. The generated code is intended to be portable and can be configured to attend standards such as MISRA C, DO-178, IEC 61508 and ISO 26262 (MathWorks, 2019b). Studies have used that tool to test and implement code into customized applications.

The work presented in (Krizan et al., 2014) discusses Simulink usage for automatic generation of C code in critical applications, according to DO-178C and DO-331 standards. They also discussed the possibility of automatic code generation of a whole application or of separate parts, or tasks.

The main difficulties when working with Simulink tools are related to its license costs and proprietary implementation source code, commonly making it impracticable for researchers and developers to perform a deeper analysis of its capabilities and algorithms.

The present research advocates to open source frameworks and tools, making it possible to take advantage of researches collaborative environment and deeper code analysis.

Related to open source software, Copilot is presented as a DSL, based on Haskell, targeting runtime verification (RV) programs for real-time, distributed and reactive embedded systems (Pike et al., 2013). Runtime verification programs are applications which runs in parallel with the systems application, monitoring its correctness ans consistency at the runtime. Copilot is implemented throughout a range of packages, including automatic generations of C code: *copilot-c99*, based on Atom Haskell package, and *copilot-sbv*, based on SBV Haskell package. Although Copilot is a powerful tool for safe C code generation, even for critical systems, it mainly targets a specific application type, the RVs. The present work aims to address a wider range of application and cyber-physical systems.

Requirements to Design Code (R2D2C) project was first presented by NASA aiming full formal development, from requirements capture to automatic generation of provable correct code (Rash et al., 2006). Its approach takes the specifications defined as scenarios using DSLs, or UML cases, infers a corresponding process-based specification expressed in communicating sequential processes (CSP), and finally transforms this design to Java programming language. That tool also makes it possible to apply reverse engineering, extracting models from programming language codes. The present research aims to analyze possible automation strategies for embedded systems design flows having as inputs formal MoCs implemented through a framework or an EDSL

Some challenges, advances and opportunities of embedded systems design automation were presented in (Seshia et al., 2017). According to that research, some CPS characteristics were listed as obstacles for this automation, including: heterogeneity, dynamic and distributed systems, large-scale and existence of human-in-the-loop. They argued that, for design automation tools, a series of features would be necessary, *e.g.,*. cross-domain, learning-based, time-awareness, trust-aware and human-centric. One of the possible presented approaches was the combination of model-based design (MBD), contributing with formal mathematical models, and data-driven learning, which inputs data resulted from extensive field testing.

The present work represents an effort on contributions to the embedded design automation based on MBD research. A case study is designed, modeled, simulated, verified and implemented in high level, targeting the identification of automation approaches to help overcoming the listed difficulties.

2.11 Summary

This chapter presented the main concepts used along with the research, starting with embedded and cyber-physical systems, followed by models of computation and the programming paradigms. In addition, it presented examples of frameworks that support formal CPS modeling. Automatic code generation tools and methodologies were also considered in this chapter.

Finally, the main related works targeting formal embedded systems design automation were presented, highlighting the main differences to this research.

Next chapter describes the methodology and analysis introduced in the present research, representing its contribution proposal.

3 A METHOD FOR POSSIBLE AUTOMATION EXPOSURE

This chapter presents the proposed method for *analysis and identification of possible automation approaches* (AIPAA) applicable to embedded systems design flow supported by formal models of computation.

Towards the automation approaches identification, the introduced method uses a design methodology, separated into well-defined steps, as described in Section 3.1. Next, the design steps are analyzed, resulting in a set of possible automation approaches, presented in Section 3.2. An illustrative example representing the methodology application is presented in Section 3.3.

3.1 Analysis and Identification of Possible Automation Approaches - The AIPAA Method

One of the first steps when designing an embedded system is the system description in the form of *problem statement*. This includes functionalities, behaviors, capabilities and constraints of the system. An accurate and detailed problem statement leads to an effective, robust and optimized design flow.

Elaborating the embedded system detailed problem statement is not a trivial task. The present research work considers a design flow methodology assuming the problem statement is already described. The problem statement is considered to be an *input* for the system design flow under use.

The AIPAA method takes into account only the *system specification* domain. The implementation domain comprehends a variety of concepts and tools other than this research scope. However, the present work presents straightforward guidelines on how the implementation domain can take advantage of the AIPAA outputs.

Fig. 3.1 illustrates the proposed method, *i.e.,* AIPAA. Each one of the five steps are detailed in the next subsections.

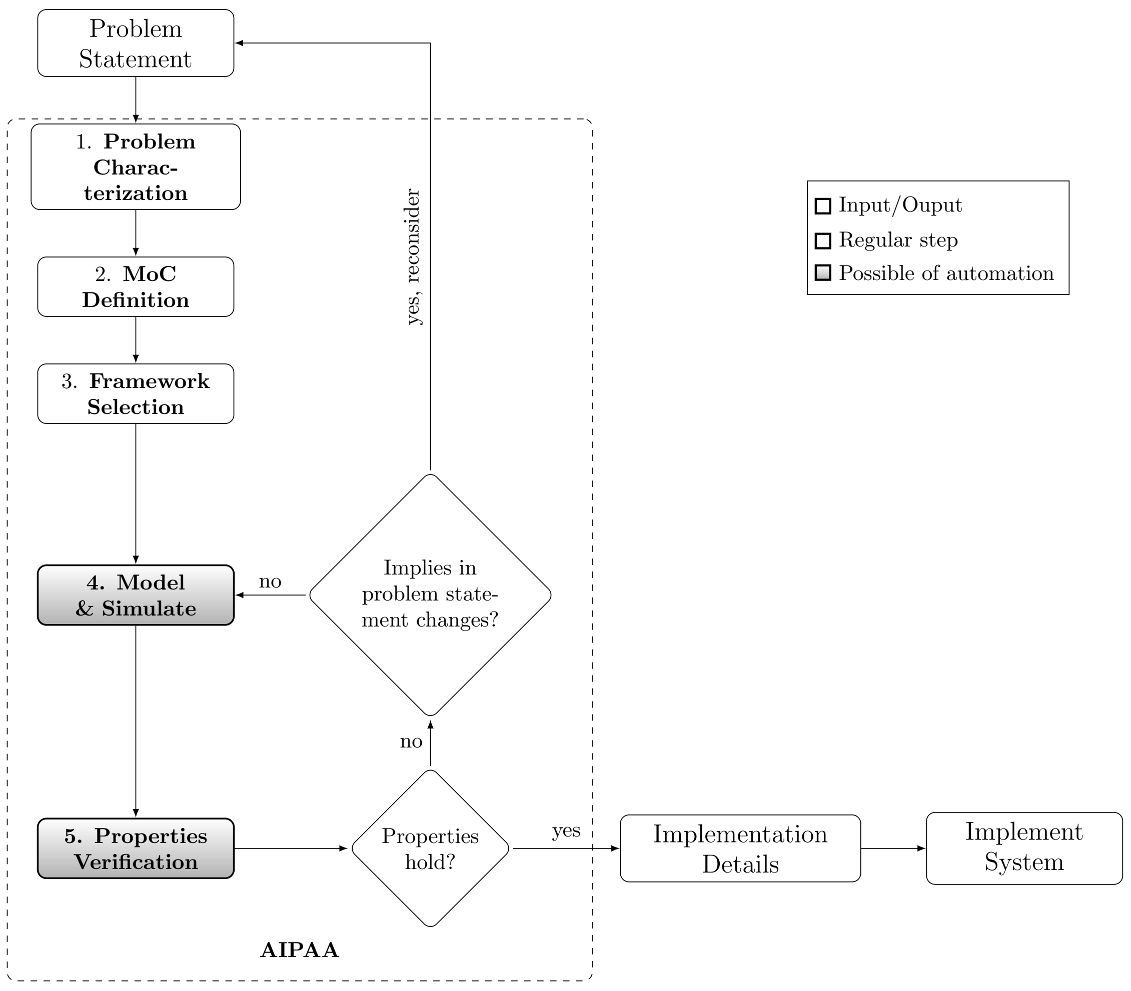


Figure 3.1. Analysis and identification of possible automation approaches (AIPAA) applicable to embedded systems design flow. AIPAA needs the "problem statement" as a given input, and aids to produce a model, which is verified and executable, as output. This output can be used as entry point in the implementation domain, *e.g.,* "implementation details" and "implement system".

3.1.1 Problem Characterization

This first step, *problem characterization*, aims to conduct an initial analysis on the "problem statement" input aiming the identification of its relevant behaviors and characteristics towards the design of the system.

To facilitate the identification of problem semantics, both the problem procedures and processing must be separated into *functions*, also defining how they should execute and communicate to each other.

These definitions allow for the problem analysis and characterization, which will aid in the further MoC definition. When applying the method, the following questions must be answered.

Q_1 - Does the problem have concurrent or only sequential processes?

Q_2 - Are the problem communication paths totally or partially ordered?

Q_3 - Is the problem a dataflow problem?

Q_4 - Are the inputs and processing cyclic?

Q_5 - Can a single clock be used to synchronize all the involved processes?

Step Required Inputs

This step requires the following inputs:

1. Problem statement describing the problem functionalities, behaviors, inputs and desired outputs.

Step Expected Output

After the application of this step considering the required inputs, the expected outputs are:

1. Problem separation into functions and their relationship definition; and

2. A list of well-defined problem characteristics according to the previously stated questions.

3.1.2 MoC Definition

As stated in Section 2.2, models of computation describe only the characteristics that are relevant for a given task. In this sense, it is needed to *analyze the problem characterization step results* aiming the identification of relevant behaviors and functionality that permits its semantics classification into one or more MoCs.

A range of complex applications comprehends behaviors that may not be possible to be modeled using just a single MoC. As a consequence, the model semantics must be split into more than one MoC, generating the concept of heterogeneous systems modeling. In that case, a second phase should be included into this step, defined as MoC division, which classifies each problem characteristic into a MoC that better fits it.

The present research work only addresses the study of a single MoC at a time within a system, *i.e.,* using either SY or SDF MoCs. For this limitation, an analysis on the available MoCs was performed and just two MoCs were selected, based on the criteria that they had to be representative, widely used and should differ on timing abstractions.

As stated in Section 2.2.2, *SY is a timed MoC*, *i.e.,* its signals are totally ordered, and it is based on the *perfect synchrony hypothesis*. The *SDF*, on the other hand, is an *untimed MoC*, *i.e.,* its signals are partially ordered, composed by nodes with fixed token rates and it is non-terminating, as described in Section 2.2.3.

Step Required Inputs

This step requires the following input:

1. List of problem characteristics.

Step Expected Output

After the application of this step, the expected outputs is:

1. The definition of a MoC to be used in the system design flow.

3.1.3 Framework Selection

A support framework is an important tool that aids in the system *modeling* and *simulation* step of the design flow. There is a number of facts to be considered when selecting a formal-based framework. The proposed AIPAA method considers the following.

- **License style**: The framework license style must comply with the project requirements. It can vary from open source or freeware to a proprietary and paid license style;
- **Included MoCs**: The set of available MoCs can drastically differ when comparing frameworks. *e.g.,* SDF^3 only supports SDF, SADF and CSDF, on the other hand, Simulink, ForSyDe or PtolemyII supports a range of MoCs, as illustrated in Fig. 2.10. The selected MoC, in Section 3.1.2, must be part in the framework supported MoC list;
- **Capabilities**: The frameworks varies on its capabilities, including system simulation and heterogeneous system modeling. In this sense, model simulation figure as an interesting feature for a framework to have;
- **Interfaces**: Frameworks differ on its modeling and simulation interfaces. It ranges from a script environment to a GUI. The simulation input and output interfaces varies from files, predefined or user inputted and must also be considered;
- **Scalability**: Towards scalable models, some frameworks are implemented based on programming languages and have interfaces that facilitates the modeling of large or distributed systems;

- **Programming language paradigm**: Frameworks are based on a wide range of programming languages, leading to a variation on its programming paradigm, *e.g.,* imperative or functional. Each paradigm has its own benefits. Functional can lead to a better model scalability and higher abstraction level, on the other hand, imperative paradigm facilitates new MoCs modeling due to inheritance capability.

A research on available frameworks must be performed, submitting them to a posterior analysis considering the previously listed criteria, as a *minimum* one. Each framework has its own benefits and drawbacks, and as a consequence the framework that best suits the system modeling and simulation varies from one application to another. (Horita et al., 2019a) conducted a research with respect to the framework selection considering a list of predefined parameters and criteria. That work parameters are used in the present research.

Step Required Inputs

This step requires the following inputs:

1. Problem characterization; and
2. Selected MoC.

Step Expected Output

After the application of this step, the expected output is:

1. The definition of the framework to be used for system modeling and simulation.

3.1.4 Modeling and Simulation

The *modeling and simulation* step next to the *properties verification* are fundamental steps towards the correct-by-construction design. After the application of these steps, the model correctness can be verified and the system is ready for advancing to the implementation domain.

Based on the problem characterization, the first part of this step is to *model the target system*, using the previously selected MoC with the aid of the defined support framework.

Aiming the model consistency verification, some frameworks detects model errors during compile time, such as inconsistent types in a communication path and processes missing information, *e.g.,* types or number of ports. Besides, a synthetic input data set must be *simulated* as the last part of this step, aiming the detection of runtime inconsistencies, such as model infinite loops or model acceptance of unwanted inputs types, leading to unexpected behaviors.

In case any model inconsistency is detect, it is necessary to modify and fix the model, recompile and re-simulate it, within an iterative way. After testing boundary conditions and different input types and sizes, resulting in expected outputs and system behavior, the model can be submitted to the next step in the AIPAA context.

Step Required Inputs

This step requires the following inputs:

1. Problem characterization;
2. Selected MoC; and
3. Selected framework.

Step Expected Output

After the application of this step, the expected output is:

1. System formally modeled, compiled and simulated aiming its further properties verification.

3.1.5 Properties Verification

After the model consistency is verified, it is necessary to *check whether the system model properties*, with respected to the chosen MoC, hold. This is performed by checking a list of minimum defined system properties.

In this sense, the first part of this step is to *identify the minimum properties set* that the system must hold. Each MoC comprehends a set of properties that needs to be verified. Besides the in-use MoC specific properties, it is also possible to verify system particular properties in this step. Focusing on the MoCs included in this work scope, the following properties must hold.

- For the SY MoC:

  P_{SY1} signals synchronized and totally ordered;

  P_{SY2} well-defined absent value for an event; and

  P_{SY3} absence of zero-delay feedback, *i.e.,* for each loop-back, a delay must be implemented, avoiding algebraic loop. In SY MoC, there are some options to handle loop-backs (Sander, 2003). The AIPAA method adopts the strategy of forbidden zero-delay loop-backs.

- For the SDF MoC:

P_{SDF1} buffer size determination based on a feasible schedule, *i.e.,* the buffers must be in accordance with token rates, so that there is nor data loss neither overflows;

P_{SDF2} system is non-terminating, *i.e.,* it cannot have deadlocks;

P_{SDF3} schedulability, *i.e.,* it must be possible to extract a valid single-core schedule as a finite sequence of actor firings; and

P_{SDF4} fixed model token rates. As SDF blocks have fixed token rates, the model also will not vary its rates.

The last part of this step consists of *verifying whether the system holds the minimum listed properties*. Some properties are verified by explicit analysis, they are simply MoC intrinsic. For the others, there are different methods for property verification, including mathematical proof, model analysis or test cases application, which is selected based on the property to be verified.

- For the SY MoC:

V_{SY1} *Signals are synchronized and totally ordered.*
This property is MoC intrinsic for SY. As long as the system is modeled following the SY MoC abstraction, this property holds. It is important that the input signals is correctly ordered considering this property, not having unexpected model behaviors. This verification is carried out by explicit model analysis;

V_{SY2} *Well-defined absent value for an event.*
This property is also MoC intrinsic for SY. The framework must handle the absent value for an event towards the maintenance of the perfect synchrony hypothesis; This verification is through explicit model analysis;

V_{SY3} *For each loop-back, a delay must be implemented.*
The verification of this property must be through the model implementation code or interface analysis. First, it is necessary to check whether the model has any loop-back, and in positive case, check the existence of the at least one cycle delay in it. This verification is possible to be automated by using a *regular expression checker*.

- For the SDF MoC:

V_{SDF1} *Buffer size determination based on feasible schedule.*
The verification of this property can be mathematically demonstrated, as presented by Lee and Messerschmitt (1987):

(a) Find the model topology matrix Γ; and

(b) Check if $rank(\Gamma) = s - 1$.

V_{SDF2} *System is non-terminating.*

The absence of deadlocks is mathematically proven, as demonstrated by Lee and Messerschmitt (1987). To verify this property it is necessary to prove the existence of a PASS, which can be done by performing the following:

(a) Check whether the model holds P_{SDF_1}. If not, the system has no PASS;

(b) Find a positive integer vector q belonging to null space Γ;

(c) Create a list L containing all model blocks;

(d) For each block $\alpha \in L$, schedule a feasible one, just once;

(e) If each node α has been scheduled q_α times, go to next feasible α; and

(f) If no block in L can be scheduled, the system has a deadlock.

V_{SDF3} *Schedulability analysis*;

For a model to be schedulable, property P_{SDF_1} must hold. Moreover, the model graph must be connected (Lee and Messerschmitt, 1987). This must be verified through model analysis.

V_{SDF4} *Fixed model token rate.*

The fixed token rate property is verified by submitting the model to test cases. Different input values and signal sizes must be used to verify the model robustness.

This is the last step of the system specification domain. This last AIPAA method step produces a possible entry point for the system implementation domain. For this reason, the system properties verification must be performed until the system correctness is proven. The output of this step must be a verified correct simulateable system, which must be also in accordance to the input problem statements.

Step Required Inputs

This step requires the following inputs:

1. MoC minimum properties list; and
2. Simulateable system model.

Step Expected Output

After the application of this step, the expected output is:

1. A formal-based system executable model with its stated correctness and properties verified.

3.1.6 Directions to Implementation Details and System Implementation

This section provides some possible directions to the implementation details, as the AIPAA method does not contemplate this domain. System implementation includes complex and extensive procedures and methods that can be considered in future works.

Here, an analysis of systems model and functionalities implementation possibilities is performed, defining a list of implementation details.

1. Division of the functionalities implemented in hardware and the ones in software, *i.e.,* hardware and software co-design;
2. Communication interfaces, methods and protocols;
3. Hardware architecture comprehending:
   (a) Necessary resources *e.g.,* memories, processors, power; and
   (b) Necessary units, such as arithmetic logic unit (ALU), floating point unit (FPU), graphics processing unit (GPU), or reconfigurable hardware (FPGA).
4. Software architecture comprehending:
   (a) Implementation programming language; and
   (b) Programming frameworks and tool-chains to be used.

Implementation models can be developed to assist in this analysis (Loubach, 2016). This step represents the base of the system implementation. Related to correct-by-construction methodologies, the definition of implementation details can only proceed when modeling, simulation, and properties checking are already verified.

Step Required Inputs

This step requires the following inputs:

1. Problem characterization;
2. Verified system model; and
3. Hardware, software and communication requirements.

Step Expected Output

After the application of this step, the expected outputs are:

1. System implementation architecture description;
2. Hardware platforms to be used;
3. Low-level programming language to be used; and
4. Software tool-chains to be used.

Based on the listed implementation details, and on the system model from Section 3.1.4, the system can be implemented using hardware description languages, *e.g.,* VHDL or Verilog, and software implementation languages, *e.g.,* C, C++ or Assembly.

3.2 Analysis of Possible Automation Approaches

This section introduces two main possibilities identified through the AIPAA method elaboration. The first one is about the *properties verification*, and the second about *automatic code generation*.

3.2.1 Proposed Automation to Properties Verification

As stated in Section 2.8, the properties verification through test cases application is a labor intensive task. For that reason, AIPAA considers the use of a partial automated verification tool towards a faster, more scalable, robust and trustable process.

In this sense *Quickcheck* can be employed. It is a widely used tool to aid in the process of automatic verification of the system properties generating test cases to evaluate whether the system behaves as expected.

Considering the MoC properties verification methods listed in Section 3.1.5, the Listing 3.1 presents the use of Quickcheck to verify the previously defined property P_{SDF_4}, which is based on the application of test cases. This example assumes the verification of a model containing 2 input signals and 1 output signal. The verification is based on two requirements:

1. The output signal length divided by its corresponding token rate must be equal to the number of model *firing* cycles, which is calculated by selecting the smaller division result among the input signal length and its token rates; and

2. The output signal length should be multiple of its corresponding token rate.

The Haskell verification code (Listing 3.1) comprehends the following variables definitions:

- `propFixTr21`: Fixed token rate property to be verified;
- `c1` and `c2`: Input signals token rates;
- `p`: Output signal token rate;
- `a1` and `a2`: Quickcheck automatic generated model test cases; and
- `actor`: The model to be tested.

Listing 3.1. Quickcheck usage example.

```haskell
-- Property to be verified: Model fixed token rates.

module PropsSDF ( propFixTr21
  ) where

import ForSyDe.Shallow
import Test.QuickCheck

-- Fixed token rate property for actor with 2 inputs and 1 output
propFixTr21 :: (Signal a -> Signal b -> Signal c) -> (Int, Int) -> Int
            -> [a] -> [b] -> Bool
propFixTr21 actor (c1, c2) p a1 a2 =
  minimum [div (lengthS in1) c1, div (lengthS in2) c2] == div (lengthS out) p
  && lengthS out ‘mod‘ p == 0
  where in1 = signal a1
        in2 = signal a2
        out = actor in1 in2
```

To illustrate the presented Quickcheck verification usage, Listing 3.2 presents a minimal model example based the SDF MoC, consisting of a system which interpolates the events of two input signals into one output signal.

Listing 3.2. SDF based system model example.

```haskell
-- Example: An SDF based model which interpolates two input signal events into
-- an output signal.

module InterpolateSDF ( interpolate
  ) where

import ForSyDe.Shallow
import Test.QuickCheck
import PropsSDF

-- System Model
interpolate :: Signal a -> Signal a -> Signal a
interpolate = actor21SDF (1,1) 2 (\[x1] [x2] -> [x1,x2])

-- Property Verification
main = quickCheck (propFixedTr21 interpolate (1,1) 2 :: [Int] -> [Int] -> Bool)

-- If model holds the property, True value will be returned
```

3.2.2 Automatic Code Generation

The AIPAA output consists on a verified high level abstraction model, in the specification domain. One of the strategies adopted to overcome the abstraction gap in a more robust and scalable manner is a semantic preservation mapping and transformation leading to an automatic code generation using as input the verified formal system model, as the one resulted in the AIPAA.

Automatic code generation is an extensive and complex subject that have been studied and implemented by researches such as (Bourke et al., 2015; Grabmüeller and Kleeblatt, 2007; MathWorks, 2019b). In this context, ForSyDe framework includes a deep-embedded DSL, named ForSyDe-Deep (Acosta, 2008). That tool is able to compile and analyze the model, automatically transforming it to be embedded into the hardware target platform.

3.3 Illustrative Example

This section presents an *illustrative minimum work example* of the AIPAA method application showing the inputs and outputs of each step. The example considered is an encoder-decoder system (EDS), based on the one presented in (Horita et al., 2019a; Loubach et al., 2016).

The EDS takes as input a sequence of values to be encrypted and then decrypted, and a sequence of encryption keys, which is used to generate encryption and decryption functions to be in turn applied to the input values. The EDS outputs two sequences, one containing the decrypted values, *i.e.,* this output sequence must be the same as the input sequence, and the other contains the encrypted values.

3.3.1 Problem Characterization

This step performs an initial analysis on the problem statement, identifying its functionalities and characteristics.

The EDS can be divided into 4 functions: the generation of encryption and decryption functions and the encryption and decryption of the input values. The developed flowchart of the system is presented in Fig. 3.2.

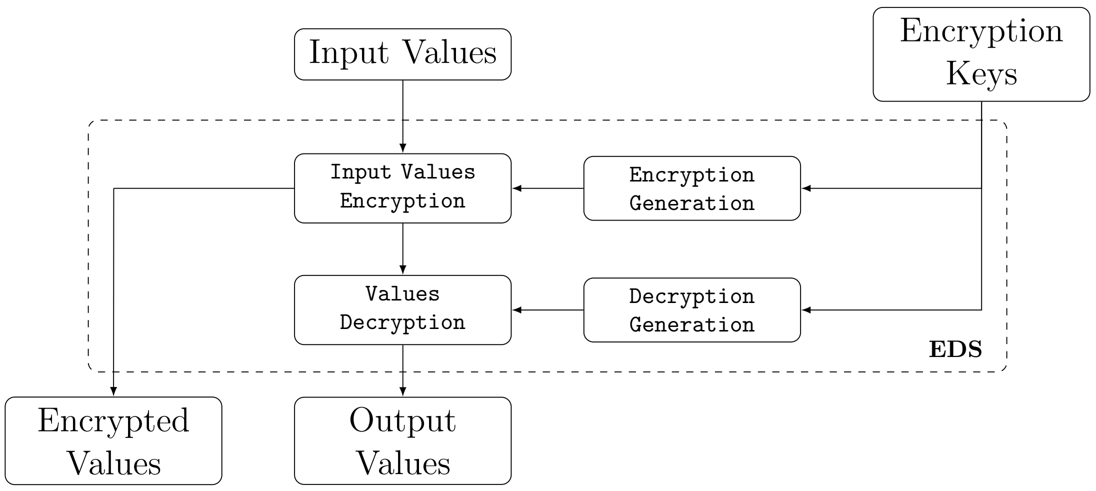


Figure 3.2. EDS system flow chart.

Considering these defined functions and their relationship, it is possible to define the characteristics listed in Section 3.1.1 for the EDS:

Q_1 - The problem has concurrent processes, *i.e.,* encryption and decryption function generation can be executed simultaneously;

Q_2 - Its communication paths are totally ordered, *i.e.,* the signal can only be encrypted or decrypted after their base functions are generated. Therefore the problem is timed;

Q_3 - The input sequence is encrypted and decrypted and the process is finished, as a consequence, this is not a dataflow problem, which is non-terminated;

Q_4 - The inputs and processing are not cyclic; and

Q_5 - A clock can be used to synchronize all involved processes, considering the absent value is well-defined in the used framework.

3.3.2 MoC Definition

This step aims the definition of one or more MoCs that best suits the target system. Based on the proposed characterization, the EDS is classified as timed, containing concurrent processes and with synchronized functions. In this sense, the system can be modeled based on the perfect synchrony hypothesis, using the SY MoC.

3.3.3 Framework Selection

This step takes into consideration the points listed in Section 3.1.3 to select the modeling and simulation framework:

- License style: the present research work focus is on open source tools;
- Included MoCs: the framework must include the selected MoC, *i.e.,* SY;
- Capabilities: the proposed design includes both modeling and simulation;
- Interfaces: this illustrative example accepts inputs and output from an user interface or from files;
- Scalability: this illustrative example targets model high scalability. In this sense, a text-based coding interface is better than a GUI;
- Programming language paradigm: towards a higher level of abstraction, Haskell, a pure functional programming language, was selected as the framework base language.

Based on the listed points, ForSyDe was selected as the modeling and simulation framework for this illustrative example, since it conforms with all the elicited requirements for the framework tool selection.

3.3.4 Modeling and Simulation

This step comprehends the system formal-based modeling and its simulation.

The EDS SY model graph is illustrated in Fig. 3.3.

Based on the Section 3.3.1 characterization, the system has 2 inputs and 2 output, which is modeled by using ForSyDe signals.

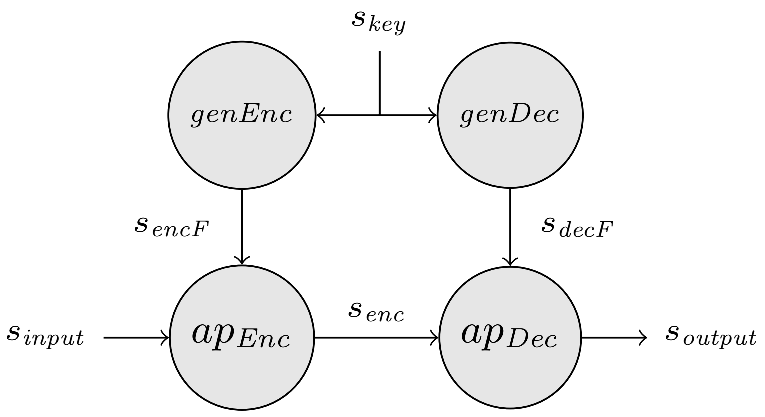


Figure 3.3. Encoder-decoder system SY graph.

- s_{input}: input signal that is encrypted and then decrypted;
- s_{key}: input signal used to generate the encryption and decryption functions;
- s_{enc}: output signal composed by the encrypted values; and
- s_{output}: output signal composed by the values that are encrypted and decrypted, *i.e.,* it must contain the same values as s_{input}.

The described problem functions are modeled through processes in ForSyDe, having the following definitions:

- *genEnc*: generates an encryption function signal s_{encF} based on s_{key} signal;
- ap_{Enc}: encrypts the input signal s_{input} using the encryption function;
- *genDec*: generates the decryption function signal s_{decF}, based on s_{key}; and
- ap_{Dec}: decrypts the encrypted signal s_{enc} using the decryption function.

The ForSyDe/Haskell code implementing the EDS system is shown in Listing 3.3.

Listing 3.3. EDS SY MoC in ForSyDe/Haskell code.

```haskell
module EDS where
import ForSyDe.Shallow
-- Process Definitions
genEnc = combSY (+)
genDec = combSY (\x y -> y-x)
ap_enc = zipWithSY ($)
ap_dec = zipWithSY ($)
-- EDS process network
eds s_keys s_input = (s_enc , s_output)
  where s_enc = ap_enc s_encF s_input
```

```
        s_output = ap_dec s_decF s_enc
        s_encF = genEnc s_keys
        s_decF = genDec s_keys

-- to execute the model:
-- eds s_keys_sig s_input_sig
```

To verify consistency, the model was compiled using the GHC, which did not indicated any compilation warnings or errors.

Next, some input data sets are used to check whether the system behaves as expected. The first one is presented in Listing 3.4. The inputs are signals with the same number of events. The system behaved as expected, outputting the encrypted signal and the output signal, equals to the input signal.

Listing 3.4. EDS input example #1.

```
-- Input Example 1
s_keys_one = signal [1, 4, 6, 1, 1]
s_input_one = signal [1..5]
-- result: ({2,6,9,5,6},{1,2,3,4,5})
```

In the second input data set, two signals containing different number of events are inputted, as shown in Listing 3.5. In that case, the inputs are not totally synchronized, as SY MoC property states. ForSyDe handles this situation by taking the number of events contained in the smaller input signal. The output has the same number of events as the smaller input signal, as expected.

Listing 3.5. EDS input example #2.

```
-- Input Example 2
s_keys_two = signal [1, 4, 6]
s_input_two = signal [1, 2, 3, 4, 5]
-- result: ({2,6,9},{1,2,3})
```

Empty signals were used as input in the third data set. In that case, a runtime error was raised, generating the output shown in Listing 3.6.

Listing 3.6. EDS input example #3.

```
-- Input Example 3
s_keys_three = signal []
s_input_three = signal []
-- result:
-- <interactive>:22:5: error:
-- • Couldn't match expected type 'Signal Integer'
-- with actual type '[t0]'
-- • In the first argument of 'eds', namely 's_keys_three'
-- In the expression: eds s_keys_three s_input_three
-- In an equation for 'it': it = eds s_keys_three s_input_three

-- <interactive>:22:11: error:
-- • Couldn't match expected type 'Signal Integer'
-- with actual type '[t1]'
-- • In the second argument of 'eds', namely 's_input_three'
-- In the expression: eds s_keys_three s_input_three
-- In an equation for 'it': it = eds s_keys_three s_input_three
```

The last test detected a system inconsistency due to the lack of empty input signals pattern handling. The model was revisited to fix this problem, resulting in the code presented in Listing 3.7.

Listing 3.7. Revised EDS code.

```
module EDS where
import ForSyDe.Shallow
import Data.Maybe
import Data.Char

-- Process Definitions
genEnc = combSY (+)
genDec = combSY (\x y -> y-x)
ap_enc = zipWithSY ($)
ap_dec = zipWithSY ($)

-- EDS process network
eds:: Signal Int ->  Signal Int  ->  Maybe (Signal Int, Signal Int)
eds s_keys s_input =
  case s_keys of
    NullS -> Nothing
    x -> case s_input of
      NullS -> Nothing
      y  ->  Just (s_enc, s_output)
                    where s_enc = ap_enc s_encF y
                          s_output = ap_dec s_decF s_enc
                          s_encF = genEnc x
                          s_decF = genDec x
eds_fix key inp =
  case eds  key inp of
    Nothing ->  putStrLn "Empty Input Signal"
    Just q -> putStrLn (show q)
```

After the model review, it was recompiled, and the same tests were performed, generating no errors at the end, as a consequence, its consistency verification is finished.

3.3.5 Properties Verification

This step aims to identify the required properties for the system to comply with the selected MoC, and then verify whether the system holds the listed properties.

For the EDS, the minimum listed properties that must hold, and the verification code are presented as follows.

V_{SY1} *Signals are synchronized and totally ordered.*
Property verification – the used framework ForSyDe-Shallow represents the events only by its values, omitting the tags. For that reason, the signals are read and produced in a totally ordered manner, following the SY MoC semantics. If an input signal has less events than others, ForSyDe will process the smallest number of events, and the system signals will be synchronized. This verification was done by explicit model analysis, and model simulation;

V_{SY2} *Well-defined absent value for an event.*
Property verification – the used framework ForSyDe-Shallow manages absent values using the `ForSyDe.AbsentExt` type. Then, this verification was done by explicit model analysis, *i.e.,* as long as the system is modeled using the SY MoC semantics, this property holds;

V_{SY3} *For each loop-back, a delay must be implemented.*
Property verification – the EDS system does not comprehend any loop-back, *i.e.,* this property is not applicable for this example.

Besides the MoC properties, system specific properties can also be verified in this step, based on the problem statement:

- V_{SY_4}: System Output s_{output} must be equal to input signal s_{input}:

```
vsy4 :: Signal Int -> Signal Int -> Bool
vsy4 s_keys s_input = s_input == snd $ eds s_keys s_input
```

3.3.6 Implementation Details and Implement System

These steps are not included in AIPAA scope. However, the implementation of the EDS system including other behaviors such as runtime reconfiguration is presented in (Loubach et al., 2016).

3.4 Summary

This chapter presented the proposed *analysis and identification of possible automation approaches* (AIPAA) method, dividing the embedded system design flow into well-defined steps, analyzing and identifying its possible automation approaches. AIPAA comprehends just the specification domain, having as input the “problem statements” and as output the verified, simulateable and correct system model.

To illustrate the AIPAA application, a minimal example was presented, following the described method steps.

Next chapter presents a more detailed and complex system design case study to demonstrate the potential and applicability of the proposed AIPAA method.

4 AIPAA METHOD APPLICATION

This chapter presents a comprehensive case study demonstrating the potential and applicability of the proposed AIPAA method. Here, it is considered a more complex algorithm when compared to the illustrative example introduced in the last chapter.

The selected algorithm is a widely used lossless data compression one, namely *Lempel-Ziv-Markov Chain Algorithm* (LZMA), first used in 7z file format (Pavlov, 2019), and presented as a CPU benchmark by Standard Performance Evaluation Corporation (SPEC) (2019). LZMA intends to generate a compressed file based on the processing of a general data stream input.

4.1 LZMA - The Problem Statement

LZMA is composed by two compression algorithms. The first one is based on an optimized *sliding window encoding* (SWE), first presented in LZ77 algorithm (Ziv and Lempel, 1977). The second algorithm is based on the *range encoding* (RE) (Martin, 1979). An additional data filter can be added prior to LZMA default phases[1] towards the optimization of the compression rate, such as the *delta encoding* (DE) algorithm, which encodes each byte of the input stream as its difference from the previous byte. In this case, the first byte is not encoded.

The next sections presents the AIPAA method steps application to LZMA.

1. Problem characterization (Section 4.2);
2. MoC definition (Section 4.3);
3. Framework selection (Section 4.4);
4. Model and simulate (Section 4.5); and
5. Properties verification (Section 4.6).

[1]Here the word *phase* is in lieu of *step*, thus making step reserved for the AIPAA.

4.2 Problem Characterization

This step aims to divide the problem into functions and identify how they behave and communicate with each other. In this sense, the LZMA flowchart is illustrated in Fig. 4.1. Each LZMA encoding phase is described next.

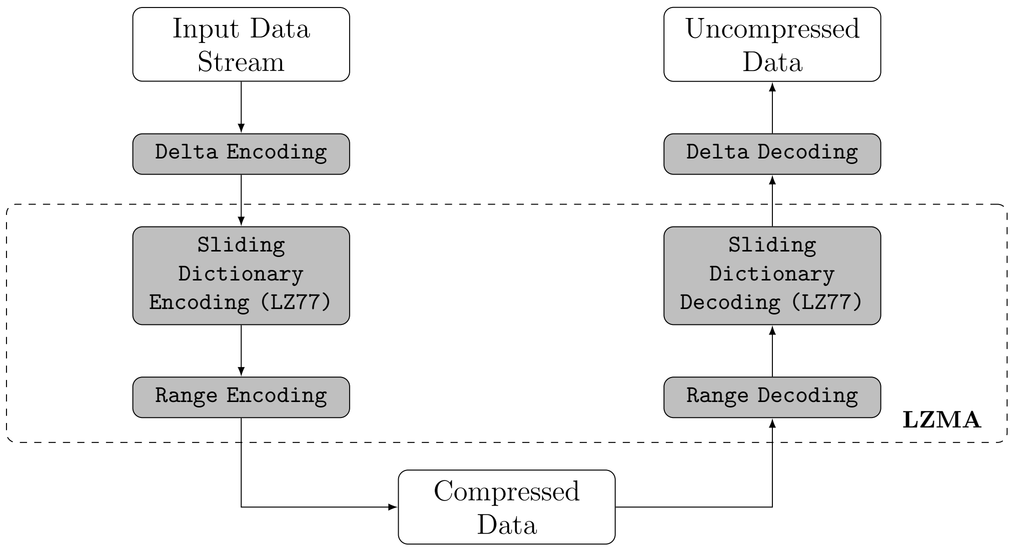


Figure 4.1. LZMA compression and decompression schemes, based on (Leavline, 2013).

4.2.1 Sliding Window Encoding

The first compression phase is based on LZ77, which searches for a match string of the current look-ahead buffer inside a determined maximum length window already processed. As the bytes are compressed, the look-ahead buffer shifts to the right, together with the search window, thus the name *sliding window*. For each compression cycle, the LZ77 outputs a tuple containing:

1. a *distance* representing the distance between the current byte and its match in the search window. It is equal to 0 if no match was found and it will be flushed in that case;

2. a *length* representing the match length, *i.e.,* how many bytes on the look-ahead buffer were repeated on its match located in the search window. It is equal to 0 if no match was found; and

3. a *next symbol* representing the next symbol in the look-ahead buffer to be processed.

The LZMA sliding window encoding implements additional features towards a higher compression rate, as presented next.

Dictionary Structure

When comparing LZMA to LZ77, the sliding window encoding supports larger dictionaries, which demands an optimized search structure targeting a faster algorithm. The LZMA implements two structure options that can be configured prior to the compression start, the *hash-chain* and *binary trees* (Salomon, 2007). Both of them are based on arrays of dynamically allocated lists, called *hash-arrays*. Each index of the array corresponds to a number *N* of bytes that are hashed from the input stream at each compression cycle.

The *hash-chain* array comprehends an array of linked lists. The nodes list contains the positions of the corresponding *N* hashed bytes of that array index in the input stream. The linked list can be long, and searching for the best match would result in a slow algorithm. For that reason, LZMA only checks for matches considering the 24 most recent positions.

In the *binary tree*, the nodes of its structure contains the same information as the hash-chain, with an optimized search structure, the binary tree. The most recent occurrences are closer to the tree root. Besides, the tree adopts the lexicographic ordering algorithm.

Without loss of generality, the present case study adopts the hash-chain array dictionary. When compared to the binary tree, the dictionary structure is simpler and the search algorithm is faster, since it limits the number of previously processed hash bytes analysis when finding a match.

Besides the dictionary structure, LZMA also employs an array containing the 4 last used match distances. If the distance of a match is equal to one of this array entries, the 2-bit array index is used to encode this variable (Salomon, 2007).

Output Format

The LZMA sliding window encoding comprehends a range of compressed packages possibilities that depends on the identified matches. Table 4.1 lists the output alternatives, presenting a brief description of each tuple.

4.2.2 Range Encoding

The range encoding algorithm was first presented by Martin (1979). It consists of a context-based compression algorithm in which the compressed range in each iteration is estimated based

Table 4.1. Sliding Window Encoding Output Packages, based on source code (Pavlov, 2019). The symbol $\oplus$ represents concatenation of binary values.

Package Code [base 2]	Package Name	Package Description
$0 \oplus byteCode$	`LIT`	One byte encoded using an adaptive binary range coder
$10 \oplus len \oplus dist$	`MATCH`	An LZ77 tuple describing sequence length and distance
1100	`SHORTREP`	One-byte LZ77 tuple. Distance is equal to the last used
$1101 \oplus len$	`LONGREP[0]`	An LZ77 tuple. Distance is equal to the **last** used
$1100 \oplus len$	`LONGREP[1]`	An LZ77 tuple. Distance is equal to the **second** last used
$11110 \oplus len$	`LONGREP[2]`	An LZ77 tuple. Distance is equal to the **third** last used
$11111 \oplus len$	`LONGREP[3]`	An LZ77 tuple. Distance is equal to the **fourth** last used

on probabilistic algorithms, and it can form a set of predefined types of packages depending on the input range size.

The LZMA range encoding performs a bit-wise compression of a sliding window tuple output at each fire cycle, outputting encoded bytes to the final output stream. It can be configured to update its bit probability at each process cycle, or to use fixed compression probability.

Based on the described LZMA functions and their relation, the analysis of the problem behavior is based on the listed questions in Section 3.1.1, which are answered as follows:

Q_1 - The problem has no concurrent processes, *i.e.,* the functions are sequentially executed;

Q_2 - The problem communication paths are partially ordered, *i.e.,* it is not possible to determine the sequence of tokens considering different data paths;

Q_3 - the model does not have any deadlock and will be finished only when there is no data in the input stream;

Q_4 - the processing of the input streams are cyclic and non-terminating; and

Q_5 - The functions cannot be synchronized by a clock, they depend on the data tokens outputted by other functions.

4.3 MoC Definition

An analysis of the problem characterization was performed to the MoC definition step. The LZMA compression comprehends the processing of an input stream, through defined phases, resulting in a compressed output stream. This behavior is classified as *untimed* and *concurrent*, which is according to *dataflow* MoC family semantics.

A first simplified LZMA model was introduced by the author of the present research in (Horita et al., 2019b) using the SDF MoC. That simplified LZMA modeling considered a set of assumptions to be valid. It was assumed that predefined fixed token rates for each actor port applies. Although it models the system process flow, it does not fully address the *dynamic behavior* of each compression phase, *i.e.,* both the sliding window and the range encoding.

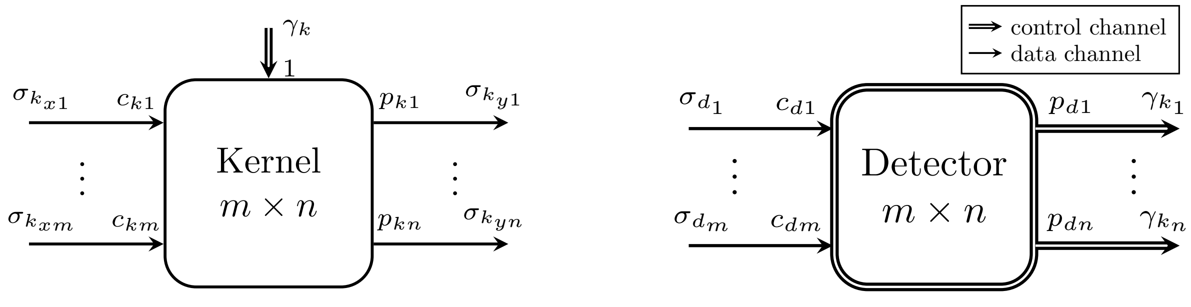
In this sense, the dynamic LZMA behavior can be better modeled based on the *scenario-aware dataflow* (SADF) MoC.

4.3.1 Scenario-Aware Dataflow (SADF) MoC

The SADF semantics was first presented by Theelen et al. (2006). It consists of a SDF generalization able to model dynamic system aspects by including the *scenarios* concept. SADF scenarios describe distinct modes of processes operation where the execution times and amounts of data can vary at each fire cycle. In this context, SADF classifies actors into two types: *kernels* and *detectors*.

Kernels are reconfigurable actors responsible for computation. Each kernel k has a set Ψ_k of scenarios. The scenarios $\psi \in \Psi_k$ selection is controlled by tokens consumed from an input control channel γ_k at each fire cycle. The kernel control channel input consumption token rate is always 1. At each kernel scenario, its data channels inputs consumption token rates c_k and outputs production token rates p_k can vary, ensuring the computation dynamism modeled by the kernels (Bonna et al., 2019). Fig. 4.2a illustrates a kernel.

Detectors are the actors responsible for determining the kernel scenarios. A detector d can have multiple *data input* channels σ_d, containing fixed token rates c_d, and multiple output channels, which can only be *control* channels γ_k, with production rates p_d, which serves as control inputs for kernel (Bonna et al., 2019). A detector is illustrated in Fig. 4.2b.



(a) Kernel **(b)** Detector

Figure 4.2. SADF actor types, both with m inputs and n outputs (Bonna et al., 2019).

4.4 Framework Selection

The framework selection consideration criteria were based on the same requirements of the illustrative example previously presented in Section 3.3.3. Thus, ForSyDe was used for the LZMA modeling and simulation.

For the sake of convenience, the points from Section 3.1.3 are also listed next.

- License style: the present research work focus is on open source tools;
- Included MoCs: the framework must include the selected MoC, *i.e.,* SADF;
- Capabilities: the proposed design includes both modeling and simulation;
- Interfaces: this case study accepts inputs and output from an user interface or from files;
- Scalability: this case study targets model high scalability. In this sense, a text-based coding interface is better than a GUI;
- Programming language paradigm: towards a higher level of abstraction, Haskell, a pure functional programming language, was selected as the framework base language.

A *functional model* for the scenario-aware dataflow (SADF) model of computation (MoC), as well as a set of abstract operations for simulating it was introduced in (Bonna et al., 2019). That MoC library is implemented on top of ForSyDe, and it is used in the present research to model and simulate the LZMA according to the SADF semantics.

4.5 Modeling and Simulation

The system case study is formal modeled in this step, using the previously selected MoC and framework, *i.e.,* SADF and ForSyDe. The model consistency is verified through its simulation aiming a correct-by-construction design, as described in Section 3.1.4.

This case study considers only the LZMA compression phase. Assuming that the decompression phase comprehends similar processes, it will not be modeled here.

4.5.1 SADF LZMA Model Description

The LZMA compression takes an input data stream, processes it through two compression phases and produces a compressed data output stream, as previously illustrated in Fig. 4.1.

When modeling LZMA using SADF MoC library from ForSyDe, the data streams are represented by *signals*, and the processing components by *kernels*. Besides, a *detector* is used to control the LZMA dynamic behavior. Fig. 4.3 illustrates the LZMA high level modeling dataflow graph based on SADF MoC.

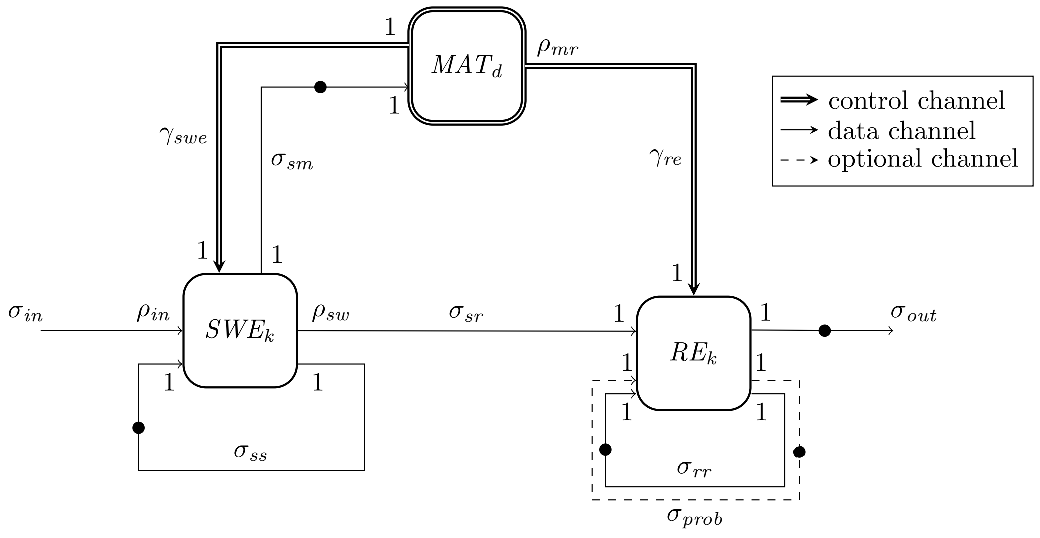


Figure 4.3. LZMA high level modeling dataflow graph based on SADF MoC. Initial tokens are represented by • .

Kernels, detector, signal paths, token rates and types in the model (Fig. 4.3) are defined as follows:

- *MAT_d* is the match detector, which controls the scenarios from both kernels, *SWE_k* and *RE_k*. In this sense, *MAT_d* comprehends two scenarios, *i.e.,* S_1 and S_2, as described in Table 4.5;
  - The input signal σ_{sm} transmits the current status of *SWE_k*;
  - The *SWE_k* scenarios, ψ_{swe}, are outputted to the signal γ_{swe}, according to Table 4.4;
  - Although *RE_k* contains a single scenario, ψ_{re}, the control channel γ_{re} is needed for detector *MAT_d* to reach a system feasible schedule. To model this behavior, its output token rate ρ_{mr} can vary from 0 to 1;
- *SWE_k* is the sliding window encoding kernel, which compresses the input stream by outputting processed tokens. Without loosing system main behavior and properties, the

present modeling adapts the SWE_k dictionary structure to a sliding window method based on the original LZ77 algorithm, aiming a simple model however preserving its behavior and properties.

- The σ_{in} is an input signal, representing a stream to be compressed;
- The consumption token rate ρ_{in} depends on the current SWE_k scenario, as described in Table 4.4. It can vary from 0 to 1 *Char*. Each *Char* represents a character from the input stream;
- The feedback signal σ_{ss} carries tuples $FB_{sw} = (Wind, Str, Dist)$. In this context, *Wind* represents the window dictionary, *Str* represents a *SWEData* package which is being processed, and *Dist* models the latest four used distances vector;
- σ_{sr} is a signal which carries the SWE_k produced tokens to RE_k;
- The production token rate ρ_{sw} can vary depending on the kernel scenario, from 0 to 1 *SWEData*, as described in Table 4.4. The possible *SWEData* formats are described in Tables 4.2 and 4.3;
- The control channel, signal γ_{swe}, carries tokens comprehending the possible SWE_k scenarios ψ_{swe}; and
- The signal σ_{sm} carries Boolean values that are either *true*, if a *SWEData* package was processed and is ready to be outputted, or *false* otherwise.

• RE_k is the range encoding kernel, which performs a bit-wise compression of *SWEData* packages, outputting encrypted bytes to LZMA output stream. Although its token rates are all constants, its fire cycles are controlled by the detector MAT_d;

- The RE_k input port, connected to the signal σ_{sr}, has a fixed consumption token rate equals to 1 *SWEData*;
- The input control port, connected to the signal γ_{re}, has a fixed consumption token rate of 1;
- The output signal σ_{out} carries the LZMA output stream, composed by encrypted bytes produced by RE_k. The RE_k fixed output token rate is 1 *Byte*, consisting on a signal of bytes;
- Signal σ_{rr} carries tuples that contains the relevant variables for the range encoding, *i.e.,* $FB_{re} = (Range, Low, Cache)$. In this context, *Range* represents the considered range when encoding the next bit. *Low* represents the lower limit of the considered range for the next bit encoding. *Cache* represents a processed value to be outputted to σ_{out} in the next range encoder token production;
- Signal σ_{prob} (dashed line in Fig. 4.3) represents the bit probabilities updated at each RE_k compression cycle, in case the variable probability is configured. Although the

fixed probability configuration has a reduction in the compression rate performance taking into account some cases where the bits patterns are often repeated, there is no loss of generality when considering fixed bit probabilities. In this sense, the present work assumes the range encoder configured to use fixed probability encoding; and

– The initial token in the σ_{out} signal is the compressed file header, which includes LZMA configuration, dictionary size and decompressed file size.

Table 4.2. σ_{sr} possible token formats.

Package	**Model** *SWEData*	**System Code**
`LIT`	$LIT \oplus Char$	$0 \oplus byteCode$
`MATCH`	$MAT \oplus Int \oplus Int$	$10 \oplus l \oplus dist$
`SHORTREP`	$SREP$	1100
`LONGREP[0]`	$LREP0 \oplus Int$	$1101 \oplus l$
`LONGREP[1]`	$LREP1 \oplus Int$	$1100 \oplus l$
`LONGREP[2]`	$LREP2 \oplus Int$	$11110 \oplus l$
`LONGREP[3]`	$LREP3 \oplus Int$	$11111 \oplus l$

Table 4.3. l variable possible sizes.

l **format**	len **range**
$0 \oplus 3\ bits$	$2 < len < 9$
$10 \oplus 3\ bits$	$10 < len < 17$
$11 \oplus 8\ bits$	$18 < len < 273$

Table 4.4. SWE_k Scenarios and token rates discrimination.

Ψ_{swe} **Scenarios**	**Rates**	
	ρ_{in}	ρ_{sw}
Search	1	0
Flush	0	1

Table 4.5. MAT_d output control channels token rates and scenarios discrimination.

Ψ_{mat} **Scenarios**	**Rates [Kernel Scenario]**	
	γ_{swe}	γ_{re}
S_1	1 [Search]	0
S_2	1 [Flush]	1 [Read]

Based on the described definitions, this modeling considers two system scenarios, which are controlled by the detector MAT_d.

4.5.2 LZMA Modeling with ForSyDe SADF MoC

ForSyDe is used to system modeling based on the model description previously stated and illustrated in the dataflow graph from Fig. 4.3. The model main definitions and processes (*i.e.,* kernel, detector, actor) signatures are presented in Listing 4.1, using the process constructors from the ForSyDe SADF MoC library. In this context, the following concepts are used:

- `SWEData` represents de kernel SWE_k output token type, which can be derived to all the *SWEData* package types;

- SWEkScenario and REkScenario models the kernels SWE_k and RE_k scenarios, respectively;
- RangeVars models the tokens contained in the feedback signal FB_{re}, which comprehends the current encoding *Range*, its *LowerLimit* value and the *Cache*, representing a stored byte to be outputted when the next output token is produced;
- matD models the detector MAT_d using the detector12SADF, with one input port and two control output ports;
- sweK models kernel SWE_k using the process constructor kernel23SADF, indicating it contains two data input ports and three data output ports; and
- rek represents the range encoder kernel RE_k, modeled using the kernel22SADF, indicating it contains two data input ports and two data output ports;

Listing 4.1. LZMA model signatures and definitions in ForSyDe.

```
-------------------------- SWEData definition --------------------------
data SWEData = LIT Char | MAT Int Int | SREP |
                LREP0 Int | LREP1 Int | LREP2 Int | LREP3 Int deriving (Eq, Show)

------------- SWE and RE Kernels and MAT detector definitions -------------
-- SWE scenarios definition
type SWEkScenario = ((Int, Int), (Int, Int, Int), [Char] -> [(String, String, [Int])]
-> ([Bool], [SWEData], [(String, String, [Int])]))

-- Range Encoder Feedback = (range, Lower Limit, cache)
type RangeVars = (Int,Int,Char)

-- RE scenarios definition
type REkScenario = ((Int, Int), (Int, Int), [SWEData]
-> [RangeVars] -> ([Char], [RangeVars]))

-- matD (Match Detector) definition
-- Input: identifier of the SWE_k current status
-- Output: matStateOut containing SWE_k and RE_k scenarios
matD :: Int -> Int -> Signal Bool -> (Signal SWEkScenario, Signal REkScenario)
matD b_size w_size = detector12SADF 1 matStateTran (matStateOut b_size w_size) Search

-- sweK (Sliding Window Encoding Kernel) definition
-- Input: Control Channel, Input Stream, feedback signal
-- Output: SWEData, feedback signal, Current status
sweK :: Signal SWEkScenario -> Signal Char -> Signal (String, String, [Int])
    -> (Signal Bool, Signal SWEData, Signal (String, String, [Int]))
sweK = kernel23SADF

```

```
-- reK (Range Encoding Kernel) definition.
-- Input: Control Channel, SWEData from SWE_k, feedback signal
-- Output: Compressed Bytes Signal, feedback signal
reK :: Signal REkScenario -> Signal SWEData -> Signal RangeVars
    -> (Signal Char,Signal RangeVars)
reK = kernel22SADF
```

Each actor is then presented in separate listings. The LZMA detector model MAT_d main functions and definitions are presented in Listing 4.2, with the following definitions:

- `FDscenario` represents the detector scenario, comprehending the output token rates of the control channels ports and the sent kernels scenarios. `FDstate` models the detector state, based on the kernel SWE_k boolean output through σ_{sm}; and
- `matStateTran` represents the detector function to select the MAT_d scenarios based on a Boolean input, and `matStateOut` outputs the SWE_k and RE_k scenarios.

Listing 4.2. LZMA detector MAT_d model functions in ForSyDe.

```

---------- Detector State transition and output functions ------------
-- data SWEState = Search | Flush
-- data REState = ReRead
type FDscenario = ((Int, Int), ([SWEkScenario], [REkScenario]))
data FDstate = Search | Flush

matStateOut :: Int -> Int -> FDstate -> (FDscenario) -- (Int, [SWEkScenario],Maybe [
    REkScenario])
matStateOut b_size _ Search =
      ((1,1), ([((1,1), (1,0,1), sweSearch b_size)], [((0,0),(0,0), rekStdBy)]))
matStateOut _ w_size Flush =
      ((1,1), ([((0,1), (1,1,1), sweFlush w_size)], [((1,1),(1,1), rekRead)]))

matStateTran :: FDstate -> [Bool] -> (FDstate)
matStateTran _ [True] = (Search)
```

Listing 4.3 introduces the main functions used to model the LZMA sliding window kernel SWE_k , with the following definitions:

- `sweSearch` and `sweFlush` model the functions executed by `sweK` depending on its scenario, where:
  - `sweSearch` represents the scenario in which SWE_k consumes a character from input signal when searching for a match; and
  - `sweFlush` models the output of a found *SWEData* package to signal σ_{sr}.

Listing 4.3. LZMA Sliding Window Kernel main functions in ForSyDe.

```

------------------------ SWE Kernel functions ----------------------------
sweSearch :: Int -> [Char] -> [(String, String, [Int])]
                 -> ([Bool], [SWEData], [(String, String, [Int])])
sweSearch b_size [c] [(win, buff, distList)]
  | stringRFind (buff++[c]) win == Nothing || length (buff++[c]) > b_size =
                ([False], [], [(win, buff++[c], distList)])
  | otherwise = ([True], [], [(win, buff++[c], distList)])

sweFlush :: Int -> [Char] -> [(String, String, [Int])]
                -> ([Bool], [SWEData], [(String, String, [Int])])
sweFlush w_size [] [(win, buff, distList)]
  | l == 0 = ([True], [LIT (head buff)], [(drop n (win ++ buff), "", distList)])
  | l == 1 = if isNothing (stringRFind (init buff) win)
    then ([True], [LIT (head buff)], [(win', [last buff], distList)])
    else if head distList == dist
      then ([True], [SREP], [(win', [last buff], distList)])
      else ([True], [LIT (head buff)], [(win', [last buff], distList)])
  | distList !! 0 == dist = ([True], [LREP0 l], [(win', [last buff], distList)])
  | distList !! 1 == dist = ([True], [LREP1 l], [(win', [last buff], distList)])
  | distList !! 2 == dist = ([True], [LREP2 l], [(win', [last buff], distList)])
  | distList !! 3 == dist = ([True], [LREP3 l], [(win', [last buff], distList)])
  | otherwise = ([True], [MAT dist l], [(win', [last buff], take 4 (dist:distList))])
  where n = max 0 $ length (win ++ buff) - w_size
        win' = drop (max 0 $ length (win ++ init buff) - w_size) (win ++ init buff)
        dist = fromJust (stringRFind (init buff) win)
        l = length buff - 1
```

The range encoder main functions and definitions are presented in Listing 4.4 with the following definitions:

- `encBit` models the bit-wise processing of the *SWEData* tokens, updating the `RangeVars` variables;
- `normRange` models the normalization procedure, executed when the range reaches its lower limit and needs to be re-scaled, producing one byte that will be outputted to the σ_{out} signal, temporarily stored in `Cache`;
- `rekStdBy` models the standby state of kernel RE_k, while waiting for a *SWEData* token; and
- `rekRead` models the function to read a *SWEData* token and encrypt it to σ_{out}.

Listing 4.4. LZMA Range Kernel main functions in ForSyDe.

```
        k = x `div` m
------------------- RE Kernel functions -------------------
normRange:: [Char] ->[RangeVars] -> ([Char], [RangeVars])
normRange nSout [(nRan,nL,nC)]
  | nRan < rangeLimit =
    if low < 0xFF000000 then
      ( nSout ++ [nC], [(nRan  * 0x100, low  * 0x100, chr cache)])
    else if nL > 0xFFFFFFFF then
      ( nSout ++ [chr (ord nC + 1)], [(nRan  * 0x100, low  * 0x100, chr cache)])
    else
      (nSout,[(nRan  * 0x100,nL,nC)])
  | otherwise = (nSout,[(nRan,nL,nC)])
  where high = quot (nL .&. 0xFFFFFF00000000) 0x100000000
        low = (nL .&. 0xFFFFFFFF)
        cache = quot (nL .&. 0xFF000000) 0x1000000

encBit :: [Bool] -> ([Char],[RangeVars]) -> ([Char], [RangeVars])
encBit [] (a,b) = (a,b)
encBit eIn (encSigOut, [(eRan,eL,eC)])
  | head eIn == True =
    encBit (tail eIn) (normRange encSigOut [(newRan,eL + (newRan),eC)])
  | head eIn == False =
    encBit (tail eIn) (normRange encSigOut [(newRan,eL,eC)])
  where newRan = quot eRan 2

toSigSig :: ([Char],[RangeVars]) -> ([[Char]],[RangeVars])
toSigSig ([],b) = ([],b)
toSigSig (a,b) = ([a],b)

rekStdBy :: [SWEData] -> [RangeVars]-> ([[Char]],[RangeVars])
rekStdBy _ fb = ([],fb)

rekRead :: [SWEData] -> [RangeVars]-> ([[Char]],[RangeVars])
rekRead [] s_rr = ([],s_rr)
rekRead [(LIT lit)] s_rr =
  toSigSig (encBit ([False] ++ (toBitsBySize 8 (fromEnum lit))) ([] , s_rr))
rekRead [(MAT x y)] s_rr =
  toSigSig (encBit ([True,False] ++ (toBitsBySize 8 x) ++ (toBitsBySize 32 y)) ([] ,
    s_rr))
rekRead [(SREP)] s_rr =
  toSigSig (encBit [True,True,False,False] ([] , s_rr))
rekRead [(LREP0 x)] s_rr =
  toSigSig (encBit ([True,True,False,True] ++ (toBitsBySize 8 x)) ([] , s_rr))
rekRead [(LREP1 x)] s_rr =
  toSigSig (encBit ([True,True,True,False] ++ (toBitsBySize 8 x)) ([] , s_rr))
rekRead [(LREP2 x)] s_rr =
  toSigSig (encBit ([True,True,True,True,False] ++ (toBitsBySize 8 x)) ([] , s_rr))
rekRead [(LREP3 x)] s_rr =
  toSigSig (encBit ([True,True,True,True,True] ++ (toBitsBySize 8 x)) ([] , s_rr))
rekRead _ s_rr =
```

Finally, the complete process network is introduced in Listing 4.5. The `lzmaCompressPn` process models the LZMA compression, connecting the `sweK` output to the `reK` input. The compressed stream is modeled as `sig_out` and the input stream as `sig_in`. In addition, the sliding window encoder process network is also presented as `swePn` to illustrate and test this compression step separately.

Listing 4.5. LZMA compression process network.

```
---------------------- SWE process network -------------------------
swePn :: Int -> Int -> Signal Char -> Signal SWEData
swePn b_size w_size s_in = s_out
  where (s_det , s_out , s_fb) = sweK ct s_in s_fb'
        (ct,rect) = matD b_size w_size s_det'
        s_det' = delaySADF [True] s_det
        s_fb' = delaySADF [("","",[-1,-1,-1,-1])] s_fb

---------------------- LZMA process network -------------------------
lzmaCompressPn ::  Int -> Int -> Signal Char -> Signal [Char]
lzmaCompressPn b_size w_size sig_in = sig_out'
    where   (sig_out,s_rr) = reK ctRe s_swetok s_rr'
            (s_det , s_swetok , s_sweFb) = sweK ctSwe sig_in s_sweFb'
            (ctSwe,ctRe) = matD b_size w_size s_det'
            s_det' = delaySADF [True] s_det
            s_sweFb' = delaySADF [("","",[-1,-1,-1,-1])] s_sweFb
            s_rr' = delaySADF [(rangeInit , 0 , chr 0)]  s_rr
            sig_out' = delaySADF [([defConfig] ++ (inttoCharList 4 defDictSize)
              ++ (inttoCharList 8 (lengthS sig_in)))] sig_out
```

4.5.3 Model Simulation

Towards the verification of system consistency and behaviors in accordance to expected, some input stream examples are used to test the LZMA model. This step is based on the GHCi usage.

Listing 4.6 introduces the defined processes to simulate the model. `sweOutTest` models the sliding window output, and `lzmaOutTest` the LZMA compression output. For a printable character set, `lzmaOutTest` in outputted as `Integer` values, based on the ASCII table.

Listing 4.6. LZMA compression test processes.

```
sweOutTest = swePn buffer_size window_size inputTest
```

```
lzmaOutTest = mapSY ord (lzmaCompressPn buffer_size window_size inputTest)
```

The first example consists on a sequence of five characters, non repeated. Its simulation was successfully accomplished, as presented in Listing 4.7. The model behaves as expected. The SWE_k outputs the correct packages, *i.e.,* literals, as it have no repetitions. The LZMA outputs the expected stream of encoded bytes, composed by an initial header followed by a sequence of byte lists, grouped by the output of each RE_k cycle.

Listing 4.7. LZMA compression input example #1.

```
inputTest = signal "12345\x00"
-- SWE output
sweOutTest
{LIT '1',LIT '2',LIT '3',LIT '4',LIT '5'}
-- LZMA output
lzmaOutTest
{[93,48,48,48,49,48,48,48,48,48,48,48,53],[0],[23],[139],[133],[96]}
```

The second example comprehends the inclusion of a *SWEData* package of a found match. Listing 4.8 presents the simulation results. The model also behaves as expected in this example, identifying the match in the previous buffer and successfully encoding all the packages.

Listing 4.8. LZMA compression input example #2.

```
inputTest = signal "abracadabra\x00"
-- SWE output
sweOutTest
{LIT 'a',LIT 'b',LIT 'r',LIT 'a',LIT 'c',LIT 'a',LIT 'd',MAT 6 4}
-- LZMA output
lzmaOutTest
{[93,48,48,48,49,48,48,48,48,48,48,48,99],[0],[47],[151],[141],[67],
        [122],[138],[253,54,123,233,181,61]}
```

In the third example, the empty input boundary situation is tested, as presented in Listing 4.9. The SWE_k outputs an empty package to RE_k, which in turn, just outputs the header, without any data to be compressed, as expected.

Listing 4.9. LZMA compression input example #3.

```
inputTest = signal ""
-- SWE output
sweOutTest
{}
-- LZMA output
lzmaOutTest
{[93,48,48,48,49,48,48,48,48,48,48,48,48]}
```

As the model compiles with no errors and behaves as expected in all the presented simulations, its consistency verification is finished.

4.6 Properties Verification

As stated in Section 3.1.5, this step aims to check whether the system model holds the required properties, with respect to the in-use MoC. In this sense, this section introduces the SADF MoC properties and their verification methods, also presenting the LZMA model verification.

4.6.1 SADF MoC Properties and Verification Methods

According to Theelen et al. (2006), a model based on SADF MoC must hold the following properties:

P_{SADF1} *Boundedness*: this property is similar to the P_{SDF_1}, presented in Section 3.1.5. The system model production and consumption token rates must be designed in a manner that the number of buffered tokens are bounded;

P_{SADF2} *Absence of deadlocks*: As SDF, the SADF must be also non-terminating, *i.e.,* it must not have deadlocks; and

P_{SADF3} *Determinacy*: the model performance depends on probabilistic choices that determine the sequence of scenarios selected by each detector. As discussed by Bonna et al. (2019), in the *performance* SADF model a detector's behavior is represented by a Markov chain. On the other hand, in the *functional* SADF model, representing the detector's behavior as a Markov chain would violate the tagged signal model definition's of *functional process*, since a Markov chain describes more than one behavior, *i.e.,* it is non-deterministic. In view of this, the functional model is used in the present work, where the detector's behavior is dictated by a deterministic finite-state machine.

Yet, Theelen et al. (2006) introduces the described properties verification methods, as presented next:

V_{SADF1} The boundedness verification considers three aspects:

(a) *Boundedness in fixed scenarios case*: In case all processes only operate in a fixed scenario, the system behaves as a SDF based model, and the boundedness verification is performed as described in Section 3.3.5;

(b) *The boundedness of control channels*: For each kernel k, controlled by a detector d, the inequality $E_{d,s} \geq \gamma(k,s) \times E_{k,s}$ must hold for every scenario s in which k is inactive, where $E_{d,s}$ represents the duration of d to detect and send the scenario s to kernel k, $\gamma(k,s)$ represents how many cycles the k computation is executed in the scenario s, and $E_{k,s}$ is the duration of each k computation cycle; and

(c) *The effect of the scenario changes on boundedness*: The number of tokens in channels between active processes after an iteration of the SADF model is the same as before. In this context, an iteration is the firing of each process p configured in a scenario s for $\gamma(p,s)$ times.

V_{SADF2} For all kernels' scenarios combinations, there must have sufficient initial tokens in each cyclic dependency between processes such that every included process can fire a number of times equal to its repetition vector entry. Feedback loops can be considered as a cyclic dependency of a single process; and

V_{SADF3} The functionality non-determinism only occurs between multiple independent concurrent processes. If existing in a model, it leads to satisfying the *diamond property*. This property states that when two or more independent processes are concurrently enabled, the system functionality is not affected by the order that they are performed. This statement proof is fully described by Theelen et al. (2006).

4.6.2 LZMA Model Properties Verification

This step submits the LZMA model to the stated SADF MoC verification methods.

V_{SADF1} *Boundedness*
Property verification – The three verification aspects are considered for the presented LZMA model:

(a) The first aspect is used for verification of models where all the processes comprehend a single scenario. In the LZMA case, the kernel SWE_k and detector MAT_d do not operate in a single scenario, as described in Section 4.5.1. In this sense, this aspect is not applicable;

(b) The LZMA kernel RE_k becomes inactive during the detector MAT_d scenario S_1. In this model, the range encoder fires a single time for each S_1 scenario selection ($\gamma(RE,S_1) = 1$). For the model to be bounded, the RE_k duration E_{RE} has to be smaller than the duration E_{MAT,S_1} between two scenarios S_1 detection by MAT_d;

(c) The third aspect must be analyzed focusing on kernel SWE_k and detector MAT_d, as these are the only processes with multiple scenarios. The interface between these processes consists on a cyclic dependency with fixed and equal token rates, not representing an unboundedness risk. Regarding the boundedness of signal σ_{sr}, the production token rate from SWE_k output port is controlled by MAT_d, and the consumption token rate from RE_k is always one. In MAT_d S_1 scenario, SWE_k will not produce any tokens in σ_{sr}, and in S_2 scenario, SWE_k will produce one token that will be promptly consumed by RE_k in the same iteration. In this context, the third boundedness aspect will always hold, as MAT_d controls the system schedule, so that SWE_k will not produce another token before RE_k consumes the last one.

V_{SADF2} *Absence of deadlocks*

Property verification – Analyzing the LZMA model, the included cyclic dependency are the feedback signals σ_{ss}, σ_{prob} and σ_{rr}, in addition to the SWE_k and MAT_d processes cycle. Initial tokens were added to each of these cycles, satisfying the necessary consumption token rates. As a consequence, it is verified that the LZMA system holds this property.

V_{SADF3} *Determinacy*

Property verification – The LZMA does not comprehend multiple independent concurrent processes, as the SWE_k depends on MAT_d and vice-versa. Besides, RE_k also depends on SWE_k. As a consequence, the system holds the determinacy property.

In addition to the introduced properties verification arguments V_{SADF_2} and V_{SADF_3}, Table 4.6 presents a feasible LZMA *static scheduling*, representing a possible sequence of determined running processes, starting with an initial token in signal σ_{sm}, and without any deadlock. The schedule finishes when input stream σ_{in} is empty.

Table 4.6. Feasible LZMA system static scheduling.

σ_{sm} token	S			S			F				S			...
Running Processes		MAT_d	SWE_k		MAT_d	SWE_k		MAT_d	SWE_k	RE_k		MAT_d	SWE_k	...
MAT_d Scenarios			S_1			S_1			S_2				S_1	...

σ_{sm} token: S = Search, F = Flush

4.7 Implementation Details and Implement System

The implementation domain is not part in AIPAA scope. Nevertheless, different programming languages, implementation architectures and platforms have been used to implement the LZMA in other related works. But, no implementation was found based on a functional programming language, such as Haskell.

The first LZMA implementation was presented in the *7zip* open source application, able to be executed using different platforms, including a range of compression algorithms, and having LZMA as default (Pavlov, 2019; Salomon, 2007). Another similar application is presented in *XZ Utils*, having less languages and configuration options (Collin, 2019). Besides the mentioned applications, hardware description languages (HDL) implementations for FPGA are also available in the literature (Leavline, 2013; Zhao and Li, 2017). A benchmark comparing different implementation architectures and platforms was presented by SPEC.

4.8 Summary

This chapter presented a comprehensive case study showing the *Analysis and Identification of Possible Automation Approaches* (AIPAA) method application, which is proposed in this research work.

All the AIPAA defined steps were followed aiming the correct-by-construction system design considering the specification domain, however taking into account an executable SADF model.

The selected application for this case study was the compression procedure contained in the *Lempel-Ziv Markov chain Algorithm* (LZMA). That is a widely used algorithm presented as a benchmark by SPEC.

The next chapter presents a discussion and analysis on the AIPAA method, its applicability and limitations.

5 RESULTS, ANALYSIS AND DISCUSSION

This chapter presents this research work results along with analysis and discussion. The achieved results are based on the requirements previously stated in Section 1.3, together with the developed illustrative example (Section 3.3) and case study (Chapter 4).

5.1 Literature Review

According to R_0, the first requirement comprehends the *literature review with respect to the main concepts and theory on MoC and high level modeling*. The review results are present in Chapter 2, including the concepts of embedded systems and cyber-physical systems, MoCs, formal-based modeling frameworks, and functional and imperative programming paradigms, besides the main existing automatic code generation tools and other research works related to the present one.

5.2 Case Studies Specification

Embedded systems case studies complete specification is listed as requirement R_1. The present work specified two case studies. The encoder-decoder system (EDS) and the Lempel-Ziv Markov chain algorithm (LZMA).

As illustrated in Fig. 3.1, the AIPAA method considers as input the problem statement and guides one to perform an initial analysis, *i.e.,* a problem characterization to identify system behaviors and characteristics, as described in Section 3.1.1.

EDS and LZMA complete specifications, based on the problem statement and characterization, are presented in Section 3.3.1 and Section 4.1, respectively. Those detailed specifications are the base towards an optimized formal-based system design flow.

5.3 Framework Selection Criteria

The requirement R_2 comprehends a *formal-based modeling framework selection criteria and list of candidates tools*. AIPAA includes a step focused on this matter, as presented in Section 3.1.3. The author of this work published a paper on this subject, named *Analysis and Comparison of Frameworks supporting Formal System Development based on Models of Computation* Horita et al. (2019a).

A range of frameworks were considered for comparison, *i.e.,* Ptolemy II, ForSyDe, SDF3 and Simulink. AIPAA lists specific frameworks characteristics that must be considered when choosing the modeling tool for a given application.

One of the facts to be taken into consideration when selecting a framework is its included MoCs, analyzing if the used/selected MoC is supported. In this context, AIPAA also includes a step describing the design MoC definition, as presented in Section 3.1.2.

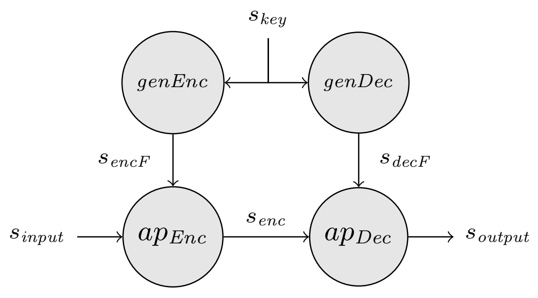
For the present work case studies, the MoCs that best describes the systems behaviors were SY for EDS and SADF for LZMA. Based on the MoCs selection and other relevant analyzed facts, ForSyDe was the selected framework. Section 4.4 describes in details the ForSyDe characteristics took into consideration for its selection.

5.4 Case Study Modeling Based on Formal Methods

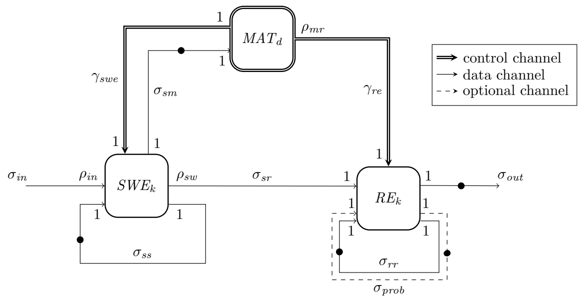
The requirement R_3 includes the *embedded system case study modeling and high level implementation, based on formal design method*. AIPAA includes two steps aiming the formal-based modeling: the system model *functional* correctness verification and the system consistency. The referred steps are *model and simulate* and model *properties verification*, described in Sections 3.1.4 and 3.1.5, respectively.

Section 3.3.4 presents the modeling and simulation of an illustrative example comprehending the EDS, based on the ForSyDe SY MoC library, as illustrated in Fig. 3.3, which is replicated in Fig. 5.1a for the sake of convenience. That section also presents how the simulation can detect an erroneous model behavior. The model properties were then verified in Section 3.3.5 following AIPAA method, to prove the model holds SY MoC properties.

An elaborated case study was based on a paper published by the author of this work, which modeled the LZMA compression using the ForSyDe SDF MoC library, not representing part of the system dynamic behavior (Horita et al., 2019b). The present work modeled the same algorithm, but now using the ForSyDe SADF MoC library, as described in Section 4.5 and illustrated in Fig. 4.3, which is replicated in Fig. 5.1b for the sake of convenience, capturing its compression procedure dynamism. Besides, AIPAA method was used to prove the model holds SADF MoC properties.



(a) Encoder-decoder system SY MoC dataflow graph.



(b) LZMA system SADF MoC dataflow graph.

Figure 5.1. Formal-based models dataflow graphs.

The AIPAA method comprehends the *specification domain*, which outputs a verified high level abstraction *executable model*.

The implementation domain consists of an extensive subject and, although it is not included in AIPAA, Section 3.1.6 presents directives on how to overcome the abstraction gap and implement a system based on a verified model. Besides, for both illustrative example and case study, implementation related works were cited as a reference to guide possible future researches.

5.5 Analysis Identifying Steps to be Automated

An analysis identifying implementation steps to be automated is presented to fulfill the requirement R_4. This research introduced design method was presented under the name *analysis and identification of possible automation approaches* (AIPAA).

In view of this, AIPAA includes well-defined design steps on a formal base, making the automation approaches identification feasible and possible to be systematically analyzed.

As a result, this research introduced in Section 3.2, as passable of automation steps the *properties verification* and the *automatic code generation* based on a formal and verified model, as described in Sections 5.5.1 and 5.5.2.

5.5.1 Properties Verification with "Quickcheck"

AIPAA includes a step comprehending the system model properties verification, as described in Section 3.1.5. Its main purpose is to prove that the model holds the properties of the selected

MoCs and possibly a set of system specific properties. In this sense, a range of methods can be applied to verify if the system holds each property, *e.g.,* mathematical prove and model analysis.

A widely used method for properties verification is the system submission to a range of test cases, which can be a labor intensive task. Besides, for a robust verification, different tests must be elaborated aiming the detection of different fail cases.

The present research identifies the automation of this verification method as a possibility. In this sense, Section 3.2.1 presents *Quickcheck* as an alternative for automatic test cases generation and system property verification. A minimal model example based on SDF MoC was presented, using *Quickcheck* for verification whether the system holds the property P_{SDF_4}, introduced in Section 3.1.5, as an example of its applicability.

5.5.2 Automatic Code Generation

Although the implementation domain is not included in AIPAA, this research recognizes the abstraction gap between the formal-based model outputted by the introduced method and the implemented system and the difficulties to overcome it. In this sense, this research proposes, as an embedded design automation approach, the automatic low level code generation based on a verified formal-based model. This represents an extensive work that have been studied in related works and can be deeper explored as future works, as mentioned in Section 3.2.2.

5.6 Scientific Publications

The author of this work published two conference papers as results of the present research.

An analysis and comparison of two frameworks supporting formal system development based on MoCs was introduced in (Horita et al., 2019a). The compared frameworks had to be open source and they had to support both modeling and simulation. Based on that criteria, ForSyDe and PtolemyII were compared considering a range of aspects, *i.e.,* programming paradigm, scalability and design methodology. To support the comparison, a case study composed by two systems was modeled using two widely used MoCs, SY and SDF. The benefits and drawbacks of each framework were listed based on the paper comparison, which can be considered when choosing the best development tool for an application.

- Horita A.Y., Bonna R., Loubach D.S. (2019) Analysis and Comparison of Frameworks Supporting Formal System Development based on Models of Computation. In: Latifi S. (eds) 16th International Conference on Information Technology-New Generations

(ITNG 2019). ADVANCES IN INTELLIGENT SYSTEMS AND COMPUTING, vol 800. Springer, Cham.

The LZMA compression routine was modeled using ForSyDe SDF MoC library in (Horita et al., 2019b). The selected algorithm and modeling framework are the same as this research case study, although fixed actors token rates were adopted to consider a simple model in the paper. The SADF model from this research case study models the LZMA adaptivity in a more complete and adequate manner.

- 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. 10th Aerospace Technology Congress 2019 (FT2019). FTF - Swedish Society of Aeronautics and Astronautics. Stockholm, Sweden. (to appear)

5.7 Summary

This chapter discussed the main results of this research based on the requirements enumerated in Section 1.3. The AIPAA findings were also presented indicating a possible automation in formal-based embedded systems design flow.

The next chapter will present this research work conclusions, author recommendations and possible future works.

6 CONCLUSION

This final chapter presents the main conclusion, contributions, author recommendations and future works suggestions towards AIPAA improvements.

6.1 Specific Conclusions

The main goal of this work was:

> **to identify possible automation approaches for design flows based on formal models of computation**, aiming to a *future* automatic code generation and also a reliable, robust and scalable embedded systems design flow.

In this sense, it was proposed a method for *analysis and identification of possible automation approaches* (AIPAA) applicable to embedded systems design flow supported by formal models of computation.

In order to develop a deeper analysis method, a review of the formal-based design flow concepts was performed, including embedded systems and CPS, modeling frameworks, MoCs topology and programming paradigms. Moreover, a research of the existing automation tools and related works was conducted.

There exist tools and methods of specific procedures and design steps automation, such as low level code generation. However, there are not many studies which analyze the embedded systems design flow depicted into well-defined steps.

The analysis of the design flow specification domain, presenting delimited steps towards the analysis and identification of their possible automation approaches is a characteristic that differs the present work from previous ones.

In this context, the AIPAA methodology was introduced. It includes five design steps, as illustrated in Fig. 3.1. Each step is then explained in details. For the sake of convenience, the main purpose of each step is described next.

1. *Problem characterization*: identify the relevant behaviors and characteristics of the target problem, defining intermediate functions that can represent the system processes and their relation;

2. *MoC definition*: based on problem characteristics, select a MoC to be used in the system design flow;

3. *Framework selection*: define the formal-based framework for system modeling and simulation;

4. *Model & simulate*: model the specified system, simulating it to verify its consistency;

5. *Properties verification*: verify that the system model holds the defined MoC properties.

To demonstrate the AIPAA application, an illustrative example and a case study system were presented, namely *Encoder-Decoder System* (EDS) and *Lempel-Ziv Markov Chain Algorithm* (LZMA).

The AIPAA application results were discussed in Chapter 5, identifying the *properties verification* and *automatic code generation* as possible embedded systems design flow automation approaches, coping with this work main goal.

6.1.1 Work Requirements Traceability

The requirement R_0 was covered in Chapter 2, by describing the main used concepts in this work.

The requirement R_1 was covered in Section 3.3.1 and Section 4.1, presenting the specification of an illustrative example and a case study, the EDS and LZMA, respectively.

The requirement R_2 was covered in a paper published by the author of this work (Horita et al., 2019a), describing benefits and drawbacks of selected frameworks supporting formal MoCs. This method was also described in Section 3.1.3.

The requirement R_3 was covered in Chapter 4, where the AIPAA was applied to a case study, the LZMA compression routine. That case study was based on a simple LZMA model based on SDF MoC introduced in a paper published by the author of this work (Horita et al., 2019b). Besides, the illustrative example EDS was modeled in Section 3.3.

The requirement R_4 was covered in Chapter 5, presenting the suggested formal-based design steps passable of automation as the result of AIPAA application.

6.2 Main Contributions

The main contributions of the present work are:

1. The introduction of a method to identify possible automation approaches applicable for embedded systems formal-based design, the AIPAA; and
2. The demonstration of AIPAA usage, modeling and simulating two different systems based on distinct behaviors and MoCs.

In this context, the model properties verification and low abstraction level code generation were pointed as steps that can be automated.

6.3 General Conclusions

The growing number of cyber-physical systems (CPS) and embedded systems application areas and their evolving complexity increases the difficulty of designing robust and trustable systems in a scalable manner. In this context, formal-based methods are presented as a solution towards a correct-by-construction design.

The proposed AIPAA method presents itself as an alternative for identifying possible design steps automation.

6.4 Recommendations and Future Works

The author of this work believes that the automation of formal-based design methods is an important alternative to increase the robustness and scalability of CPS and embedded systems development.

In this sense, the author recommends that more researches focus on this matter. The possible automation approaches identified by this work AIPAA application case study, *properties verification* and *low level source code generation*, could be implemented and tested. Moreover, a possible automation approach of *framework selection* step consists on listing each analyzed aspect alternatives, aiming the framework automatic selection based on its desired characteristics.

Furthermore, systems with different behaviors, modeled using other existing MoCs, or even heterogeneous systems, could be submitted to AIPAA method application. This could lead to the identification of other steps or MoC specific properties verification that can be automated. Besides, AIPAA could be extended to the implementation domain, facilitating the identification of its possible automation approaches.

References

Acosta, A. (2008). Forsyde tutorial website. `https://hackage.haskell.org/package/ForSyDe-3.1/src/doc/www/files/tutorial/tutorial.html`.

Association, T. M. (2019). Modelica website. `https://www.modelica.org/`.

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

Bourke, T., Colaço, J.-L., Pagano, B., Pasteur, C., and Pouzet, M. (2015). A synchronous-based code generator for explicit hybrid systems languages. In *Lecture Notes in Computer Science*, pages 69–88. Springer Berlin Heidelberg.

Buttazzo, G. C. (2011). *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications: 24 (Real-Time Systems Series)*. Springer.

Castagna, G. and Gordon, A. D., editors (2017). *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*. ACM.

Claessen, K. and Hughes, J. (2000). Quickcheck: a lightweight tool for random testing of haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00), Montreal, Canada, September 18-21, 2000.*, pages 268–279.

Collin, L. (2019). Xz utils. `https://www.tukaani.org/xz/`.

Community, H. (2019). Hackage website. `http://hackage.haskell.org`.

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.

Eker, J., Janneck, J., Lee, E., Liu, J., Liu, X., Ludvig, J., Neuendorffer, S., Sachs, S., and Xiong, Y. (2003). Taming heterogeneity - the ptolemy approach. *Proceedings of the IEEE*, 91(1):127–144.

Fernández, M. (2009). *Models of Computation: An Introduction to Computability Theory*. Springer, London, UK.

ForSyDe Group, K. (2019). Forsyde hackage page. `https://forsyde.github.io/tools.html`. Accessed: 2018-04-30.

Fowler, J. and Huttom, G. (2016). Towards a theory of reach. In Serrano, M. and Hage, J., editors, *Trends in Functional Programming*, pages 22–39, Cham. Springer International Publishing.

Grabmüeller, M. and Kleeblatt, D. (2007). Harpy. In *Proceedings of the ACM SIGPLAN workshop on Haskell workshop - Haskell 07*. ACM Press.

Haskell (2018). The haskell purely functional programming language home page. `https://www.haskell.org`.

Herrera, F. and Villar, E. (2007). A framework for heterogeneous specification and design of electronic embedded systems in SystemC. *ACM Transactions on Design Automation of Electronic Systems*, 12(3):22–es.

Horita, A. Y., Bonna, R., and Loubach, D. S. (2019a). Analysis and comparison of frameworks supporting formal system development based on models of computation. *Springer - Advances in Intelligent Systems and Computing*, (800).

Horita, A. Y., Bonna, R., and Loubach, D. S. (2019b). Lempel-ziv-markov chain algorithm modeling using models of computation and forsyde. *Aerospace technology Congress 2019 (FT2019)*.

Horváth, Z., Plasmeijer, R., and Zsók, V., editors (2010). *Central European Functional Programming School - Third Summer School, CEFP 2009, Budapest, Hungary, May 21-23, 2009 and Komárno, Slovakia, May 25-30, 2009, Revised Selected Lectures*, volume 6299 of *Lecture Notes in Computer Science*. Springer.

Hudak, P., Hughes, J., Peyton Jones, S., and Wadler, P. (2007). A history of haskell: Being lazy with class. In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*, HOPL III, pages 12–1–12–55, New York, NY, USA. ACM.

Jantsch, A. (2003). *Modeling Embedded Systems and SoC's*. Morgan Kaufmann, San Francisco, USA.

Jantsch, A. (2005). Models of embedded computation. Embedded Systems Handbook, chapter Models of Embedded Computation. CRC Press.

Jantsch, A. and Sander, I. (2005). Models of computation and languages for embedded system design. *IEE Proceedings - Computers and Digital Techniques*, 152(2):114.

Krizan, J., Ertl, L., Bradac, M., Jasansky, M., and Andreev, A. (2014). Automatic code generation from matlab/simulink for critical applications. In *2014 IEEE 27th Canadian Conference on Electrical and Computer Engineering (CCECE)*, pages 1–6.

Leavline, E. J. (2013). Hardware implementation of lzma data compression algorithm. In *International Journal of Applied Information Systems (IJAIS)*.

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

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

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. (2010). CPS foundations. In *Design Automation Conference*. ACM Press.

Lee, E. A. and Tripakis, S. (2010). Modal models in ptolemy. In *3rd International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools; Oslo; Norway; October 3*, number 047, pages 11–21. Linköping University Electronic Press.

Li, Q. and Yao, C. (2003). *Real-Time Concepts for Embedded Systems*. CRC Press, Inc., Boca Raton, FL, USA, 1st edition.

Li, Z., Park, H., Malik, A., Wang, K. I.-K., Salcic, Z., Kuzmin, B., Glaß, M., and Teich, J. (2017). Using design space exploration for finding schedules with guaranteed reaction times of synchronous programs on multi-core architecture. *Journal of Systems Architecture*, 74:30–45.

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.

Loubach, D. S., Nóbrega, E. G. O., Sander, I., Söderquist, I., and Saotome, O. (2016). Towards runtime adaptivity by using models of computation for real-time embedded systems design. In *Aerospace Technology Congress*, Solna, Stockholm. FTF - Swedish Society of Aeronautics and Astronautics.

Martin, G. N. (1979). * range encoding: an algorithm for removing redundancy from a digitized message.

Mathaikutty, D., Patel, H., Shukla, S., and Jantsch, A. (2008). Ewd: A metamodeling driven customizable multi-moc system modeling framework. *ACM Trans. Des. Autom. Electron. Syst.*, 12(3):33:1–33:43.

MathWorks (2019a). Simulink documentation. https://www.mathworks.com/help/simulink/index.html.

MathWorks (2019b). Simulink embedded coder website. https://www.mathworks.com/products/embedded-coder.html.

Moraes, R. K. and Loubach, D. (2017). Functional programming paradigm application to real-time embedded systems. UNICAMP Undergraduate Work.

Naylor, M. and Runciman, C. (2007). Finding inputs that reach a target expression. In *Seventh IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2007), September 30 - October 1, 2007, Paris, France*, pages 133–142.

Open Source Modelica Consortium (OSMC) (2019). Openmodelica website. https://openmodelica.org/.

Paul, J. M. and Thomas, D. E. (2005). Chapter 15 - models of computation for systems-on-chips. In Jerraya, A. A. and Wolf, W., editors, *Multiprocessor Systems-on-Chips*, Systems on Silicon, pages 431 – 463. Morgan Kaufmann, San Francisco.

Pavlov, I. (2019). 7z format. http://www.7-zip.org/7z.html.

Pike, L., Wegmann, N., Niller, S., and Goodloe, A. (2013). Copilot: Monitoring embedded systems. *Innovations in Systems and Software Engineering*, 9.

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

Rash, J. L., Hinchey, M. G., Rouff, C. A., Gračanin, D., and Erickson, J. (2006). A requirements-based programming approach to developing a nasa autonomous ground control system. *Artificial Intelligence Review*, 25(4):285–297.

Runciman, C., Naylor, M., and Lindblad, F. (2008). Smallcheck and lazy smallcheck: Automatic exhaustive testing for small values. *SIGPLAN Not.*, 44(2):37–48.

Salomon, D. (2007). *Data Compression: The Complete Reference*. Springer. With contributions by Giovanni Motta and David Bryant.

Sander, I. (2002). The forsyde methodology. In *Swedish System-on-Chip Conference*.

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

Sander, I. and Jantsch, A. (1999). Formal system design based on the synchrony hypothesis, functional models, and skeletons. In *Proceedings Twelfth International Conference on VLSI Design. (Cat. No.PR00013)*. IEEE.

Sander, I., Jantsch, A., and Attarzadeh-Niaki, S.-H. (2016). ForSyDe: System design using a functional language and models of computation. In *Handbook of Hardware/Software Codesign*, pages 1–42. Springer Netherlands.

Scott, M. L. (2009). *Programming Language Pragmatics, Third Edition*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 3rd edition.

Seshia, S. A., Hu, S., Li, W., and Zhu, Q. (2017). Design automation of cyber-physical systems: Challenges, advances, and opportunities. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 36(9):1421–1434.

Sinaei, S. and Fatemi, O. (2016). Novel heuristic mapping algorithms for design space exploration of multiprocessor embedded architectures. In *2016 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*, pages 801–804.

Standard Performance Evaluation Corporation (SPEC) (2019). 657.xz_s spec cpu 2017 benchmark description. `http:/www.spec.org/cpu2017/Docs/benchmarks/657.xz_s.html`.

Stuijk, S. (2018). Sdf3 website. `http://www.es.ele.tue.nl/sdf3/`.

Stuijk, S., Geilen, M., and Basten, T. (2006). Sdf3: Sdf for free. In *Sixth International Conference on Application of Concurrency to System Design (ACSD'06)*, pages 276–278.

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

Turner, D. A. (2013). Some history of functional programming languages. In Loidl, H.-W. and Peña, R., editors, *Trends in Functional Programming*, pages 1–20, Berlin, Heidelberg. Springer Berlin Heidelberg.

University of California, B.-o. (2018). Icyphy home page. `https://ptolemy.berkeley.edu/projects/icyphy/`. [Online; Stand 19. Dezember 2012].

University of California, Berkeley-online (2018). Ptolemyii download page. `http://ptolemy.berkeley.edu/ptolemyII/ptII11.0/index.htm`.

Zhao, X. and Li, B. (2017). Implementation of the lzma compression algorithm on fpga. In *2017 International Conference on Electron Devices and Solid-State Circuits (EDSSC)*, pages 1–2.

Ziv, J. and Lempel, A. (1977). A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343.