A Model-based Approach To Sychronous Elastic Systems

Mohamed Ammar Ben Khadra

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science

in the

European Masters in Embedded Computing Systems
Department of Electrical and Computer Engineering
University of Kaiserslautern

23rd October 2013

Supervisors:

Prof. Dr. Klaus Schneider

Prof. Dr. Wolfgang Kunz

M.Sc. Yu Bai

EIGENSTÄNDIGKEITSERKLÄRUNG

Hiermit versichere ich, dass ich die von mir vorgelegte Arbeit mit dem Thema 'A Model-based Approach To Sychronous Elastic Systems' selbststndig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollstndig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Kaiserslautern, den 20. Oktober, 2013.

Mohamed Ammar Ben Khadra

ABSTRACT

Synchronous design is currently by far the mainstream design paradigm for digital circuit. However, the move to modern nano-meter technologies has brought unprecedented delay variability issues. That makes clock distribution only possible at a very high cost in terms of power and area. Elastic circuits a.k.a. latency-insensitive circuits is an emerging method for tackling delay-variability while avoiding the technology disruption and design issues of asynchronous design. To this end, we survey and classify in this work key approaches to elasticity discussed in the literature and focus on synchronous elastic systems a.k.a latency-insensitive systems. The discussion motivates adopting model-based design in our considered approach. Our model-based design approach starts by specifying a synchronous system using the synchronous language Quartz. Then, the specified system is *elasticized*. In this process, the system is partitioned to components. Components are inter-connected to form a network of actors. Our goal in this work is to *synthesize* the resulting network to a corresponding SysteMoC actor network. SysteMoC is a SystemC based library for actor-oriented modeling. We have developed a library that implements the synthesis functionality. Our library supports the majority of Quartz features including all of its datatypes and most operators. We are also able of generating assertions that makes sure that the original synchronous specifications are not violated e.g. out-of-bound array accesses and write-conflicts. A tool `aif2sysmoc` has been developed based on our library. Given a Quartz system with its input stimuli, `aif2sysmoc` generates the complete SysteMoC simulation code of the given system. This thesis work builds the basis for future research in the area of system partitioning for systems modeled in Quartz or synchronous languages in general.

ACKNOWLEDGMENTS

This thesis has been carried out in the Embedded Systems group, Department of Computer Science, Technical University of Kaiserslautern. I'd like here to express my thanks and gratitude to the people who helped me toward the completion of the thesis. Firstly, I would like to thank Prof. Klaus Schneider, chair of the group, for giving me the opportunity to work on this topic and opening my eyes to the exciting world of model-based design. I'd like also to thank Yu Bai, PHD student at the group, for his help during this long journey. Additionally, my special thanks to Joachim Falk, lead developer of SysteMoC in the HW/SW codesign group at Erlangen-Nuremberg University, for his tips on introducing clock synchronization in SysteMoC. He also provided an unofficial version of the library that supports clock synchronization. Finally, since I'm officially enrolled as a student in the department of Electrical and Computer Engineering, I've to be co-supervised by a professor from my department. Prof. Wolfgang Kunz, chair of Electronic Design Automation group, accepted to be my department supervisor and I'd like to thank him very much for that.

Contents

List of Figures

To my parents…

Chapter 1

Introduction

Thesis organization

In this chapter 1, we will first take a look at the challenges to traditional synchronous design of digital circuits. Then, we survey various approaches to address those challenges. Later, elastic systems are discussed. The chapter concludes with a classification of elastic systems that makes the case for adopting model-based design in this field. Necessary background theory is given in chapter 2. The chapter starts by discussing model-based design and motivating its adoption. The chapter also discusses modeling using the synchronous language Quartz, and concludes with a discussion of actor-oriented modeling using SysteMoC. Implementation details of our synthesis library are discussed in chapter 3. Chapter 4 gives an inside look into a suggested target model of elastic hardware modules. It provides another perspective to the traditional black-box wrapping approach to elasticity. Later, limitations of our approach are discussed. The chapter concludes with experimental results.

1.1 Motivation

Nowadays, the majority of digital systems design is based on the synchronous paradigm. Synchronous circuits can be simply defined as circuits where the sequencing of events depend on the availability of a global periodic timing signal called a *clock*. The clock in synchronous circuits determine when data is valid to be stored and when computation sequences can start. Basically, the clock abstracts away logical timing and sequencing from the actual computation which eases the design process. That simplicity made synchronous design the mainstream digital design paradigm since early 1960s [1]. However, designing modern complex SoCs has showed that the simple synchronous assumption of a global and synchronized clock can no longer be retained due to the challenges of clock distribution.

To appreciate clock distribution challenges we need to discuss the special characteristics of clock signals. Clock signals are typically loaded with the greatest fanout, travel over the longest distances, and operate at the highest speeds of any signal within the system.

Since clock signals provide temporal reference for sequencing, the clock waveforms must be particularly clean and sharp. Additionally, the clock needs to arrive with minimum *skew* between elements and minimum *jitter* to the same element. Furthermore, these clock signals are particularly affected by technology scaling where long global interconnect lines become much more highly resistive as line dimensions are decreased. Finally, at current high clock frequencies and chip size no signal can cross the chip in one clock cycle time due to physical limits [2].

Clock signals are distributed by means of clock distribution network. This distribution network is generally a complex hierarchical structure that consumes substantial power and area of a typical SoC. A thorough account for clock distribution techniques is provided by Friedman in [3]. From a system performance perspective, clock skew and jitter are key factor in determining timing constraints. That gives rise to the *timing closure* problem. Clock distribution is a major issue not only for the digital system itself, but also for the whole digital design flow. Meeting timing closure often requires multiple time consuming iterations of synthesis and place & route. Additionally, designers may even modify the original design if timing constraints are still not met.

In addition to the increased cost and longer design flow, the synchronous paradigm has some intrinsic inefficiencies. Firstly, the whole chip operates at a clock frequency that accommodates its slowest component. Additionally, the frequency is chosen based on a worse case timing analysis of clock skew, jitter and manufacturing process variability. The migration to modern nano-meter processes exacerbates those issues as (1) the delay is dominated by interconnect delay rather than gate delay which means that the actual performance is more difficult to predict before actual layout and (2) the gap is widening between the average case and worst case delay scenarios creating a wider inefficiency. In Matzke's *Computer* magazine article [2], it has been argued that the interconnect will be the main challenge to Moore's law since wire delay is getting slower relative to the clock. That makes proper place & route of a chip with more than billion transistors a big challenge.

As market demand for SoC based products grows, the effective use of existing design components in form of *Intellectual Property* becomes critical to reduce time to market and maintain quality [4]. These cores can be developed internally or acquired from third parties, which implies that they need to be modular and flexible to be used across many designs. That gives rise to a modern design flow where legacy cores can be easily replaced with new ones. This flexibility of modular integration is challenging to achieve in a pure synchronous design. It's common for IP cores in a complex SoC to operate in different clock *timing domains* since it's difficult and power consuming to distribute a single clock signal across chip. Different timing domains are interfaced using synchronizers. Synchronizers are known to be imperfect in countering metastability [5]. Therefore, the focus is more on calculating the failure probability to insure an acceptable Mean Time Between Failures in the designed systems [6].

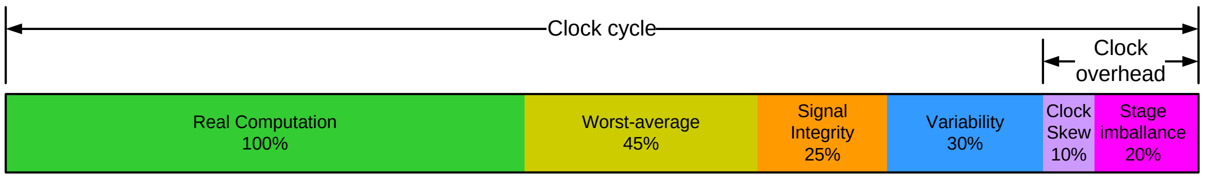


FIGURE 1.1: Worst-cast timing affects

We will continue motivating our approach by providing a look at the big picture, identifying emerging trends and challenges to the pure synchronous design paradigm. We will discuss next delay variations which is the key factor to the widening gap between average and worst-case performance. Later, we discuss trends and challenges associated wire scaling and on-chip interconnect.

1.2 Challenges to synchronous paradigm

1.2.1 Delay variations

Meeting timing-closure through careful circuit layout is a critical part of synchronous design. To this end, statistical [7] and probabilistic methods [8] for timing analysis has provided improved predictability. However, delay in modern nano-meter processes is proving increasingly difficult to predict. Unpredictability is handled by pessimistically increasing timing margins to accommodate worst-case delay. Actually, timing margins have reached 100% of actual computation time as consequence of many delay variation factors [9] as depicted in Figure 1.1. That means that maximum frequency is restricted to about double the actual predicted frequency which is obviously inefficient. There are many factors that can cause delay variations in different ways. A summery of those factors is provided in the following based on [10]:

- **Fabrication process parameter variations**. Some device parameters can vary due to fabrication process variations. Process variations can happen across dies or even within the same die. With the advance of nano-meter technologies, process variations have been increasing and the prediction has become difficult. In their work, Bowman et al. demonstrated that up to 30% of performance can be lost due to process variations alone in scaled technologies [11].

- **Voltage changes due to Dynamic Voltage Scaling**. With the advance of nano-meter technologies, the value of supply voltage has become low. Generally, switching speed becomes faster at higher voltages and slower at lower voltages. Dynamic Voltage Scaling (DVS) technique is sometimes used in order to reduce power consumption. Delay variations due to DVS can be predicted because the value of supply voltage is changed intentionally.

- **Temperature variations**. Since electric current flows during operation on VLSI circuits, heat is produced when current flows through resistances and the temperature of a chip increases. Temperature can be measured using an embedded temperature sensor. Thus, if an appropriate delay line for each temperature is selected, the performance overhead due to temperature variations can be small.

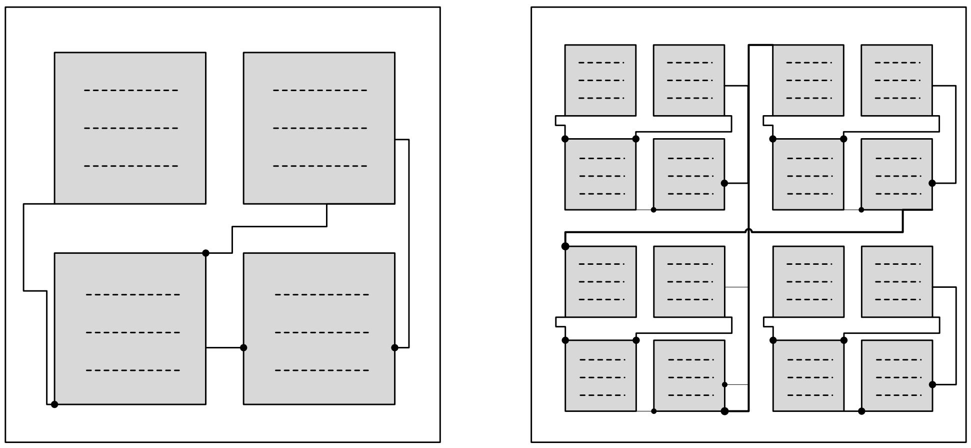
- **Crosstalk noise**. When capacitance becomes large between adjacent wires, the signal transition speed on the inside wire changes. For adjacent wires, when two rising or falling transitions occur in the same direction, the signal transition speed becomes faster. On the contrary, when two transitions occur in the opposite direction, the signal transition speed becomes slower. With the advance of deep sub-micron technologies, the crosstalk has become a serious problem because the capacitance between wires have been increasing.

- **Power supply noise and IR-drop**. Dynamic power supply noise causes fluctuations in the voltage differences between power supply and ground rails within high-density digital integrated circuits . IR-drop is a signal integrity effect caused by wire resistance and current drawn from the power supply and ground rails. It is very difficult to estimate the delay variations caused by these factors because they depend on dynamic signal transitions.

It is worth noting that the worse-case timing analysis is based on the longest critical path taken by any input. This path can be rarely taken by the ”ordinary” input. For example, the critical path of a ripple-carry adder is determined by the carry chain. That path is taken only in rare cases for typical applications. Variable latency techniques such as telescopic units [12, 13] or more recently function speculation [14] can be used in order to set the clock frequency based on the latency needed by the average critical path. Some added circuitry can detect input patterns that require a longer critical path, and adapt accordingly to provide the output after more than one clock cycle.

1.2.2 Wires and interconnect

Before moving to VLSI era, gates and wires scaled roughly linearly to a new technology. Discrepancies started to appear later with wires resisting such a trend. Basically, wires affect a circuit in three ways (1) wire capacitance adds load to driving gates, (2) wire resistance, capacitance, and inductance all add signal delay and (3) inductive and capacitive coupling between wires adds signal noise (and maybe delay variation). Wire resistance grows under scaling, since the width and height both scale down, although the height does so more slowly. Wire capacitance and inductance decreases very slowly with a scaled down technology. These scaling trends result in wires increasingly dominating circuit delay compared to gates in circuits [15].



(a) Before technology scaling (b) After technology scaling

FIGURE 1.2: Local wires get shorter while global wires get longer

Another aspect of technology scaling is related to wire length. At first glance, one might think that with reduced distance between gates, wires should go shorter too. That is not exactly true for all wires. Basically, one can observe a hierarchical paradigm where *local* wires shrink with scaling while *global* wires keep their length if not go longer since more smaller modules are packed into the same chip as depicted in Figure 1.2. Longer wires are often broken into multiple segments separated by buffers to repower the signal and minimize propagation latency. For improved throughput, the buffers can be replaced by latches [16]. The size of the considered modules is between 50K to 100K gates according to [17].

During synthesis, the capacitances of the global wires are generally unknown, and wire-load models are typically used as estimators. The accuracy of such estimations is generally acceptable for short wires, but increasingly unacceptable as the wire delays reach levels where they constitute a significant portion of the critical path delay. The dominance of wire delay and the fact that wire delay is becoming difficult to predict before place & route, and varies greatly after final tape-out is a big challenge. That challenge motivated researchers to investigate routing packets instead of wires and bring the mature knowledge of computer networking in building Networks on Chip [18, 19].

Network-On-chip (NoC) approach has emerged as a promising alternative to classical bus-based communication architectures. Aside from better predictability and lower power consumption, the NoC approach offers greater scalability compared to other on-chip communication solutions [20]. As depicted in Figure 1.3, modern SoC architectures consist of heterogeneous IP cores such as CPU, video processors, embedded memory blocks, etc. In NoC architectures, each such processing element (PE) is attached to a local router. Routers receive data from PE in the form of *packets* using a network interface (NI). Then, the packet is stored at the input channels, and the router forwards it to its destination.

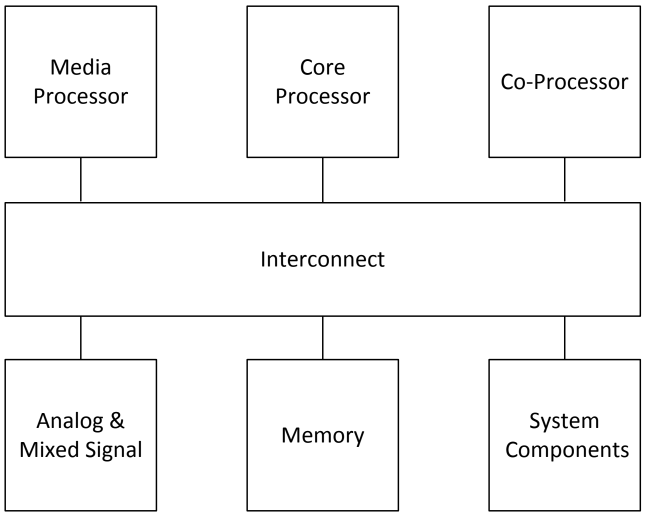


FIGURE 1.3: A model of a system-on-chip design

1.3 Asynchronous design paradigm

1.3.1 Overview

The issues discussed so far concerning the synchronous paradigm motivates having a look at the other competing paradigm, namely, asynchronous design. The fully asynchronous design methodology eliminates the need to a global clock. Circuits rely instead on local hand-shake protocols with neighbors to sequence computations. The hand-shake protocol is a simple *request* (data valid) and *acknowledgment* (data received). In the synchronous paradigm, the mechanisms that provide system timing are completely separated from the system behavior. In contrast, the asynchronous paradigm incorporates system timing into the system behavior model which makes it more difficult to design [21]. The asynchronous paradigm holds many promised advantages over the synchronous one. We list those advantages here based on [22, 23]:

- **Elimination of clock skew and jitter problems**. Basically, there is no need to distribute a globally synchronized signal anymore. The time consuming stage of iterative clock tree re-distribution and retiming [24] can be removed from design flows.

- **Average case performance**. The performance of the design is not determined based on the worst case timing analysis of its slowest circuit. Asynchronous design allows for an average case performance where each circuit signal its completion to the next circuit as soon as it has finished the current computation.

- **Adaptability to process and environmental variation**. As discussed before in 1.2.1. The delay of digital circuit can vary significantly across wafer, different production runs, supply voltages and operating conditions. Asynchronous circuits are adaptive as they can simply speedup and slow down as necessary.

- **Component modularity and reuse**. Asynchronous components can be simply interfaced and flexibly re-used without the complex multi clock domain synchronization issues of synchronous circuits.

- **Lower systems power requirement**. Asynchronous circuits reduce power consumption by not requiring a clock distribution tree all with its clock drivers and buffers. They also do not require the overhead associated with clock-gating.

- **Reduced noise**. In synchronous designs, activity is tied to a defined clock frequency. The result is that the energy is concentrated in a narrow spectral band which creates substantial electrical noise. Activity in asynchronous designs are uncorrelated, resulting in a more distributed noise spectrum.

Asynchronous design is not a newly established concept. It dates back to the 1950s with the work of Muller et al [25] and Huffman [26]. The 1952 ILLIAC and 1962 ILLIAC II computers built at the University of Illinois have contained both synchronous and asynchronous parts [27]. The modular properties of asynchronous design have been studied by the Macromodule project of the University of Washington, St. Luis in late 60s [28]. However, the wide interest in asynchronous design didn't truly start until the 1980s inspired by the chapter of Seitz in the Mead and Conway book [29]. The evolution of interest in asynchronous design in terms of publication per year is depicted in Figure 1.4 based on the asynchronous bibliography [30].

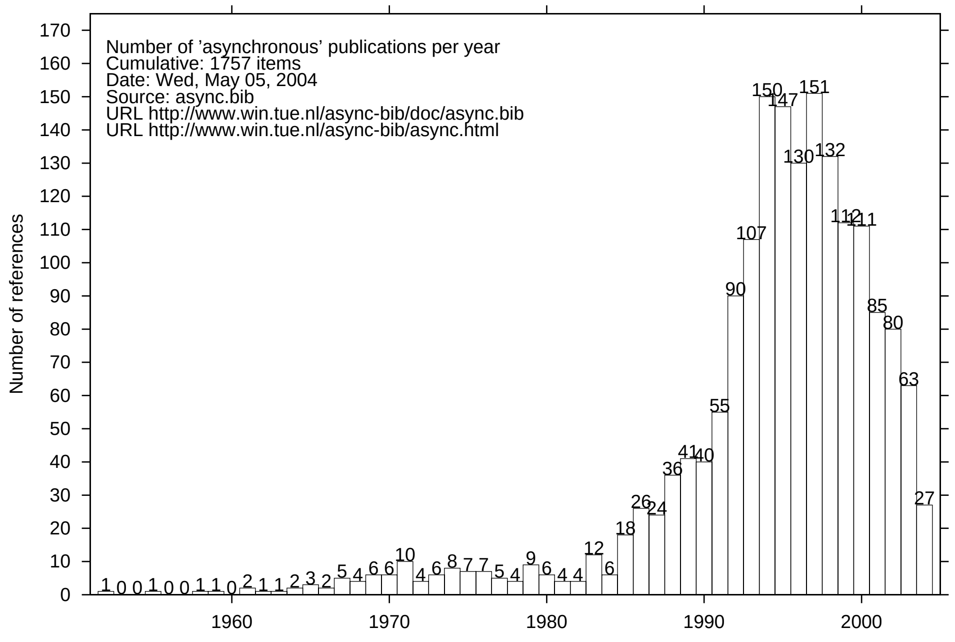


FIGURE 1.4: Publication evolution in asynchronous circuits

1.3.2 Delay models

Asynchronous approaches do not use a clock for sequencing. However, delay models are need to properly implement sequencing. The following are the most popular delay-models (assumptions).

- **Delay-insensitive** (DI). Circuits impose no timing assumptions, allowing arbitrary gate and wire delays except that the delays are finite and positive [29]. Unfortunately, this class is limited and impractical. Actually, it has been proven by Brzozowski et al [31] that DI circuits are not realizable using simple gates only.
- **Speed-independent** (SI). Circuits assume that the wire delays are negligible compared to gate delays. Imposing such a restriction on all wire forks is becoming increasingly impractical for large designs and with shrinking feature size where the share of wire delays is growing.
- **Quasi-delay-insensitive** (QDI) circuits use assumptions similar to that of SI, but partition wires into critical and non-critical. No assumptions are made about non-critical wires while in critical wires the skew between different branches is assumed to be smaller than the minimum gate delay. Critical wire forks are called isochronic forks.
- **Matched delay** (MD) circuits use delay lines to match the delay of a combinational logic island. The delay of the delay line is assumed to exceed the maximum delay of the critical path. Matched delays are implemented using the same gate library as the rest of the data path and they are subject to the same operating conditions (temperature, voltage). This results in consistent tracking of data path delays by matched delays and allows reducing design margins with respect to synchronous design

Assumptions in asynchronous circuits are not only needed for internal gates and wires but also on the environment. In *fundamental-mode* it is assumed that all circuits start from a stable state. Upon a change in a single input value the environment shall wait until the circuit stabilizes again before changing another input value. The fundamental-mode can be extended to a *burst-mode* if multiple inputs are allowed to change at the same time. The *input-output* mode is the most generic one and it allows for the input to change before the circuit stabilizes.

1.3.3 Design issues

Signaling protocols. To communicate the valid and acknowledgment signals between to adjacent asynchronous block two protocols exist (1) the 2-cycle protocol where signal detection is based on a transition and (2) 4-cycle protocol where a signal detection is based on the level. There is a misconception that the 2-cycle protocol is more efficient power-wise. Sutherland's micopipelines turing award lecture [32] promoted the 2-cycle version as the logical way of doing things. However, the 2-cycle interface has higher implementation complexity compared to the 4-cycle which may increase power consumption as a consequence [1]. The ARM1 (2-cycle) and ARM2 (4-cycle) implemented at the university of Manchester demonstrate that. For joining the requests of two or more blocks, Muller's *C-element* can be used [25]. The C-element is a fundamental communication element in asynchronous circuits. It is rare that large asynchronous circuits can be built without C-elements. However, the liberal use of them can lead to reduced circuit performance [1].

Enconding data. To communicate data between two blocks we can use the bundled data protocol. To transmit n bits, *n+2* data wires are needed. The two additional wires are used for *req/ack* signaling. This protocol is highly efficient in terms of required wires, however it is based on the bundled data assumption. It assumes that the data and control signals always arrive either at the same time or the control data arrives later. Without this assumption incorrect data maybe sampled by the receiver block. Delay elements may be deliberately added to the control signals to insure such an assumption. The common alternative to bundled data protocol is the dual-rail encoding. To transmit n bits *3*n* wires are required in total. For each bit 2 wires are needed for data and one additional wire of acknowledgment. The mainstream encoding of data on wires is 00 (idle), 10 (valid 0), 01 (valid 1) and 11(considered illegal). Obviously, this protocol has significant wiring overhead however it is more robust than bundled data. A more efficient version can be built based on the bundled data assumption. That is, by associating a single acknowledgment wire with *2*n* data wires.

Completion detection. One of the inevitable overheads in asynchronous circuits is the added circuitry needed to detect computation completion. The completion detection controls, directly or indirectly, the acknowledgment signal of a circuit. There are many methods for completion detection and none of them is universally satisfactory [1]. Timing analysis can be used to estimate how many cycles a given circuit needs to complete and associated *internal* clock generator can generate timing ticks. However, this technique doesn't lend it self to high performance designs since there is a significant time overhead in starting and the clock generator and waiting for it to stabilize before processing input. Another approach is based on the delay model of the circuit together with timing analysis. Here a delay element, such as inverter chain, is added to the circuit. The circuit is assumed to have completed its processing by the time the signal propagates through the delay element.

Countering hazards. Treatment of hazards is a fundamental issue in asynchronous circuits. In synchronous circuits hazards during the clock cycle can be simply ignored, since the final value of computation is sampled at a clock tick. In contrast,any glitch in an asynchronous circuit maybe treated as a real value change and therefore may cause the system to malfunction. Extra hardware circuitry is needed to suppress hazards and implement completion detection. Depending on the delay-model, the amount of hardware circuitry can be

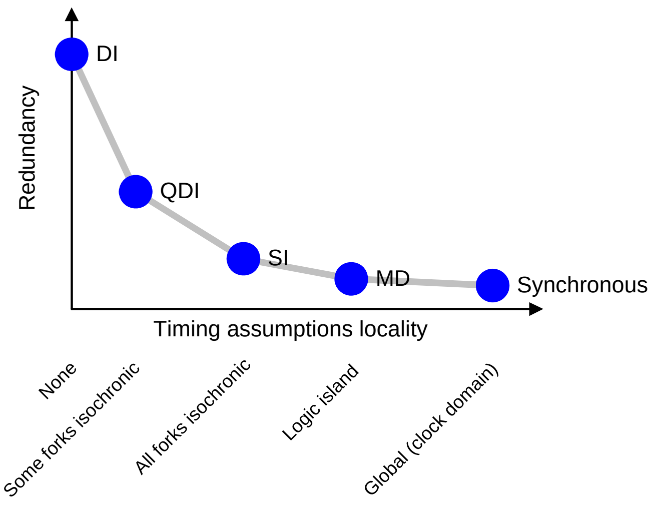


FIGURE 1.5: Relation between delay-models and redundancy overhead

substantial. Figure 1.5 provides intuition on how different delay-models affect asynchronous circuit area [33].

Despite all of the appealing features of asynchronous design, the majority of the design done today is still synchronous. That can be contributed to multiple factors including (1) handling the issues of hazards and completion detection can add some significant area overhead to a digital circuit, (2) difficulties in synthesis and optimization of hazard-free circuits, (3) lack of reliable CAD tools for asynchronous design especially, (4) significant investment required for retraining qualified personnel especially, (5) lack of asynchronous IPs and finally (6) current asynchronous designs didn't demonstrate a radical advantage over synchronous ones to the extent that can justify the investment and finally (7) usage of special purpose design languages such as Balsa [34].

1.4 Hybrid paradigms

1.4.1 Globally Asynchronous Locally Synchronous systems

It's clear by now that both synchronous and asynchronous paradigms can't provide alone the best design flow needed in complex SoC designs. It's logical to think how the best of both paradigms can be combined. Combining both paradigms in Globally Asynchronous Locally Synchronous (GALS) was systemically discussed first by Chapiro [35]. GALS are a natural evolutionary step provided that a synchronized clock can't be maintained at system level, whereas a synchronous clock is still feasible at module level. This means that existing IPs and synchronous design flow can still be used for designing modules synchronously. Then, those modules can be connected asynchronously in a GALS system.

The GALS group at ETH Zurich has a notable experience with GALS design [36, 37]. Their experience has been summarized in [38]. In their GALS design a wrapper for synchronous modules has been designed that included async/sync input and output port controllers. The wrapper included high precision programmable clock generator which means that the actual clock frequency for each module can be set after final tap-out. Leaving frequency to be set after tape-out tackles manufacturing process variability in a flexible manner. Proper automatic partitioning of a digital system into interconnected GALS subsystems is considered as major challenge for GALS design [38]. CAD tools should support early design space exploration of different GALS partitions. Another key GALS issue is synchronous/asynchronous domain interfacing. We have already mentioned how synchronizers are not perfect. Nonetheless, the area of synchronous/asynchronous interfacing has witnessed significant advances [39, 40].

We think that the synchronous and asynchronous paradigms should not be thought of as two different worlds to be interfaced. On the contrary, we should think of them as a unified set of techniques and circuit design strategies that can be mixed at various hierarchical levels. For example, the asynchronous FIFO protocol Gasp [41] used for design pipelining depends on timing analysis to control the reset phase of the handshake protocol. Another example is the telescopic units [12] a variable-latency circuit techniques used for synchronous circuits to give average instead of worst case performance. The Algebraic Decision Diagrams (ADD) used for telescopic units for input analysis can be used for completion detection in asynchronous circuits. One noteworthy technique to manipulate synchronous designs, and thus producing a kind of hybrid design, is mesochronous clock. Mesochrony means that the clock iteself is delayed to match the delay of data. Mesochonous clocks have been suggested in pipelining [42].

1.4.2 Desynchronization

In desynchronization, the traditional synchronous design flow including synthesis is followed. Then, the synthesized net-list is converted automatically to an equivalent pure asynchronous design where clocks are removed and sequencing is done by local hand-shaking. Desychronization allows the industry to fully utilize their existing investments in synchronous design. Also, it provides pure asynchronous design, to a certain extent, with the established methodology and tools that it lacks. In GALS, the synchronous and asynchronous paradigms are mixed in the architecture itself. Whereas in desynchronization both are mixed in the design flow. That is, the designer starts with a synchronous design flow and then ends with an asynchronous design.

Several (similar) techniques have been proposed for desynchronization [43–45]. Worth noting also is the desynchronization flow provided by Thesus Logic (now Wave Semiconductor) based on Null Convention Logic [46]. Among the desynchronization techniques, the work of Cortadella et al. [47, 48] was notable since it has been successfully implemented on a low power 8 bit Atmel AVR processor [49]. In that work, the design flow started form a VHDL core of the processor freely available on OpenCores.org. Conventional EDA tools of synchronous design were used. The desynchrnonized processor has shown lower energy consumption, EMI and virtually zero-wakeup time instead of about 2 ms.

It's worth noting that this approach depends on delay elements for completion detection and thus the synthesis tool has to do timing analysis for the combinational logic block and insert proper number of delay elements. A matched delay element is sensitive to process variations and thus delay elements has to be provided with sufficient margins. An alternative option to completion detection, at a higher area cost, is dual-rail encoding. Other than the required timing analysis, desynchonization is critisized in that it introduces asynchrony late in the design flow which makes it hard to take full advantage of asynchronous design.

1.5 Elastic systems

1.5.1 Overview

System elasticization [50] is a promising technique(s) for tackling challenges of synchronous design enabling it to better tolerate delay variability. It takes advantage of the orthogonality of computation and communication concerns and tries to handle them somehow separately. Given a synchronous system, the process of elasticization *partitions* the system into a set of communicating components. Then, components are *wrapped* into shells to enable them to communicate using a latency-insensitive protocol instead of plain wires. Partitioning can be inferred from existing design modules or it can be based on other criteria. Key criteria to the success of this transformation process is that it should be *correct-by-construction*. That is, the original synchronous system should be designed and verified using a synchronous design flow. Elastic transformations should preserve the functional correctness of the system.

It should be stressed that original and elasticized systems are not *cycle-equivalent*, however they must be *flow-equivalent*. Two synchronous systems are cycle-equivalent *iff* they produce the same output at the same clock cycle given the same input sequence. Flow-equivalency relaxes that definition in the sense that the condition of exact cycle matching is withdrawn. Basically, flow-equivalency should maintain output *completeness* i. e. all outputs produced in the original system are also produced in the elasticized system, and output *order* which means that they should be produced in the same order. In other words, the *total order* relation between events in synchronous systems becomes a *partial order* in elastic systems. Clock- and flow-equivalence relation is depicted in Figure 1.6. Elastic systems have not only the potential of better coping with variability but also to attain better performance since it removes cycle-equivalence constraint that should be observed by synthesis tools [51].

(a)

cycle	1	2	3	4	5	6
a	3	1	2	3	1	0
b	5	0	4	6	2	4
$a+b$	8	1	6	9	3	4

(b)

cycle	1	2	3	4	5	6	7	8	9	10
a	3		1	2		3	1		0	
b		5	0	4	6			2	4	
$a+b$		8		1	6		9	3		4

FIGURE 1.6: (a) Clock equivalent behavior, (b) Flow equivalent behavior

Since *elasticity* and *elastic systems* are newly emerging terms in the literature, a single widely accepted definition of what constitutes an elastic transformation doesn't exist. We refer to the generic definition (1.1) as our definition. Note that a GALS transformation can provide better tolerance to delay variations. However, a GALS transformation can be considered elastic *only* if it was correct-by-construction which might not be generally the case. On the other hand, desynchronization can be considered elastic under definition (1.1) since its correct-by-construction and provides delay tolerance by definition of asynchronous circuits.

Definition 1.1. Elastic transformation: *a correct-by-construction transformation to a given synchronous system that enables it to better tolerate computation and communication delay.*

The separation of computation and communication in elastic systems beers similarity to the actor model for describing parallel and distributed computation [52]. Two issues are of key concern in this separation (1) granularity of the computation components (2) type of communication. Both issues are dependent on each other, in the sense that computation components of bigger granularity makes it feasible to implement more sophisticated communication shells. Note that component communication can be synchronous or asynchronous as depicted in 1.7. We will consider in the following two different approaches to elasticity, namely, Latency Insensitive Protocols of Carloni et al. [53, 54] and elastic circuits of Cortadella et al. [50]. Note that both approaches use synchronous communication. However, they differ in the considered computation component granularity.

1.5.2 Latency Insensitive Protocols - LIP

Latency Insensitive Protocols of Carloni et al. [53, 54] is an elasticity approach devised to address timing issues associated of integrating complex SoC using multiple IP cores. It enables building functionally correct SoCs by promoting IP intensive reuse. In LIP, long inter-IP wires are segmented swith relay stations to break critical paths which brings robustness to IP against variability of data latencies by encapsulating them into communication

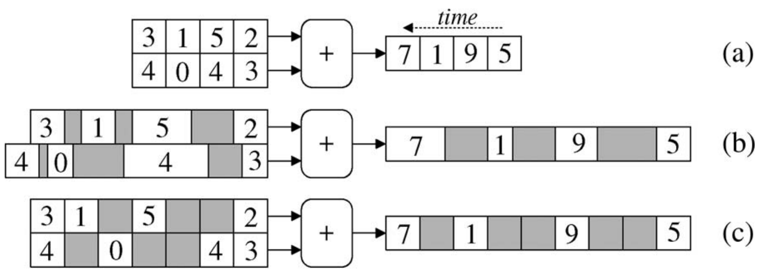


FIGURE 1.7: (a) No elasticity, (b) asynchronous elasticity, (c) synchronous elasticity

wrappers. Encapsulated IPs are called *pearls* and encapsulating wrappers are called *shells*. The approach considers modules of significant granularity and assumes a simple plain wire communication model between them. Figure 1.8 from [55] depicts an elasticized design using LIP.

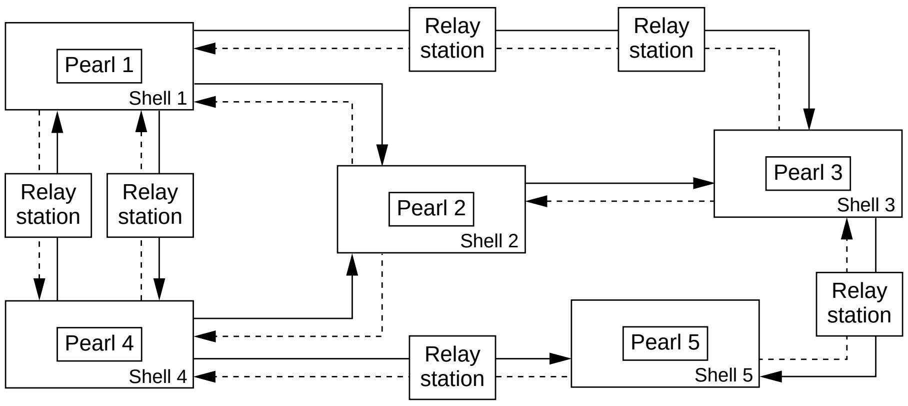


FIGURE 1.8: A LIP elasticized design

LIP simplifies timing closure by not restricting wire routing to usual single cycle setup and hold times. Place & route tools can freely segment long wires with relay stations. Relay station act as distributed FIFO implementation. Although LIP considers IP modules as black boxes, it requires a key assumption for their correct operation which is the *patience* capability. A patient IP is activated only if all its inputs are valid and all its outputs are able to store a result produced at next clock cycle. If not activated, an IP can be clock-gated to save-power. It has been proved that IPs with such a patient capability can communicate correctly using LIP [53, 54].

Waiting on all inputs before firing (activating) a reaction can be inefficient since only a subset of inputs is needed in many cases in order to fire a reaction. A simple example is the case of the multiplexer where the input selection key and the selected input are sufficient to react. Singh et al. [56] have considered improving LIP by replacing the mechanism of waiting on all inputs (combinational logic) with a FSM that that enables firing based on a subset of input depending on the state (sequential logic). In that scheme, stalling the pearl is needed only when relevant inputs (and output space) is not available. The major drawback of this scheme is that the FSM can be quit big in complex communication behaviors which leads to unacceptable area overhead in real world application.

There is no well-defined protocol on how relay-station communicate with each other or how they communicate with communicating modules. However, relay station must be capable of handling *valid* and *back-pressure* control signals for the correct operation of LIP. The valid signal informs next relay-station (or shell) that there is a valid token to be communicated. Back-pressure control signal is needed to inform previous relay-station (or shell) to stall because the FIFO is full. Hardware synthesis of relay-stations together with their control circuitry can add some significant area overhead. Casu et al. [57] have proposed to use scheduling to reduce required area. They proved that if it is possible to determine a static scheduling of all the IPs firings, then the relay-stations can be replaced by simple flip-flops and control signals can be removed. They used a shift register to drive IP firing.

1.5.3 Elastic circuits

The elastic circuits approach of Cortadella et al. [58, 59] is probably the most well developed approach to elasticity in system design. At its core is a well-defined latency insensitive communication protocol called the Synchronous ELastic Flow protocol (SELF) [59]. Like LIP, it uses synchronous communication to reduce overhead and simplify synthesis. However, it considers elastic transformations at finer granularity. Basically, elasticity is considered as another possible microacrhitectural transformation like retiming [24]. The potential of their approach in autmotatic pipling and microarchitectural transformation has been discussed in [60, 61].

Relay-station of LIP are replaced with Elastic Buffer (EB) each consists of two Elastic Half Buffers (EHB). An EHB is capable of storing one token which means that an EB can store two tokens at one time in total. The communication protocol (SELF), like in LIP, depends on a forward channel (valid) and a backward channel (stop). The sender in SELF can be in one of three states namely, **Transfer** (T) when $valid \wedge \neg stop$, **Idle** (I) when $\neg valid$, and **Retry** (R) when $valid \wedge stop$. These states are depicted in Figure 1.9. In general, EBs can have multiple input/output channels. This can be supported by using elastic *fork* and *join* control structures.

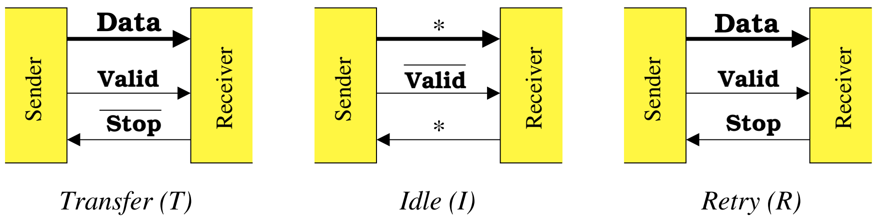


FIGURE 1.9: SELF protocol

The set of elastic transformations considered in this approach have been summarized in [51, 62]. We will outline those transformations in the following:

- **Early evaluation** [63]: an optimization technique to fire a node even when not all inputs are available. Anti-tokens should be sent on input ports not involved in current firing, otherwise spurious reactions could happen.
- **Recycling**: it's always possible to add or remove empty EB (called *bubble*) which can help in delay matching. This technique can be combined with retiming for improved performance [64].
- **Manipulating buffer capacity**: in some cases, adding capacity (more EBs) can prevent a deadlock or increase the performance of the system.
- **Sharing of functional units**: if early evaluation was implemented, some computations can be delayed or canceled. This allows for an opportunity to share the functional units implementing early evaluation.
- **Scheduling** [65]: finding some paths in the circuit where a known scheduling can be extracted allows to reduce the hardware overhead of SELF implementation e.g. the backward channel control can be eliminated.

1.6 A classification model for elastic systems

We have already discussed key synchronous approaches to elasticity. They are synchronous in the sense that their communication mechanism relays on the availability of a clock. That means that the delay between reactions is an integer number of clock cycles. Elasticity is certainly not limited to synchronous communication since computation and communication are considered separately. Additionally, communication mechanisms can't be limited to simple protocols like SELF since more sophisticated communication services, like error control, might be needed. Note that component granularity is not the only factor affecting communication style and feasibility. Physical constraints play a key role in that too. For example, applicability of bundled-data assumption can significantly reduce hardware area compared to dual-rail encoding.

To better understand the relationship between component granularity, physical assumptions, and communication options we propose the hierarchical model depicted in Figure 1.10. The inner circle considers circuits of smallest granularity (*cells*). The size of the considered component increases outwards till reaching subsystem components. Sub-systems represent components of the biggest considered size. In that figure, arrows refer to an issue or a physical constraint that affects communication style. The coloring refers to our (rough) answer on whether that issue is valid or not at a given level of the hierarchy.

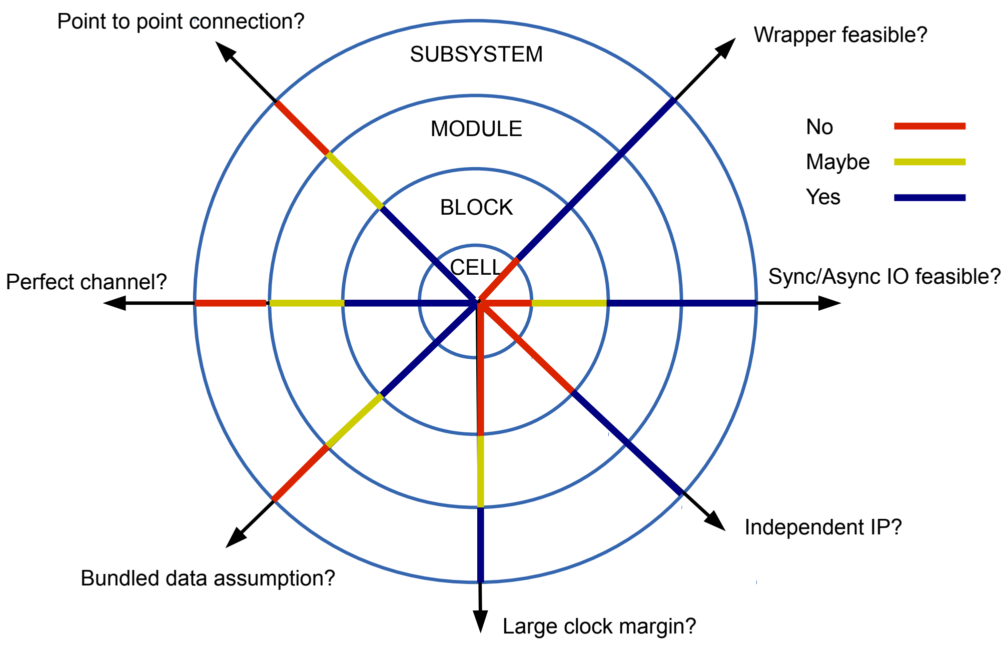


FIGURE 1.10: Hierarchical classification model

We will describe the issues considered in Figure 1.10 in a counter clock-wise manner starting from *wrapper feasibility*. Wrappers (shells) can provide various kinds of services to inner-modules (pearls). These services might include an independent clock generator, synchronous/asynchronous interface, power-gating, etc. A *synchronous/asynchronous interface* is a wrapper service. However, it has been considered separately due to the availability of fine-grained interfacing solutions like [39]. That makes it applicable at smaller scales compared to the previous issue. Next issue to consider is *IP independence*. An independent IP can have certain requirements in since it should be flexible enough to be used across different designs. Therefore, this issue has to be considered separately by the designer.

Clock margins refers to the cost of maintaining the synchronous paradigm at a given level. Synchronous paradigm can provide a big saving in terms of hardware requirement compared to asynchronous paradigm. However, the bigger the considered module the harder, more costly, is to maintain a synchronized clock signal with low delay-variability margin across it. *Bundled-data assumption* , see section 1.3, is a key assumption to maintain low control overhead in asynchronous communication. Unfortunately, it doesn't hold in communication across chip between larger modules.

It has been long the case that digital modules communicate in a *perfect channel* assumption. A perfect channel provides an in-order and error-free communication between communicating parties. However, with higher density, noise, and capacitance coupling this assumption can't be held for granted anymore, especially in NoC architectures. Designers should think about implementing proper error detection (and recovery) schemes [66]. Finally, we consider the issue of *point-to-point communication* between digital components.

Traditionally, smaller digital components use point-to-point connections for communication. On the other hand, sub-system level components (e.g. processors) use sophisticated multi-point, often standardized, interconnect fabric to communicate with other sub-systems. Bus-based interconnect like AMBA [67] is the dominating technology for sub-system level communication. However, it's being slowly replaced by more sophisticated NoC technologies. It should be expected that components smaller than sub-systems might need multi-point communication in the form of *Micro-NoC* to achieve better communication efficiency.

Having discussed the various design issues of Figure 1.10, we can give in the following an intuitive description of the various granularity levels depicted there:

- **Cell level**: a combinational logic element with well defined purpose (adders, multipliers, comparators, ... etc). In synchronous terms, elements at this level need at most one cycle to complete. It may be asynchronously micropipelined [32] for higher throughput. These components are normally found in technology libraries.
- **Block level**: a combinational or sequential component. In order delivery and error-free communication assumptions are expected to hold at this level, however wrappers might not be feasible due to their relative size. Physical assumptions (e. g. bundled-data assumption) can draw the border between this level and the next level in the hierarchy.
- **Module level**: well defined components which can be in the form of independent IP. Composable of one or more block-level components. In order delivery and error-free communication assumption might not hold. Heavier wrappers with internal clock generators might be feasible.
- **Subsystem level**: The size justifies the overhead of additional clock-generator and input/output port-controllers that can handle async/sync and different sync/sync communication. In-order delivery and error-free communication assumption are expected not to hold at this level. Sophisticated, and often standardized, communication interconnect is expected. Processors, memories, and various types of controllers fall in this category.

We discuss now hybrid paradigm and system elasticization approaches and show how they can fit into our model. GALS components for example fit easily into sub-system level. Communication is asynchronous which allows maximum elasticity. Note that communication here is not point-to-point. Desynchronization, works on cell level (and combinational block level) replacing synchronous clock sequencing with asynchronous control mechanism. Elasticity can can be considered as an extreme form of elasticization where the synchronous paradigm is completely removed. LIP approach considers independent modules and relays on synchronous communication. LIP can be placed at module-level in our model. Finally, there is the elastic circuits approach. Since it proposes fine-grained transformation, it can best placed at block-level. Note that it also proposes a synchronous communication mechanism.

This hierarchical perspective enables us to appreciate challenges of system elasticization. Firstly, there is the *partitioning challenge*. How a system can be partition into components in an optimized (or at least in a better) way, for example w.r.t. a criteria such as minimizing communication. Additionally, there is the *communication challenge* where suitable communication mechanisms should be synthesized maybe for each level separately. This challenge might not applicable to sub-system level components since their interconnect is sophisticated and usually standardized. Also, there is a *correctness challenge* that should insure that the complete elasticization approach is correct-by-construction.

Additionally, our classification model has considered hardware only. It can be generalized to consider software too. Software can be modeled as the outer-most level. Note that in addition to partitioning and communication synthesis, software needs to be *mapped* and *scheduled*. That is, each group of independent software components needs to be mapped to specific hardware sub-system component e.g. a processor. Also, a mapped group of software components need to be scheduled on the particular resource. Both mapping and scheduling can be static (at compile time), dynamic (at run-time) or a mix of both. For dynamic mapping and scheduling, designers need to consider having an OS or a middleware layer.

We believe that the current synchronous design flow falls short on addressing hardware elasticity challenges. Basically, traditional HDL languages, e.g. VHDL or Verilog, were originally designed to be simulation languages. They have been later adapted, by feature restriction, to be synthesizable. Elasticity requires modeling languages that are not only amenable to synthesis, but also to analysis and formal verification. Formal verification, can reduce the huge effort currently invested in verifying and validating designs using mostly a mix of simulation and assertion-based verification.

To this end, we believe that the synchronous approach [68, 69] to system modeling can provide an appropriate starting point for a synchronous system design flow that can address both hardware and software design challenges in a unified way. The design flow in this thesis starts from the synchronous language Quartz [70]. We aim at synthesizing an elasticized synchronous system written in Quartz. Elasticizing in this context requires *partitioning* the original system to an actor network of synchronous components which should provide a flow-equivalent output to the original synchronous system. The elasticized system will be simulated using SysteMoC [71, 72], an actor-oriented modeling library based on SystemC. We will discuss our approach in detail in chapter 3. Necessary background theory is provided in the next chapter.

Chapter 2

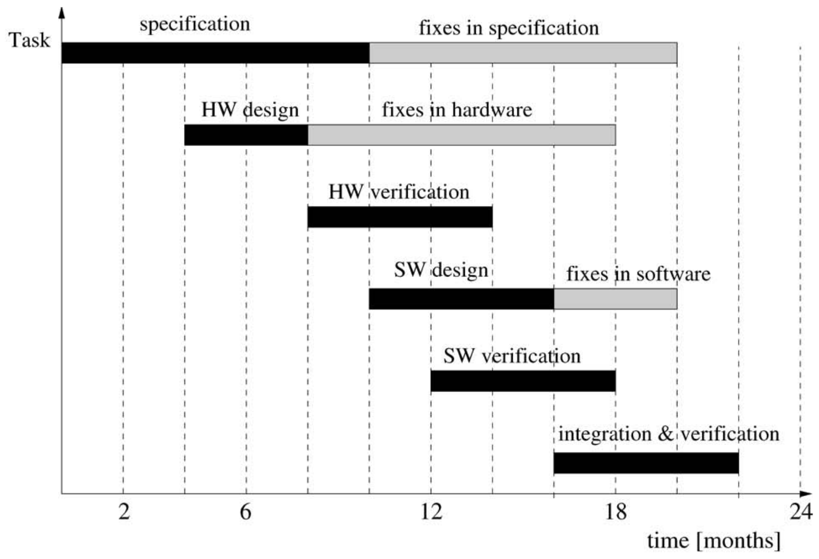
Background Theory

2.1 On model-based design

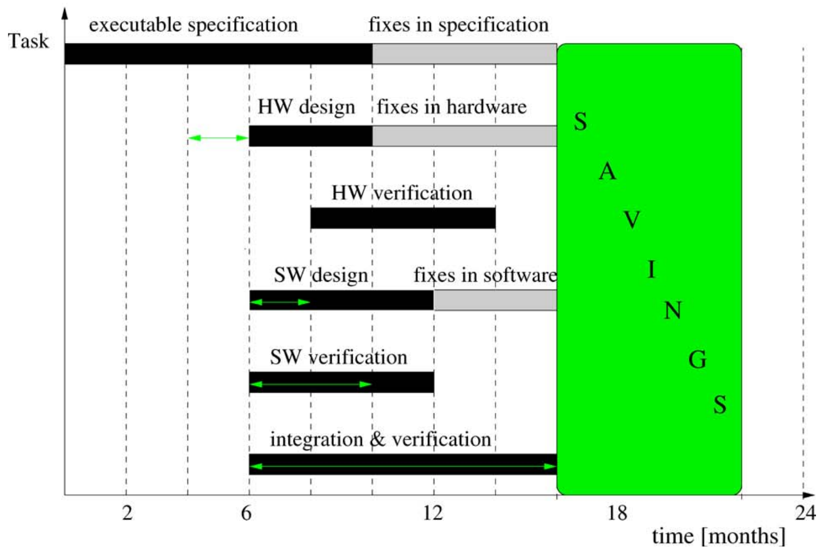
Moving design flows of increasingly complex systems to a higher level of abstraction allows many forms of verification to be performed much earlier in the design process, thus reducing time to market, and lowering costs by discovering problems earlier. Starting from an informal human readable document specification, an *executable* specification model is generated. Executable specifications have the advantage to provide a common reference for architectural exploration, hardware and software design flows, and integration and verification processes. Furthermore, they resolve ambiguities which may be present in the informal system specification.

Executable specifications of embedded systems have special emphasis on the modeling of concurrency and time. Concurrency is emphasized due to the increased distributed nature of embedded designs and the move from simple uni-processors to multi- and even many-processor designs. Additionally, modeling of time is essential to satisfy the real-time requirements that embedded systems usually have. Specification models are usually not directly executable by themselves. They need to be synthesized in order to run *natively* on a platform. That is, code generators produce source code from the model in the form of HDL for hardware and/or software code in a language like Java or C++. Platform independent models enable design space exploration by synthesis based on different architectural specifications.

The benefits of model-based design is better understood by comparing the time-to-market (TTM) of a typical embedded system project with and without employing executable specifications. The comparison is depicted in Figure 2.1, taken from [73]. The figure shows typical savings in TTM for a 24 month project involving hardware/software co-design. It's widely accepted that addressing design productivity gap seen in complex systems is possible only by moving to higher levels of abstraction and taking key design decisions earlier in the design process. To this end, the design of the executable specification language and methodology themselves has become a very active research area.



(a) Without executable specification



(b) With executable specification

FIGURE 2.1: Typical time-to-market reduction with model-based design

Increasing complexity and stringent quality requirements make the of design executable specification languages particularly challenging. The challenge is only increasing with embedded systems interfaced with physical world to control variety of sensors and actuators to in a so called Cyber Physical Systems (CPS). CPS is an emerging and increasingly important application field of embedded system. CPS are pushing the modeling design challenge to a whole new level by mixing discrete value digital systems with continuous value analog systems.

It's important for a model-based design language designer to keep in mind how his/her modeling methodology supports synthesis, analysis, and formal verification. These criteria are required in order to justify and make the adoption of this disruptive technology feasible.

Typically, developers of embedded systems should be able of specifying, refining and composing existing specification models. Then formal properties are verified against the model. After that, the model is analyzed for optimization. Finally, the final functional model is synthesized based on a given architectural model to get the final running system. Note that each step can be iterated multiple times.

In order to support analysis and thus support the synthesis of optimized code, model-based design methodologies should at least support a Models of Computation (MoC). Supporting a MoC means that computation and communication behavior should be expressed in some well-defined operations and adhere to well-defined rules. Some examples of well-known MoCs are Finite State Machines (FSM), Data-flow Process Networks (DPNs), and Communicating Sequential Processes (CSP). Normally, MoCs restrict expressiveness to provide better analyzability. For example, Static Data-Flow networks (SDF) [74] is a restricted type of DPNs where the number of tokens produced/consumed in a node firing on each link is fixed. That restriction provides unparalleled static scheduling possibility, however expressiveness limitations makes it useful in special applications only.

To this end, research in model-based design has focused on how multiple MoCs can be combined in a unified modeling methodology. The goal is to take advantage of each MoC's analyzability properties in a methodology that can be applied to model embedded systems for various application areas. The Ptolomy II project [75] is a pioneering project in the area of model-based design using heterogeneous MoCs. This chapter continues by discussing the synchronous MoC and the modeling language Quartz [70]. Later, necessary background on SysteMoC and actor-oriented modeling will be provided. Remember that Quartz is the starting point of our design flow in this thesis. Our synthesis target is a an actor model written in SysteMoC.

2.2 Synchronous modeling with Quartz

2.2.1 Overview

Reactive systems [76] is a category of computer systems that are event-driven and should continuously react to external (or even internal) stimuli. It's often the case that the reaction is also tied with a certain time bound which makes the system real-time also. There is a consensus in the literature on software and systems engineering recognizing the existence of major problem in the specification and design of complex reactive systems. Examples of reactive systems include operating systems, avionics, and missile-control.

Synchronous languages [68, 69] have emerged since early eighties to address the specification problem of reactive systems. They are based on a simple MoC hypothesis of *perfect synchrony* where the execution of the system is divided into discrete reaction steps called *macro-steps*. In each macro-step, the system reads all inputs, executes a finite number of *micro-steps*, and finally produces outputs. All micro-steps are executed in a single variable environment which is built constructively. Simplicity of the synchronous hypothesis enables easier reasoning about system behavior. Additionally, it lends itself naturally to formal verification by model checking.

Landscape of synchronous languages is crowded with many languages and derivatives, some of which are imperative like Esterel [77] and others are declarative like SCADE/Lustre [78]. Both of which have made their way to commercial applications. Lustre (and its graphical environment SCADE) is a notable success for synchronous languages. It has a certified compiler that made it easier to certify SCADE/Lustre programs for avionics. SIGNAL [79] is a synchronous language notable for relaxing perfect synchrony (single clock) by enabling polychrony (multiple clocks).

We consider in this thesis a model-based design flow that starts with the synchronous language Quartz [70]. Quartz is an imperative synchronous language derived from Esterel. It is developed in the Embedded Systems group at the University of Kaiserslautern. Quartz is the specification language of the Averest framework [1]. The framework includes a Quartz compiler among other tools required for the development and synthesis of reactive systems. Synthesis is implemented in Averest to software (C) and to hardware (Verilog).

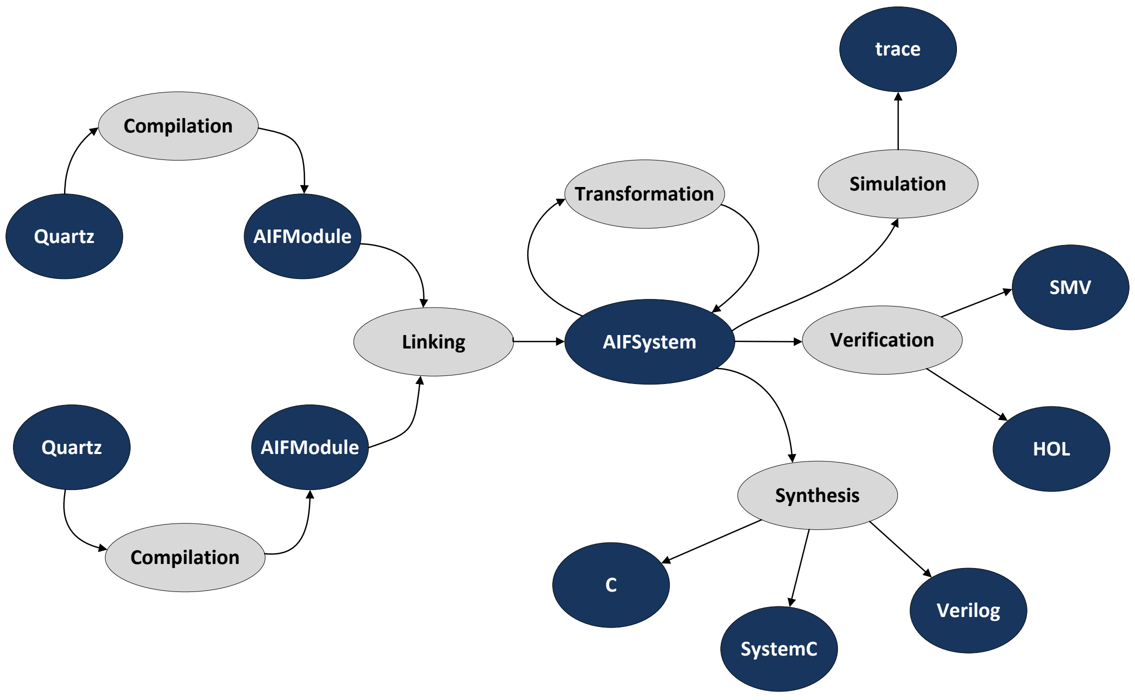


FIGURE 2.2: Design flow of Averest framework

Figure 2.2 depicts the development flow in Averest framework. A system specified in Quartz is compiled using the Quartz compiler (`qrz2aif`) to an Averest Intermediate Format (AIF) system. AIF format is a simple *flattened* format where the whole system is described

[1]Averest framework can be found at `http://www.averest.org/`

by a list of *synchronous guarded actions* (or simply guarded actions). Guarded actions are discussed in detail later. An AIF system can be *transformed* e. g. by optimizing guarded actions. Trace simulation of an AIF system is done using `aif2trc` tool. Formal verification can be done by model checking e.g. with NuSMV or by theorem proofing e.g. with Higher Order Logic (HOL).

Note that our focus in this work is on implementing a synthesis library that takes an *elasticized* AIF system as input. Elasticizing an AIF system means partitioning it to so called AIF components and linking those components in a topology. We haven't considered such partitioning transformation in this work and left that for a future work. Next, we discuss the syntax and semantics of Quartz. Later, we discuss guarded actions and the AIF system format.

2.2.2 Fundamentals of Quartz

We provide in this section a crash course on the imperative synchronous language Quartz [70]. Basically, an imperative language like C can be transformed to a synchronous language by introducing the keyword `pause` thereby distinguishing between micro-steps and macro-steps. All statement between two `pause` statements would belong to the same reaction i. e. same macro-step. In turn, each of those statements is considered a micro-step. To illustrate the idea, let's consider Quartz statements of Listing 2.1. Values of `x` and `y` in the second macro-step is 9, which is consistent with imperative languages. Note that reordering the assignments in Listing 2.2 doesn't matter and their value remains 9.

```
x=5;
w1:pause;
x=9;
y=x;
w2:pause;
```

Listing 2.1: Quartz example (1)

```
x=5;
w1:pause;
y=x;
x=9;
w2:pause;
```

Listing 2.2: Quartz example (2)

Having a single variable environment in a macro-step is the first thing people used to imperative languages may find odd. We continue with another "odd" feature which is delayed assignments. A delayed assignment sets the value of a variable in the next macro-step while evaluating the assignment's Right-Hand Side in the current variable environment. Let's consider the delayed assignment at line #4 in Listing 2.3. It assigns the value 6 to `y` in the next macro-step. Assignments in Quartz are either immediate (as seen before) or delayed (using keyword `next`). Note that both variables have been declared as `int` in Listing 2.3.

Things have been modified in Listing 2.4 where `y` has been declared as *event* variable. Variables of storage type event have the special behavior that if their value has not been explicitly set, a reaction to absence takes place that gives them the default value of their type which is 0 for integers. That is, the value of y in the second micro-step is 0 whereas it's

5 for the *memorized* variable x. Variables in Quartz can be either of storage type *event* or *memorized*. Note that not setting a variable's storage type implies that it is of storage type memorized.

```
1  int x;
2  int y;
3  x=5;
4  next(y) = x + 1;
5  w1:pause;
6  x=9;
7  w2:pause;
```

Listing 2.3: Quartz example (3)

```
1  int x;
2  event int y;
3  y = 6;
4  x = 5;
5  w1:pause;
6  next(y) = x;
7  w2:pause;
```

Listing 2.4: Quartz example (4)

Quartz programs are arranged in modules where the main module calls sub-modules in a tree like hierarchy. Each module should be stored in a separate file that has the extension `.qrz`. A module must have the same name as its file name. Related module can be arranged in a *package* in a folder hierarchy similar to Java. Listing 2.5 depicts an example Quartz module with some important concepts to discuss. Firstly, module's signature has three variables, an input variable `a`, input/output variable `b`, and an output variable `c`. All of type *bool* and storage type *event*. Basically, variable's declaration in the signature has the elements of (1) storage type (event or memorized), (2) variable type, and (3) variable flow (input only, in/out or output only).

```
01 module example05(event bool ?a, b , !c) {
02   nat{16} x;
03   {
04      next(x) = x +1;
05      if (a) c= true;
06      pause;
07   } ||
08   {
09      if (!a) b= true;
10      pause;
11    }
12 }
```

Listing 2.5: Quartz example (5)

In Listing 2.5 a local *natural* variable `x` was defined in Line #2 . Note that it was defined over a range, in this case, `x` can take a value between 0 and 15. Defining variable over ranges is required for formal verification. Note how the value of `x` can be incremented in line #4. Note also that there are two code blocks running concurrently. Synchronous parallelism is possible using operator || where both threads run in lock step.

```
module example6(event bool o){
   if (!o) o=true;
}
```

Listing 2.6: Quartz example (6)

```
module example7(event bool o){
   if (o) o=true;
}
```

Listing 2.7: Quartz example (7)

The semantics of synchronous languages like Quartz requires that all variables have a unique value in each reaction. However, this is not always possible to find due to *causality problems*. Listing 2.6 shows a module where the value of the output depend on itself. Obviously, no unique value of `o` can satisfy the required behavior. Listing 2.7 depicts a module with two different satisfying values of `o` false and true. Therefore, it is also not causally correct. It is the responsibility of the compiler to perform causality analysis to identify and reject such causally incorrect modules. We conclude this section with the list core statements of the language Quartz provided in definition 2.1.

Definition 2.1. The set of core statements of Quartz is the smallest set that satisfies the following rules, provided that S, S_1, and S_2 are also core statements of Quartz, ℓ is a location variable, x is a local or output variable, σ is a Boolean expression, and α a type:

- $x = \gamma$ and `next`$(x) = \gamma$ (immediate/delayed assignment)
- `assume` (σ) (assumption)
- `assert` (σ) (assertion)
- `nothing` (empty statement)
- ℓ `:pause` (consumption of time)
- `if` (σ) S_1 else S_2 (conditional)
- S_1 ; S_2 (sequence)
- $S_1 || S_2$ (synchronous concurrency)
- `do` S `while` (σ)(iteration)
- $\{\alpha x; S\ \}$ (local declaration)
- `during` S_1 `do` S_2 (invariant action)
- `abort` S `when` (σ) (abortion)
- `weak abort` S `when` (σ) (weak abortion)
- `immediate abort` S `when` (σ) (immediate abortion)
- `weak immediate abort` S `when` (σ) (weak immediate abortion)
- `suspend` S `when` (σ) (suspension)
- `weak suspend` S `when` (σ) (weak suspension)
- `immediate suspend` S `when` (σ) (immediate suspension)
- `weak immediate suspend` S `when` (σ) (weak immediate suspension)

2.2.3 Synchronous guarded actions

The result of compiling a Quartz program is an Averest Intermediate Format (AIF) file. Basically, an AIF file is a list of *synchronous guarded actions*. These guarded actions can be separated to control-flow guarded action (CGA) and data-flow guarded actions (DGA). Actions and guarded actions are formally defined in definitions 2.2 and 2.3 respectively. CGAs are of the form $\langle\gamma \rightarrow next(l) = true\rangle$ where l is a control flow label. DGAs are more generic in the sense that any valid lhs can be used, however they strictly do not use labels as lhs. Guarded actions are designed in the spirit of guarded commands [80] which are a well-established formalism for the description of concurrent systems. Note that it is possible to translate any synchronous program e. g. Esterel or Lustre, to a list of guarded actions.

Definition 2.2. Action: an action is represented by the tuple $(lhs, rhs, assn)$ where lhs (left-hand side) refers to the variable that is assigned a value, rhs (right-hand side) refers to the expression the should be evaluated to get the value of lhs. Finally, $assn$ (assignment type) refers to the type of the action which can be immediate (AssignNow) or delayed (AssignNxt).

Definition 2.3. Guarded action: A guarded action is represented by the tuple $(grd, lhs, rhs, assn)$ where grd (guard) is a boolean expression, and tuple $(lhs, rhs, assn)$ represents the action that is executed if the guard evaluates to **true**.

Definition 2.4. Guarded action dependencies: let $G = \langle\gamma \rightarrow x = \tau\rangle$ be a guarded action where γ is the boolean guard, x is the lhs i. e. the variable assigned a value, and τ is a general rhs expression. Let $FV(\tau)$ be the set of free variables in expression τ. We define the following:

$$rdVars(\gamma \Rightarrow x = \tau) = FV(\gamma) \cup FV(\tau)$$
$$rdVars(\gamma \Rightarrow next(x) = \tau) = FV(\gamma) \cup FV(\tau)$$
$$wrVars(\gamma \Rightarrow x = \tau) = \{x\}$$
$$wrVars(\gamma \Rightarrow next(x) = \tau) = \{x\}$$

We say that two guarded actions G_1 and G_2 have dependency iff $wrVars(G_1) = wrVars(G_2) \vee wrVars(G_1) \in rdVars(G_2) \vee wrVars(G_2) \in rdVars(G_1)$, otherwise G_1 and G_2 are considered independent.

It is essential in many situations to analyze the dependencies between different guarded actions e. g. when considering synthesis of synchronous guarded action to a language with sequential semantics e. g. C or Java. To this end, definition 2.4 contains notations needed later when we synthesize our elastic synchronous system to SysteMoC. To better understand the relation between guarded actions in an AIF system and the original Quartz program we discuss the Quartz module `test` of Listing 2.8. Result of synthesizing module `test` is the list of guarded actions shown in Listing 2.9.

Note that `pause` statements have been labeled to make tracing of behavior easier. Let's examine first CGAs and compare them to the Quartz module. For example, to reach label `w1` we should either be at the `start` label or at label `w1` itself. This behavior is encoded as a guarded action on line #3. Similarly, for label `w5` to be reached then the control should be at either `w3` or `w4` the the condition `t1 < t2` should hold in both cases. Moving to DGAs the situation doesn't change much except that assignments are either done on local variables (`t1` and `t2`) or output variables (`s` and `s2`). Input variables are only readable in Quartz.

```
01  module test(int{256} ?a, ?b, bool ?test, int{256} !s, event bool !s2) {
02      int{256} t1 , t2;
03      s2 = true;
04      loop{
05              w1: pause;
06              t1 = a;
07              t2 = b;
08      }
09      ||
10      loop{
11              w2: pause;
12              if (a < b) {
13                  w3: pause;
14                  s = 5;
15                  next(s2) = true;
16              }
17              else{
18                  w4: pause;
19                  s = a+1;
20              }
21              if (t1 < t2)
22                  w5:pause;
23      }
```

Listing 2.8: Example Quartz module `test`

Figure 2.3 depicts an Extended Finite State Machine representation of module `test`. This representation is generated from CGAs and DGAs in AIF system file. CGAs are used to generate the states and transitions. DGAs are later *attached* to their corresponding states. Note that each state is defined by a set of labels. EFSM generation is required later in our synthesis process and will be discussed in detail in section 3.3.

```
Control Guarded Actions:
  $true \rightarrow next(start) = true$
  $start \vee w1 \rightarrow next(w1) = true$
  $start \vee (w3 \wedge \neg(t1 < t2)) \vee (w4 \wedge \neg(t1 < t2)) \vee w5 \rightarrow next(w2) = true$
  $w2 \wedge (a < b) \rightarrow next(w3) = true$
  $w2 \wedge \neg(a < b) \rightarrow next(w4) = true$
  $(w3 \wedge (t1 < t2)) \vee (w4 \wedge (t1 < t2)) \rightarrow next(w5) = true$

Data Guarded Actions:
  $start \rightarrow s2 = true$
  $w1 \rightarrow t1 = a$
  $w1 \rightarrow t2 = b$
  $w3 \rightarrow s = 5$
  $w3 \rightarrow next(s2) = true$
  $w4 \rightarrow s = a + 1$
```

Listing 2.9: List of guarded actions of module `test`

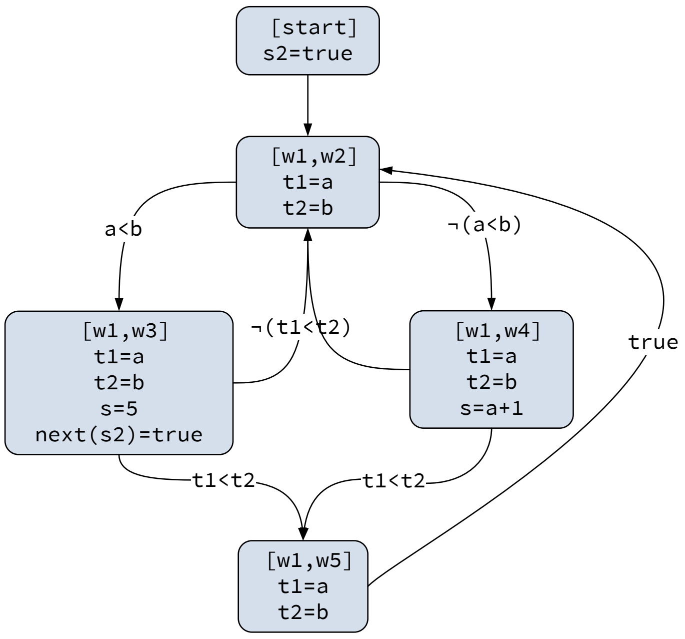


Figure 2.3: Extended FSM of module `test`

2.3 Actor-oriented modeling with SysteMoC

Actor-oriented modeling is commonly used in modern design of embedded systems [81]. In actor-oriented modeling, a system is described by concurrently executed entities called actors which communicate among each other via dedicated channels only. The model was originally formalized by Agha and others [52] for reasoning about concurrent software. The original proposal assumes that each actor has its independent thread of control and communicates using message passing over an asynchronous channel with other actors.

The original actor model is very expressive. Therefore it's usually restricted when used in the modeling of embedded systems e. g. actors do not need to communicate asynchronously. That brings back the issue of expressiveness vs. analayzability. With emergence of SystemC [82] as the de-facto standard for hardware/software modeling, interest grew among researchers in SystemC based modeling of heterogeneous MoCs. Patel et al. [83] approached the issue by modifying the SystemC kernel itself. That approach proved not to be viable in the long-run since a huge amount of effort should be spent to maintain the new kernel.

Falk et al. [71, 72, 84] of the HW/SW co-design group at Erlangen-Nuremberg University, on the other hand approached the problem by developing SysteMoC, an actor-oriented modeling library built on top of existing SystemC kernel. Multiple levels of expressiveness restrictions can be imposed on actor communications in order to have better analayzability. SysteMoC is considered as input "language" for SystemCoDesigner [85] an Electronic System Level (ESL) design tool capable of rapid design space exploration. We will discuss SysteMoC in the following first informally and then provide formal definitions of described entities.

SysteMoC is used to describe a network graph of communicating actors. Each actor has a set of input ports $\mathcal{I}$ and a set output ports $\mathcal{O}$. Ports have a supported token type e. g. double. An actor's input port should be connected to the output port of another actor that support the same token type. The connection is a FIFO that has a configurable size. Actor's internal state is defined by a set of local variables that are readable and writable only inside the actor. The behavior of the actor is defined by a Firing-FSM (FFSM). A FFSM is a finite state machine with a special kind of state transitions. FFSM's transitions have *guards* and also have firing actions associated with them. That is, whenever an actor is in a certain state, it moves to next state only when one of the transition guards is satisfied. When moving to a new state a firing action is executed.

A transition guard can be generally separated to three sub-guards (parts) (1) input availability sub-guard which is a guard on input ports that is satisfied when they have the specified number of tokens available, (2) output availability sub-guards which is a guard on output ports that is satisfied when they have the specified number of buffering space available, and (3) boolean sub-guard which is simply a boolean expression. The boolean sub-guard can be written inline or as a separate C++ function that returns boolean value. Note that sub-guard function are not allowed to change actor's internal state.

Beside transition guards, an actor firing action may also be specified for a state transition. Firing actions are defined as C++ procedures. Their input parameters are given in the FSM definition. A firing action can manipulate the internal state of the actor. However, note that input ports are only readable, while output ports are only writable. In C++ terms, the whole SysteMoC actor network is defined within a class derived from class `smoc_graph`. Inside it, actor instances and port connections are defined. Connections are defined as `smoc_fifo`. In turn, an actor is C++ class derived from class `smoc_actor`. Formal definitions of previously discussed entities is discussed in the following based on [71].

Definition 2.5. Actor network graph: is a directed bipartite graph $g = (A, C, P, E)$ containing a set of actors A, a set of channels C, a channel parameter function $P : C \rightarrow N^{\infty} \times V^{*}$ which associates with each channel $c \in C$ its buffer size $n \in N^{\infty} = \{1, 2, 3, ...\infty\}$, and possibly also a non-empty sequence $v \in V$ of initial tokens , and finally a set of directed edges $E \subseteq (C \times A.\mathcal{I}) \cup (A.\mathcal{O} \times C)$. The edges are further constraint such that exactly one edge is incident to each actor port and the in-degree and out-degree of each channel in the graph is exactly one.

Definition 2.6. Actor: is the tuple $a = (\mathcal{P}, \mathcal{F}, \mathcal{R})$ where $\mathcal{P}$ is a set of input and output ports $\mathcal{P} = \mathcal{I} \cup \mathcal{O}$, $\mathcal{F}$ is a set of functions, $\mathcal{R}$ is the firing FSM.

Definition 2.7. Firing-FSM: of an actor $a \in A$ is a tuple $a.\mathcal{R} = (T, Q_{firing}, q_0)$ containing a finite set of firing state transitions T, a finite set of firing states Q_{firing} and an initial firing state $q_0 \in Q_{firing}$

Definition 2.8. Firing transition: is a tuple $t = (q_{firing}, k, f_{action}, \acute{q}_{firing}) \in T$ containing the current firing state $q_{firing} \in Q_{firing}$, an activation pattern k, the associated action $f_{action} \in a.\mathcal{F}$, and the next firing state $\acute{q}_{firing} \in Q_{firing}$. The activation pattern k is a boolean function which decides if transition t can be taken (true) or not (false).

Figure 2.4 from [84] shows the graphical representation of a SysteMoC actor. It depicts actor (a_2) of Newton square root approximation algorithm (see [84] for more details). Noteworthy is the set input ports $a_2.\mathcal{I} = \{i_1, i_2\}$ and output ports $a_2.\mathcal{O} = \{o_1, o_2\}$. Actor's functionality $a_2.\mathcal{F}$ is defined by four functions among them $f_{\texttt{check}}$ is a boolean sub-guard function (used in firing transitions only). Note that Firing-FSM $a_2.\mathcal{R}$ has only two states $q_{\texttt{start}}$ and $q_{\texttt{loop}}$. Note also that this actor representation is verbose, therefore we will be using a simplified version of it in the rest of the thesis.

We move now to discuss actual SysteMoC syntax to get a feeling on how the previously discussed entities are represented in C++ code. The discussion will continue with the Sqr root actor network example of [84]. We focus on the structure of class definitions and omit details of the algorithm and what actually gets calculated since it's not relevant. We follow the same top-to-down approach starting with actor network definition.

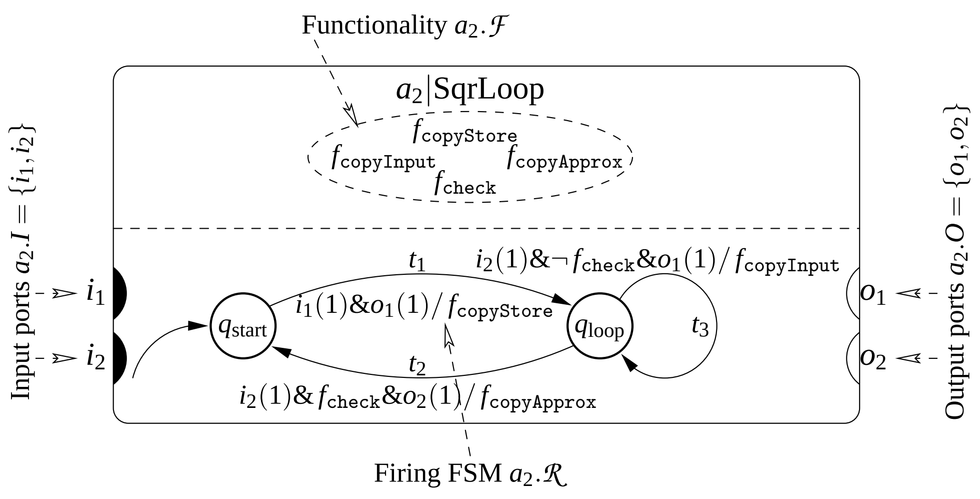

Functionality $a_2.\mathcal{F}$
$a_2|\text{SqrLoop}$
$f_{\texttt{copyStore}}$
$f_{\texttt{copyInput}}$
$f_{\texttt{copyApprox}}$
$f_{\texttt{check}}$
Input ports $a_2.I = \{i_1, i_2\}$
i_1
i_2
t_1
$i_2(1)\&\neg f_{\texttt{check}}\&o_1(1)/f_{\texttt{copyInput}}$
$i_1(1)\&o_1(1)/f_{\texttt{copyStore}}$
q_{start}
q_{loop}
t_3
t_2
$i_2(1)\&f_{\texttt{check}}\&o_2(1)/f_{\texttt{copyApprox}}$
o_1
o_2
Output ports $a_2.O = \{o_1, o_2\}$
Firing FSM $a_2.\mathcal{R}$

FIGURE 2.4: A graphical representation of an actor

```
01   // Declare network graph class SqrRoot
02   class SqrRoot: public smoc_graph {
03   protected: // Actors are C++ objects
04      Src a1; SqrLoop a2; Approx a3; Dup a4; Sink a5;
05
06   public: // Constructor assembles network graph
07   SqrRoot( sc_module_name name ): smoc_graph(name)
08        ,a1("a1", 50) // parametrized Src actor a1
09        ,a2("a2"), a3("a3"), a4("a4"), a5("a5")
10        {
11       // The network graph is instantiated
12       // in the constructor
13       connectNodePorts(a1.o1, a2.i1);
14       connectNodePorts(a2.o1, a3.i1);
15       // Setting FIFO to size 1
16       connectNodePorts(a3.o1, a4.i1, smoc_fifo<double>(1));
17       // Providing an initial token 2 to a FIFO
18       connectNodePorts(a4.o1, a3.i2, smoc_fifo<double>() << 2);
19       connectNodePorts(a4.o2, a2.i2);
20       connectNodePorts(a2.o2, a5.i1);
21     }
22   };
```

LISTING 2.10: SqrRoot actor network graph definition

Class definition of the actor network graph is depicted in Listing 2.10. Two pieces of information are of interest to us, firstly, there is the declaration of actor instances available in the network (#line 4). Secondly, there is the definition of network topology in the form of a list of connections. Each connection is defined using `connectNodePorts` from a source port to a destination port. The default connection is a empty FIFO with size 1. One can configure the size of the FIFO as well as the initial tokens.

```
01   // All actor classes are derived from smoc_actor
02   class SqrLoop: public smoc_actor {
03   public: // Declaration of input and output ports
04      smoc_port_in<double> i1, i2;
05      smoc_port_out<double> o1, o2;
06   private:
07   // Functionality consists of local variables
08      double tmp_i1;
09   // and actions transforming func state and data,
10   void copyStore() { o1[0] = tmp_i1 = i1[0]; }
11   void copyInput() { o1[0] = tmp_i1; }
12   void copyApprox() { o2[0] = i2[0]; }
13   // and guard definitions which are used
14   // only by the firing FSM
15   bool check() const
16   { return fabs(tmp_i1-i2[0]*i2[0]) < BOUND; }
17
18   // State declaration for the firing FSM
19   smoc_firing_state start, loop;
20
21   public: // Constructor builds firing FSM
22       SqrLoop(sc_module_name name);// code not shown here
23   };
```

Listing 2.11: Sqr root actor definition

Moving to the class definition of `SqrLoop` actor itself depicted in Listing 2.11. Note the definition of input ports `smoc_port_in` and output ports `smoc_port_out` of the actor in lines #4 and #5. In line #8, a local actor variable is defined. After that, we find the definitions of actor actions `copyStore`, `copyInput`, and `copyApprox`. Boolean guard `check` is defined in line #15. Firing FSM states `start` and `loop` are defined in line #19. The definition of the firing FSM is done in the constructor of `SqrLoop` actor which is depicted in Listing 2.11. It is easy to get familiar with how state transitions can be defined. One can also compare the definition in Listing 2.11 with the graphical definition in Figure 2.4.

```
SqrLoop::SqrLoop(sc_module_name name)
: smoc_actor(name, start /* Initial state */) {
start = // Start state declaration
// Transition t1 with activation pattern re-
// quiring at least one token in channel con-
// nected to port i1 and free space for one
// token in channel connected to port o1.
   i1(1) >> o1(1) >>
   CALL(copyStore) >> loop; // t1
// State transition declaration for second state
loop =
   (i2(1) && GUARD(check)) >> o2(1) >>
   CALL(copyApprox) >> start // t2
   | (i2(1) && !GUARD(check)) >> o1(1) >>
   CALL(copyInput) >> loop; // t3
}
```

Listing 2.12: Sqr root actor constructor with firing FSM definition

Chapter 3

Implementation

3.1 Considered approach

Our goal in this work is to build a framework for the simulation of *elasticized* synchronous specifications written in the synchronous language Quartz [70]. The original model is elasticized by first partitioning it to components. Components are then linked together in a network topology where links can have arbitrary delay. Communication is synchronous i. e. delay across it is an integer number of clock cycles. Communication is simulated at a high level of abstraction with a simple abstract protocol that has valid (forward) channel and back-pressure (backward) channel. It is relatively simple to map our abstract communication protocol to one of the protocols discussed previously in section 1.5.

Some issues need to be addressed in order to provide a viable simulation solution, firstly we need to identify a suitable DPN simulation framework. Secondly, we need to resolve the expected semantic miss-match between the distributed synchronous specification and DPN semantics. It should be stressed that the simulation platform obtained by code generation should accurately preserve the semantics of the original synchronous system. Thirdly, there should be a clear path to synthesize hardware circuits out of the partitioned synchronous system. Hardware synthesis is a key criteria that should be kept in mind for future work.

To address the first issue, two DPN simulation frameworks have promising potential. OpenDF [86] a data-flow simulation framework based on *CAL* language, and SysteMoC [71, 72] a SystemC based actor-oriented (DPN) simulation framework. Partitioning and synthesis of Quartz programs to OpenDF/CAL networks has already been considered in an earlier work [87]. However, it was considered in a pure DPN context. In that work, guarded actions would be clustered in DPN nodes based on some specific criteria (e.g. writing to a single variable). No distinction has been made between data-flow and control-flow guarded actions and the network would run completely asynchronously. That makes the resulting DPN not suitable for hardware synthesis.

In contrast, SysteMoC can be a better option for our purposes. Firstly, SysteMoC node behavior is defined by means of Extended Finite State Machine (EFSM) which provides a clear path to hardware synthesis, particularly when compared to the rule-based node behavior specification of CAL. Secondly, EFSM based analysis and optimization of Quartz programs has already been considered in previous work by Bai et al. [88, 89]. Thirdly, SysteMoC is essentially a library written on top of SystemC, the defacto standard for hardware/software modeling. That provides us with access to the huge IP and modeling capabilities of SystemC.

The general flow of our approach is depicted in Figure 3.1. The flow starts from a synchronous specification in the form of AIF system. Basically, an AIF system is a set of control-flow and data-flow guarded actions. It's obtained from the compilation of a Quartz program. The AIF system should be partitioned to a set of synchronous AIF components and an AIF topology. Each AIF component is a synchronous program by itself. An AIF topology is at its simplest forms a list of port connections. Each connection is just a specification of source port (of output port an AIF component) and destination port (of input port of another AIF component) and minimum delay across connection. The topology enables building the network graph where each AIF component is a node.

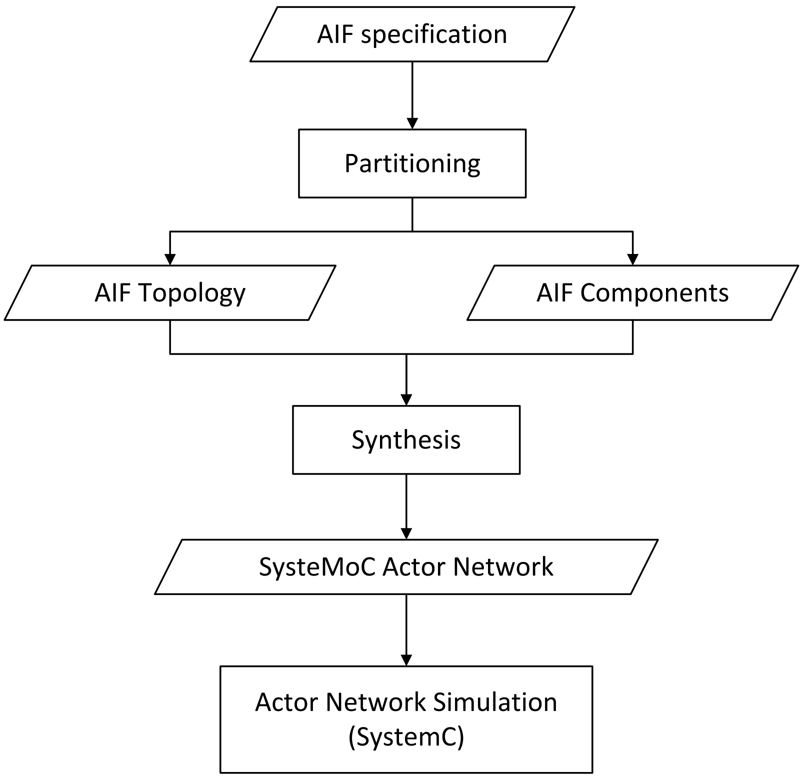


FIGURE 3.1: Flow of considered approach

Note that automatic partitioning of original AIF system to obtain topology and components has not been considered in this work. Automatic partitioning can provide enough work for a thesis by itself. Additionally, we think that the AIF format should be changed to support it. In our work, the (manually) obtained topology and list of AIF components are given inputs to the next stage *synthesis*. This part constitutes the core work of this thesis. We map each AIF component to a corresponding SysteMoC actor. EFSM of the component is first extracted and then synthesized directly to the Firing-FSM syntax of SysteMoC. DGAs of each state are synthesized to an action in SysteMoC. More details on the synthesis process is provided later. For the topology, each connection is mapped to a connection in SysteMoC. The generated code of the whole network graph should be compiled and linked to SystemC and SysteMoC libraries. The resulting executable can simulate the network behavior based on SystemC's event-driven simulation kernel. We compare in the following our approach with similar approaches and show its merits:

- We start our flow from a high level representation of synchronous systems in the form of guarded actions. Compared to conventional RTL design. Our approach reliefs the designer from the cycle-by-cycle FSM based control of the system. Guarded-actions provide a better way for describing concurrency and are more suitable for hardware synthesis [90].

- Vijayaraghavan et al. have considered in [91] the synthesis of elastic data-flow networks from guarded actions. There proposed language has a mix of atomic guarded actions and Verilog code. Our guarded actions representation is synthesized from the synchronous language Quartz [70]. A complete language with precise semantics that enables verification and analysis in addition to synthesis of hardware as well as software.

- Brandt et al. [92] have considered synthesizing a (not partitioned) Quartz synchronous system to SystemC. Compared to their work, we show how an elasticized synchronous system can be synthesized and simulated in SysteMoC. We not only bridge the semantics but also show a flexible way for simulating a generic elastic protocol at a high level of abstraction.

3.2 Synthesis process

We have discussed in the previous section the general flow of our approach. This section focuses on a specific part of that flow, namely, the synthesis process. The input to the synthesis process is a list of AIF components and a list of connections that describe the topology. Ideally, these lists should be obtained from an automatic partitioning process, however we are creating those lists manually in this work. In our manual partitioning procedure, we consider each sub-module defined in the original Quartz program as potential AIF component. List of port connections is also recovered manually based on variable names from the sub-modules. We characterize the discussed entities in a more formal way in the following.

Definition 3.1. AIF Component: is the tuple $AC = (\mathcal{P}, \mathcal{L}, \mathcal{CF}, \mathcal{DF})$ where ports is a set of input and output ports $\mathcal{P} = \mathcal{I} \cup \mathcal{O}$. $\mathcal{L}$ is the set of local variables which define the state of the component. $\mathcal{CF}$ and $\mathcal{DF}$ define the list of control-flow and data-flow guarded actions respectively. A component can be seen as a class type that can be instantiated multiple times in the same network.

Definition 3.2. Port Identifier: is the tuple $(AC, ins, qname)$ where AC is the AIF component, *ins* is the instance identifier of AIF component (remember that a network can have more than one instance of an AIF component), and *qname* is a unique qualified name of the port in the AIF component.

Definition 3.3. Port Connection: is the tuple $(srcPort, dstPort, delay)$ where $srcPort$ is the source output port of a component instance and $dstPort$ is a destination input port of another component instance. *delay* represents the *minimum* number of clock cycles needed to transport a token though the connection. Delay can increase in case the channel is congested.

Note that the definition 3.1 beers similarity to the definition 2.6 of SysteMoC actor. The only difference is that an actor is described by its firing FSM, whereas a component is characterized by $(\mathcal{CF}, \mathcal{DF})$. Fortunately, a corresponding EFSM can be generated for a component based on availability of $(\mathcal{CF}, \mathcal{DF})$. $\mathcal{CF}$ is used to generated states and transition (including conditions) of the EFSM, while $\mathcal{DF}$ is used to generate the actions corresponding to each state. The process of EFSM generation will be discussed later in section 3.3.

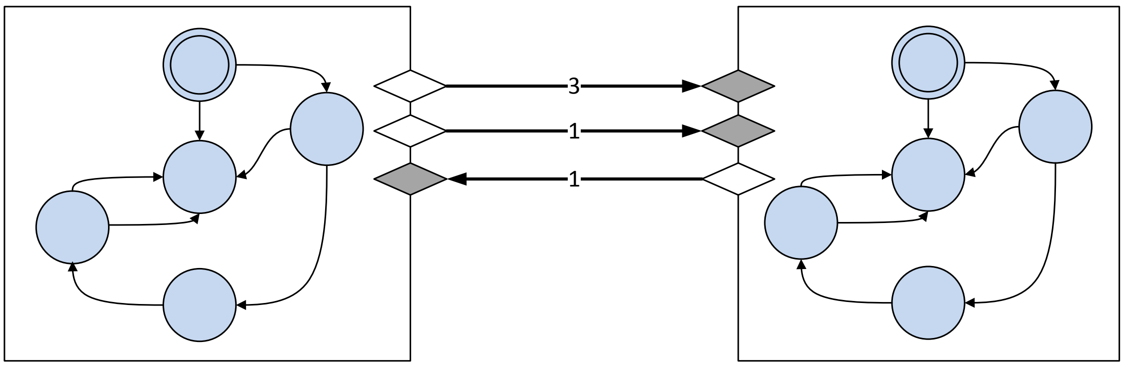


FIGURE 3.2: An abstract actor communication model

Figure 3.2 depicts an abstract communication model involving two actors. Each actor is synthesized from an AIF component after EFSM generation phase. An EFSM is shown inside each actor. Arrows correspond to port connection in the topology. The arrow starts from *srcPort* and points to *dstPort*. Each connection is annotated with its corresponding delay. Note that the minimum delay on each connection is one, otherwise output of an actor can appear instantaneously on the input of the other actor which is not possible in real hardware.

Note that our focus is on the synthesis of partitioned (elasticized) synchronous system only. Therefore, we assume that partitioning process is semantically correct. We discuss in the following some fundamental issues that need to be tackled to have semantically correct synthesis:

1. **Notion of time**: SysteMoC was originally devised as an actor-oriented modeling library in SystemC. Actor computations are driven by data availability instead of time. That is, it takes place as soon as sufficient data is available and a transition condition is satisfied. For our purposes, evaluation of firing conditions should be triggered by a clock and firing should happen once and only once per clock cycle.

2. **Notion of synchronous reaction**: synchronous elastic components execute in reaction steps. In each such step, values on all inputs ports are read, computation is done, and then output is produced on all output ports. That means that if there was a non-valid input on any input port or if there was no buffer space on any output port, the reaction can't take place and the actor should wait (patiently) until the next clock cycle.

3. **Elastic channels**: a port connection should have a certain delay which needs to be correctly modeled. Delay is at least one cycle since outputs should be available at the next clock cycle. In SysteMoC, a produced token is instantaneously available to be consumed which shouldn't be allowed.

4. **Elastic channel buffering**: having delay in the channel dictates that a channel should be able of temporarily storing tokens while being transported. That is, a channel acts as a FIFO where tokens are advances one-step forward at each clock cycle. Obviously, tokens can't override other tokens which means that a token can't advance if an earlier token can't advance.

5. **Hardware applicability**: since we are modeling hardware, we can not consume (produce) more than one token at each clock cycle from input ports (to output ports). This assumption is also compatible with the synchronous MoC.

To address those issues we had first to introduce a clock (sc_clock) in the actor network. This clock is not defined network-wide, but internal to each actor. That is due to the actor-oriented modeling approach of SysteMoC which restricts sharing any information between actors except through ports. Ideally, each actor should wait till a clock edge event to evaluate firing conditions and execute an action if a firing condition is satisfied. Unfortunately, SysteMoC doesn't allow waiting for a clock edge condition in actor firing rule syntax. Therefore, we had to introduce a **smoc_event** (SysteMoC event) and a *notifier* method [1].

The notifier method is a SC_METHOD sensitive to clock edge. All of what a notifier method does is to set the SysteMoC event to notify (event.notify()) when a clock edge happens. Firing rules of actors has to be modified to wait for that event to happen using the firing condition `TILL(event)`. That is, firing rules are *indirectly* evaluated at clock edge only with the help of a `smoc_event`. Note also that all actor actions has to do an event.reset() when executed, otherwise an actor would be able to fire multiple times in a single clock cycle. With the help of sc_clock and smoc_event we have introduced time and synchronous reactions to the behavior of actors. Note that we also need to introduce patience in actors. A patient actor should wait until there is valid input on all input ports and there is buffering space on all output ports. This property can be specified as a condition on all actor firing rules.

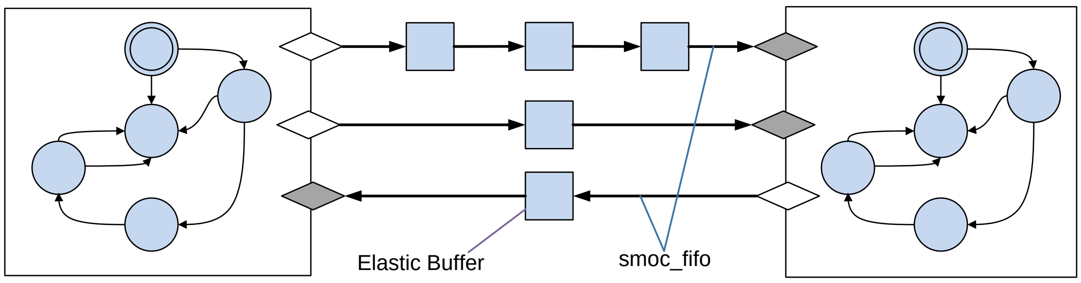


FIGURE 3.3: An model for an elastic channel

Finally, there is the issue of modeling elastic channels between actors (issues #3 and #4). In our approach, we opted for the strict separation between communication and computation. Actors specification shouldn't be modified with different connection delays. That is, we should be able of manipulating delays on different connections in a flexible way while keeping actors specification intact. In order to achieve that, we have introduced a special kind of actor called *Elastic Buffer*. The role of elastic buffers is to model delay (not storage) on a channel. Adding an elastic buffer on a connection means adding a delay of one clock cycle on that connection. We kept a one-to-one mapping between delay cycles and the number elastic buffers on a connection.

Figure 3.3 depicts the same abstract connection topology of Figure 3.2. Connections have been segmented by a number of elastic buffers equal to their delay. Each segment (arrow) represent a **smoc_fifo** (SysteMoC FIFO) capable of buffering exactly one token. That means that a connection with a delay of n (number of elastic buffers) would be able to buffer $n+1$ tokens in total before the producing actor goes to *patience mode*. Beside bounding the minimum delay with the number of elastic buffer our model has the nice property that

[1]Thanks to Joachim Falk of HW/SW co-design group, Erlangen-Nuremberg university for the tip.

the maximum delay is bounded only by responsiveness of consuming actor. The separation between delay (done by elastic buffers) and buffering (done by smoc_fifo) provides a clean and simple implementation.

Elastic buffers are represented by the actor model depicted in Figure 3.4. An *empty* elastic buffer should wait (patiently) until there is valid input on its input port. Then, valid input token should be stored internally which make the buffer *full*. On the next clock cycle, if there was a buffer space in the output port then one of two actions can happen; the elastic buffer can either forward the stored token to be *empty* again, or it can forward the stored value and at the same time store a valid input value if available and thus stay *full*.

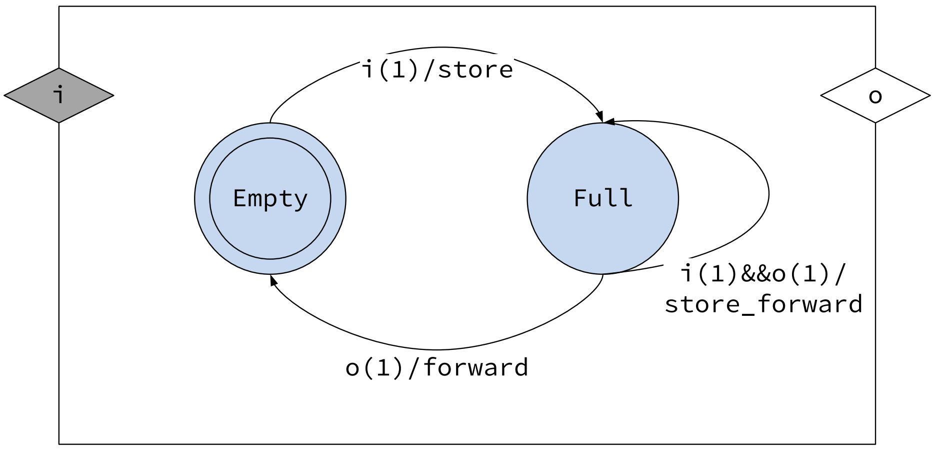


FIGURE 3.4: An model for an actor representing an elastic buffer

Elastic channel can also be modeled by (1) replacing smoc_fifo of one token capacity with rendezvous smoc_fifo, and (2) providing more buffering capacity in the elastic buffer. This approach would more closely resemble the SELF protocol and elastic circuit approach of Cortadella et al. [50, 51]. One drawback of this approach is that it would break the direct mapping between channel delay and number of elastic buffers. Additionally, it requires discriminating between the case of elastic buffer (two token buffer capacity) and elastic half buffer (one token buffer capacity) which also complicates synthesis. Also, current version of SysteMoC doesn't support rendezvous semantics. Elastic channel behavior can also be modeled using a single elastic buffer per connection instead of n number of elastic buffers as we have done. The channel buffer in this case would be more complex than our elastic buffer since it has to simulate connection delay and storage. We went for simplicity in our design.

3.3 EFSM generation

We consider more closely the synthesis of an AIF component to a corresponding SysteMoC actor. This process starts by generating an Extended Finite State Machine (EFSM) of the AIF component which will be discussed in detail in this section. Note that it's possible to model a complete Quartz synchronous system as EFSM with a single state. In each reaction, all inputs are read, all guarded actions are evaluated and all outputs are produced. In SysteMoC terms, the system would be mapped to an actor with a single state and a single action. That action would contain all guarded actions of the system. Obviously, this representation is far from being efficient and hard to optimize.

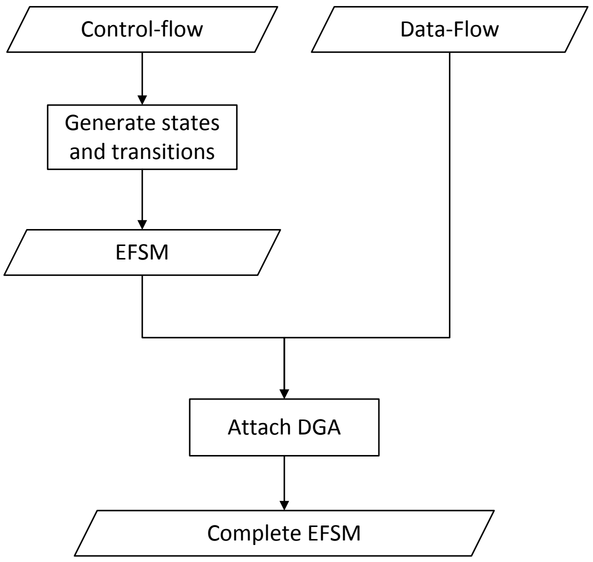


FIGURE 3.5: Flow of EFSM generation

Figure 3.5 depicts the flow of EFSM generation. CGAs are used to generate a basic EFSM. Then, EFSM is traversed and DGAs are attached to each visited state. The result is a complete EFSM that can be used directly for SysteMoC actor synthesis, or it can be first optimized [88, 89] before being used. We have not considered EFSM optimization in this work. Interested reader should refer to those references for details. Note that CGAs are of the form $\langle \gamma \rightarrow next(\ell) = true \rangle$ where γ is a boolean condition and $\ell \in \mathcal{L}$ is a control flow label. Basically, a control flow label is a boolean variable. EFSM is formally defined in definition 3.4. Note that an EFSM is a special form of a Firing FSM (definition 2.7) where we are restricted to one-to-one mapping between states and firing actions.

Each state s in the EFSM is *uniquely* defined by a set of control flow labels $Labels(s) \subset \mathcal{L}$. A label environment $env(\mathcal{L})$ is a value assignment to all labels in the Quartz program. A label environment of state s can be obtained by setting state labels to true and all other program labels to false. Formally, $env_s(\mathcal{L})$ is an injective function $\mathcal{L} \rightarrow \mathbb{B}$ where $\forall \ell \in Labels(s), \ell \rightarrow$ **true** otherwise $\ell \rightarrow$ **false**. An evaluation of boolean condition γ based on label environment of state s , written as $eval(\gamma, s)$, is replacing all label variables in γ with their boolean value in $env_s(\mathcal{L})$. The result boolean condition is simplified using boolean algebra to yield either $true$, $false$, or another boolean condition γ'. Note that γ' would be written in terms of input and local variables only in case of CGAs. On the other hand, guards of DGA can also be written in terms of output variables.

Definition 3.4. Extended Finite State Machine (EFSM): is a tuple $(S, s0, T, D)$, where S is a set of states, $s0 \in S$ is the initial state, and $T \subseteq (S \times C \times S)$ is a finite set of transition relations where C is the set of transition conditions. D is a mapping $S \rightarrow D$, which assigns each state $s \in S$ a set of DGAs $D(s) \subseteq D$ which are executed in state s.

Figure 3.5 depicts two processes, namely, generation of EFSM graph and attaching DGAs to states. We will start discussing the second process since it's relatively simple. Basically, we do a Breadth-First-Search (BFS) traversal to the FSM graph generated in the first process. For each visited state, we do the procedure described in Figure 3.6 to attach DGAs to it. The procedure starts by getting the label environment $env_s(\mathcal{L})$ of the visited state. Based on $env_s(\mathcal{L})$, guards of all DGAs of the AIF Component needs to be evaluated. DGA with guards that evaluate to false shall be discarded. All other DGAs are rewritten to use the resulting simplified guards and then get attached to the visited state.

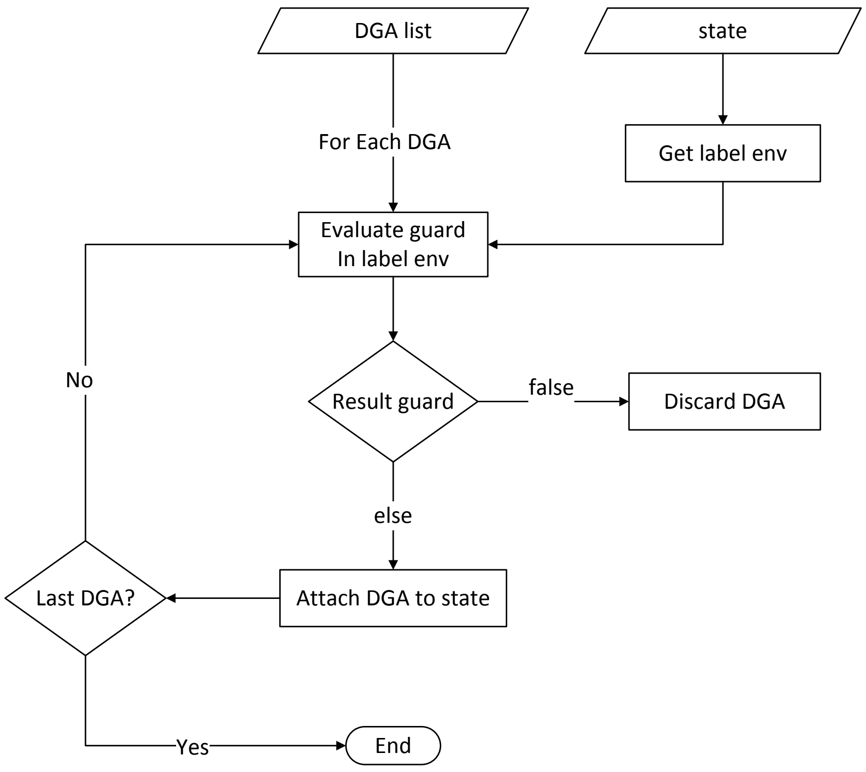


Figure 3.6: Flow of attaching DGAs to a given state

Now, we discuss the process of EFSM generation. Generating EFSM states and transitions is also done by a BFS traversal. However, in contrast to DGA attachment process

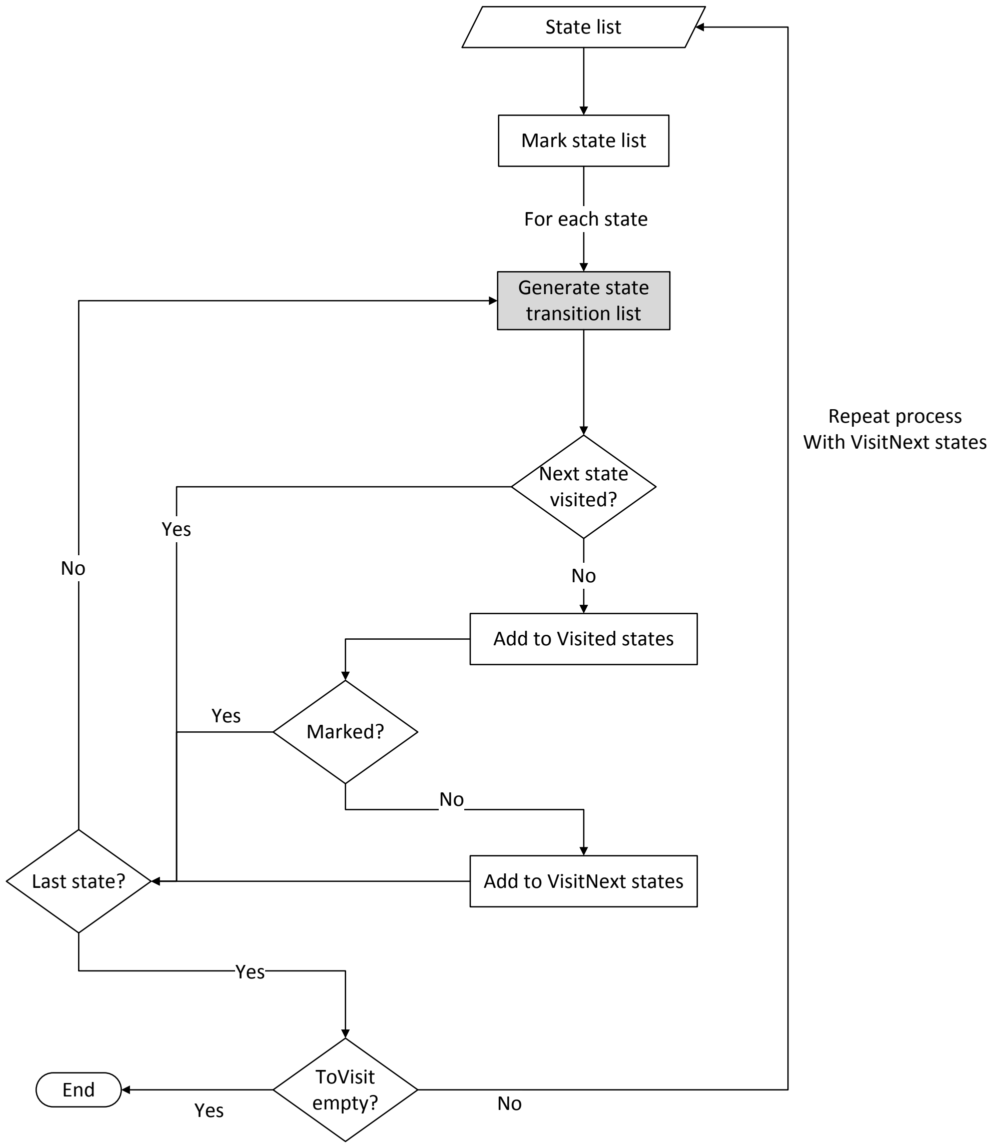


FIGURE 3.7: On-the-fly EFSM generation

discussed previously, we do BFS traversal *on-the-fly* here. That is, given a list of states, we generate the transition list of each state individually. Among the states generated we get the unique set of states that should be visited next. This process terminates when we have visited all states i. e. the set of unique states to be visited is empty. Obviously, the traversal starts with a list containing the starting state. Figure 3.7 depicts the procedure. To add a generated state to the set of states to be visited we need to make sure that (1) it was not visited in before, and (2) it is not going to be visited in this iteration i. e. not marked.

The highlighted procedure in Figure 3.7 generates the state transition list for a given state which requires more discussion. Flow of state transition generation is depicted in Figure 3.8. Given a state and CGAs list as an input, the procedure starts by separating conditional labels from unconditional labels based on $env_s(\mathcal{L})$ guard evaluation. Note that a state transition

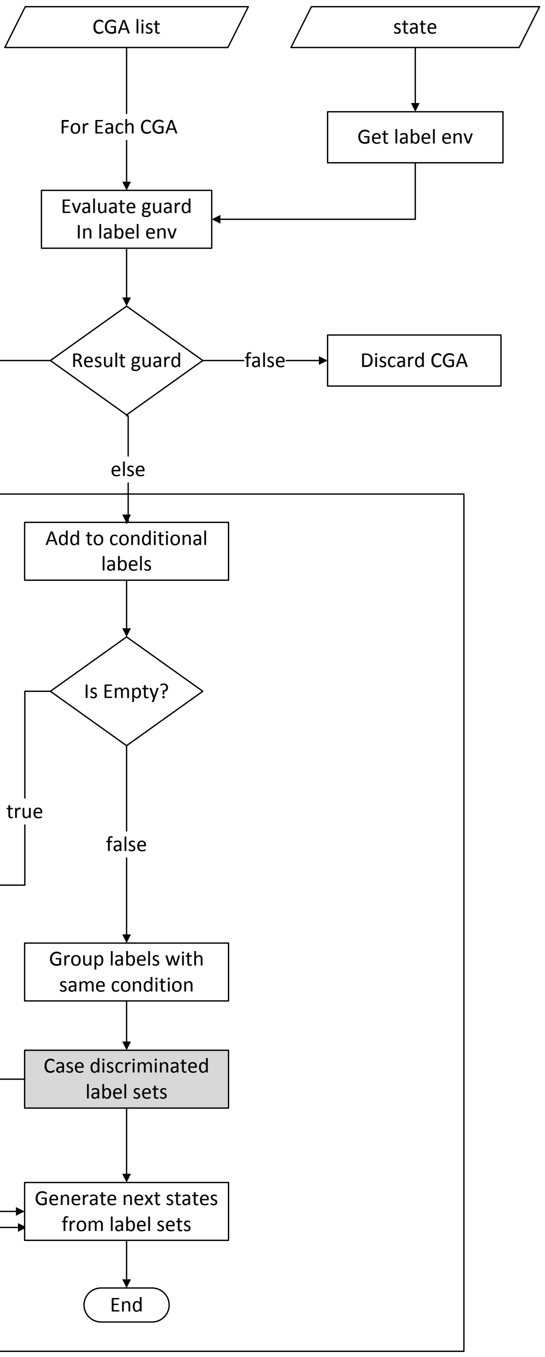

CGA list
state
For Each CGA
Get label env
Evaluate guard In label env
true
Result guard
false
Discard CGA
else
Add to unconditional labels
Add to conditional labels
Is Empty?
Is Empty?
true
false
true
false
Group unconditional labels in a single state
Group labels with same condition
Attach unconditional labels to label sets
Case discriminated label sets
Generate next states from label sets
End

FIGURE 3.8: Flow of generating next state transition of a given state

list at this stage is represented with a list of tuples (γ, ℓ) where γ is a transition condition and ℓ is a label. If $\gamma =$ **true** we say that the transition label is unconditional, otherwise the transition label is conditional.

After finishing the separation stage, the procedure starts from the list of conditional and unconditional labels. If both of them was empty we generate a transition to the **final** state (not depicted in Figure 3.8). Otherwise, if conditional labels list was empty, we group unconditional labels in a single generated state (remember that a state is identified by a set of labels). The case when the conditional label list is not empty requires more attention. Firstly, we group labels with the same condition together which means that the state transition list is represented now with a list of tuples $(\gamma, \mathcal{L})$ where $\mathcal{L}$ is a list of labels. Note that $\mathcal{L}$ can have a single label if its corresponding γ was unique. Grouped labels are then given as input to the next stage, namely, case discrimination which is again highlighted since it requires further discussion.

```
1-    engineOff => __ell000
2-    !engineOff&beltUnfastened => __ell003
3-    !engineOff&cruise => __call001.__ell007
4-    !engineOff&!beltUnfastened  => __ell002
5-    !engineOff&!cruise => __call000.__ell004
```

Listing 3.1: Before case discrimination

To better understand the motivation behind case discrimination we need to have a look at an actual example. Let's consider the set of conditional transitions [2] depicted in Listing 3.1. If `engineOff` was valid, then we know that the only label we can go to in the next state is `__ell000`. However, if `!engineOff&!cruise` was valid we know that we will be going to label `__call000.__ell004`, but we will additionally be going to `__ell003` or `__ell002` depending on `beltUnfastened`. That means that label transitions of Listing 3.1 can't be converted directly to state transitions since the behavior of the system is not correctly captured.

In order to correctly capture that behavior of system we need to make label transition of Listing 3.1 well-formed transitions. Well-formed transitions have two properties, firstly, the canonical form of the conjunction of any two guards is `false` i. e. guards are orthogonal to each other. Secondly, they capture the whole behavior of the system which means that any transition that is valid in the original form should also be valid after making the transitions well-formed. Note that guard label transition #1 is orthogonal to other guards but this is not the case for other guards. Converting transitions of Listing 3.1 to the well-formed form yields that transitions of Listing 3.2. It is easy to check that the two properties hold in the well-formed transitions.

```
1-    engineOff => __ell000
2-    !engineOff&beltUnfastened&!cruise => __ell003,__call000.__ell004
3-    !engineOff&beltUnfastened&cruise => __ell003,__call001.__ell007
4-    !engineOff&!beltUnfastened&cruise => __ell002,__call001.__ell007
5-    !engineOff&!beltUnfastened&!cruise => __ell002,__call000.__ell004
```

Listing 3.2: After case discrimination

[2] These transitions are a result of applying EFSM generation to CruiseControl example available on http://www.averest.org/examples/

Label transitions of Listing 3.2 can be converted to state transition where each group of labels will be mapped to a new state. We are now sure that the behavior of the system is deterministic since no two transition can be valid at the same time. This is also a required property for later synthesis to SysteMoC. Remember that SysteMoC engine will be choosing *non-deterministically* one transition if two are valid at the same time which should be avoided.

We discuss now the procedure of doing case discrimination i. e. converting a set of labeled transitions to well-formed transitions. Pseudo code of the algorithm is shown in Listing 3.3. The algorithm starts from a *currentCondition* equals to **true**, an empty list of *fixedLabels* and the list of transitions *unfixedLabelTransitions*. Basically, the algorithm explores recursively the conjunction of all guard combinations. If a guard is assumed to hold at a certain stage e. g. `engineOff`, its corresponding transition labels are added to the fixed list of labels. Otherwise, we try the case that the guard doesn't hold, however we do not add any labels to *fixedLabels* in that case. The algorithm terminates when *unfixedLabelTransitions* is empty and the resulting *currentGuard* and *fixedLabels* will be added to the list of well-formed transitions.

```
Procedure GenerateWellFormedTransition(currentGuard, fixedLabelList,
    unfixedLabelTransitions)
   if unfixedLabelTransitions is empty then
      return (currentGuard, fixedLabelList) // This is a well formed transition
   else
      head := Head(unfixedLabelTransitions)
      tail := Tail(unfixedLabelTransitions)

      assumePositive := MakeCanonical(head.guard && currentGuard)
      assumeNegative := MakeCanonical(!(head.guard) && currentGuard)

      if (assumePositive != false) then
        GenerateWellFormedTransition(assumePositive, Append (fixedLabeList,
    head.labels), tail)

      if (assumeNegative != false) then
        GenerateWellFormedTransition(assumeNegative, fixedLabeList, tail)
```

Listing 3.3: Algorithm of case discrimination

For n transitions, it is easy to see that 2^n guard combinations should be tried in total. Fortunately, we can *short-circuit* many of those trials as soon as we discover that the conjunction of their guards yields **false**. In order to do boolean operations we had to develop the function `MakeCanonical` which returns a canonical form of a boolean expression. The canonical form is obtained by extracting and sorting the list of boolean variables in the given expression. Then, the *ITE* operator is applied recursively to the boolean expression based on variable's order. Note that the algorithm of Listing 3.3 may yield some transitions that have no labels at all. These transitions are mapped to the **final** state[3].

To summarize this section, we first discussed the complete flow of EFSM generation (Figure 3.5) which consists of two processes, EFSM graph generation and DGA attachment. The simpler process of DGA attachment has been discussed first (Figure 3.6). Then, EFSM graph generation has been discussed in increasing detail starting for on-the-fly generation of the entire EFSM (Figure 3.7). Then focusing on generating state transitions of a single state (Figure 3.8). Finally, we concluded with the case discrimination algorithm of Listing 3.3. The result is the complete EFSM that will be used in the next section for synthesis.

3.4 Synthesis of actor network

We have discussed in section 3.2 the main design decisions we have taken in order to synthesize a network of AIF components to corresponding network of actors. We will discuss that process in this section in greater detail. Our discussion of synthesis details has been separated into two parts for better readability. Firstly, the synthesis of actor network class and definitions of elastic buffer classes is discussed in this section. Secondly, the synthesis of a single AIF component to its corresponding actor, which will be discussed in the next section 3.5.

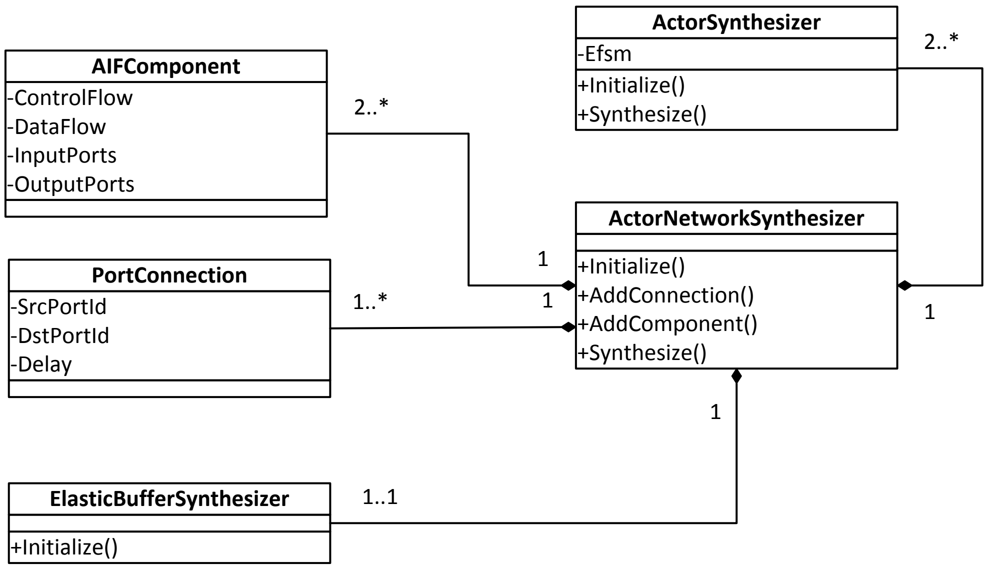


FIGURE 3.9: Class diagram of main entities

Figure 3.9 depicts a class diagram showing the main defined entities and their relation. At the core of our implementation is the `ActorNetworkSynthesizer` class which acts as an interface for using the implemented synthesis library. To define an actor network, one needs

[3] Only in case unconditional label's list was empty.

to define the partitioned synchronous system to the network synthesizer. This is done by adding all AIFComponents (see definition 3.1) and their corresponding port connections (see definition 3.3. When adding an `AIFComponent`, the number of instances of that component should be specified, otherwise it's assumed to be one. Then, the network synthesizer needs to be initialized where all the magic happens!

In the initialization of the `ActorNetworkSynthesizer` every `AIFComponent` is transformed to a `Efsm` instance which in turn is given as input to construct an `ActorSynthesizer`. `ActorSynthesizer` is responsible for synthesizing the SysteMoC class definition corresponding to an `AIFComponent`. Initialization of the actor network includes parsing all connections and inferring the type and number of elastic buffers needed. One `ElasticBufferSynthesizer` instance is responsible for synthesizing the class definition and instance declarations of elastic buffers. Note that an `ActorNetworkSynthesizer` needs to be re-initialized whenever the topology (port connections or components) changes. Synthesis to c++ text can be done multiple times after initialization at no extra computation cost.

```
class Graph: public smoc_graph {
protected:
   [[Actor Declarations]]
   [[Elastic Buffer Declarations]]
public:
    Graph(sc_module_name name)
       : smoc_graph(name)
        , [[Actor and Buffer Initialization]]
   {
     [[Port Connections]]
   }
};
```

Listing 3.4: Code generator template for network graph

Synthesis of the network graph class done by `ActorNetworkSynthesizer` is described based on its code generation template shown in Listing 3.4. Basically, fields defined between double brackets are automatically generated by the synthesizer. `ActorNetworkSynthesizer` generates all of those fields itself except for *Elastic buffer declaration* field which is generated by with the aid of `ElasticBufferSynthesizer`. These template fields will be described in the following:

- **Actor declarations**: these are the declarations of actor instances in the network. The class type name of each actor is based on the name of `AIFComponent`. Instance names are automatically generated.

- **Elastic buffer declarations**: these are the declarations of elastic buffer instances in the network. Their instance names is automatically generated based on the names of port identifiers that it connects and the delay number on the connection.
- **Actor and buffer Initialization**: straightforward constructor instance initialization based on the name of each instance.
- **Port connections**: a list of port connection where each port connection in the original topology is mapped to multiple connection definitions here based on the delay of the connection.

In addition to elastic buffer declarations, the `ElasticBufferSynthesizer` object instance is responsible of generating an elastic buffer class definition. Elastic buffer definition is a single C++ template class that is instantiated at compile-time by C++ compiler. In that way, we support almost all Quartz data types. The definition of elastic buffer is shown below in Listing 3.5 where a code generation template is depicted. Listing 3.5 also helps the reader to get acquainted to the syntax of actor class definition of SysteMoC. It is interesting to compare the text of the template with it's corresponding state machine depicted before in Figure 3.4.

```
template <typename DATATYPE>
class [[ name ]] : public smoc_actor {
    SC_HAS_PROCESS([[ name ]]);
private:
    smoc_firing_state buffer_empty;
    smoc_firing_state buffer_full;

    smoc_event cevent;
    sc_core::sc_clock clk;

    DATATYPE carry;

    void buffer_empty_action() {
      carry = in[0];
      cevent.reset();
    }

    void buffer_forward_action() {
      out[0] = carry;
      cevent.reset();
    }

    void buffer_store_forward_action() {
      out[0] = carry;
      carry = in[0];
      cevent.reset();
    }

    void notifier() {
      cevent.notify();
    }

public:
    smoc_port_out< DATATYPE > out;
    smoc_port_in< DATATYPE > in;

    [[ name ]](sc_module_name name)
    : smoc_actor(name, buffer_empty), clk(""clk"", [[ Clk Period ]],
sc_core::SC_NS)
    {


        buffer_empty =
        TILL(cevent)
        >> in(1)
        >> CALL([[ name  ]]::buffer_empty_action)
        >> buffer_full;

        buffer_full =
        TILL(cevent)
        >> in(1)
        >> out(1)
        >> CALL([[ name  ]]::buffer_store_forward_action)
        >> buffer_full |
        TILL(cevent)
        >> out(1)
        >> CALL([[ name  ]]::buffer_forward_action)
        >> buffer_empty;

        SC_METHOD(notifier);
        sensitive << clk.posedge_event();
        dont_initialize();
    }
};
```

Listing 3.5: Code generator template of elastic buffer

It can be seen by investigating Listing 3.5 that the only difference between different elastic buffer definitions is the *name* and *DATATYPE* of the elastic buffer. We have fixed the class name of in the code to be `ElasticBuffer`. Moreover, the listing deserves to be discussed further since the elastic buffer is essentially a (special) SysteMoC actor. Our discussion would serve as an introduction to the following section 3.5 on actor synthesis.

Basically, note the state declaration (line #5 and #6) and local variable declaration (line #11). Between lines #13 and #27 you can find the definitions of actor actions. Ports have been declared on lines #34 and #35. What is more interesting is the definition of the constructor starting at line #37. State transitions are defined in the constructor. Each state transition starts by waiting on a clock event `TILL(cevent)`. A clock and a SysteMoC event defined on line #8 and #9 are used to synchronize reactions and advance time.

3.5 Synthesis of an actor

3.5.1 Overall procedure

We come now to the last piece of the synthesis process which is to synthesize an AIF component to a corresponding SysteMoC actor. To this end, we describe in the follow key information required for actor synthesis and discuss how this information is obtained from the AIF component:

- **Input & output ports**: obtained directly from the interface of the AIF component. Note that AIF interface variables of type Input/Output can't be mapped directly.
- **Local variables**: obtained directly from local variable definitions of the AIF component.
- **Finite state machine**: obtained be generating the EFSM (see section 3.3) from the control-flow and data-flow guarded actions of the AIF component.
- **Action definitions**: we generate for each state in the generated EFSM a corresponding actor firing action . The action function should execute DGAs attached to that state.

We will be discussing next the synthesis details of all aforementioned issues except action definitions. Action definitions need a more elaborate treatment. Therefore, synthesis of action definitions will be addressed separately later in subsection 3.5.2. As has been done previously, synthesis details will be discussed based on a code generation template. The template of our SysteMoC actor is shown in Listing 3.6. We discuss the generated fields in sequential order starting by *name* which is a user defined name of the AIF component. We'll continue with *state declarations* field (line #8) which is generated simply by sequential passing over all states of EFSM and generating a `smoc_firing_state` for each state.

```
class [[ name ]]: public smoc_actor {
   SC_HAS_PROCESS([[ name ]]);
private:
    //  Generated local declarations, carry and flag variables
    [[ Local definitions ]]

    //  Generated state declarations
    [[ State Definitions ]]

    //  Action definitions. Each action executes attach DGA of a state
    [[ Action Definitions ]]

    //  Guard definition, guards on state transitions.
    [[ Guard Definitions ]]

    //  Local Definitions needed for clock event handling
    smoc_event  clkevent;
    sc_core::sc_clock clk;
    void notifier() {
       clkevent.notify();
   }

public:
   [[  Port Definitions  ]]

   [[ name ]](sc_module_name name)
      : smoc_actor(name,  [[ start_state ]]), clk(""clk"", [[ Clk Period]], sc_core::SC_NS)
    {
    // Setting locals to default values
    [[ Locals Default Values ]]

    // Definitions of state transitions
    [[ State Transitions ]]

    SC_METHOD(notifier);
    sensitive << clk.posedge_event();
    dont_initialize();
   }
};
```

Listing 3.6: Template of an SysteMoC actor representing an AIF component

The generated field of *action definitions* (line #11) will be discussed in detail in the next subsection 3.5.2. Additionally, for each state transition with a guard in the EFSM we need to generate a guard function. Guard functions are not allowed to change the internal state of the actor. They are generated in the field *guard definitions* (line #14). Input and output ports of the actor are generated in field *port definitions* (line #24). For each input(output) variable in the original AIF component a corresponding `smoc_port_in`(`smoc_port_out`) is generated supporting its type.

Variable flow	Variable type	Generated variables
Input	Memorized	Port variable used, no local variable generated.
	Event	Port variable used, no local variable generated.
Output	Memorized	One local carry variable.
	Event	One local carry variable and one flag variable.
Local	Memorized	One local direct variable, one local carry variable, and one flag variable.
	Event	One local direct variable, one local carry variable, and one flag variable.

TABLE 3.1: Synthesizing AIFComponent variables

We have deliberately deferred the discussion of *local definitions* field (line #5) to discuss it together with *local default values* field (line #30). Basically, the former declares local and helper variables of the actor. These variable are set to their default values in the later field. We discuss the variables defined in *local definitions* based on Table 3.1. Depending on variable flow and type we may synthesize one or more variables in order to preserve the original semantics of the synchronous AIF component. The details behind Table 3.1 are discussed in the following:

- **Input variables**: input variables of `AIFComponent` are read-only by definition. That is consistent with SysteMoC usage of input ports. Therefore, the declared input port of field *port definitions* (line #24) is used and no helper local variables are required.
- **Output variables**: output variables `AIFComponent` are writable and readable. In contrast, output ports are only writable in SysteMoC. To resolve the semantic issue, we had to introduce a helper local variable (carry) that would be read and written to. When value of the carry variable is determined in the current reaction, it should be propagated to the corresponding output port. Note however that **event** output variables need a boolean flag that is used to prevent reaction to absence to take place in the next reaction.

- **Local variables**: Locals variables are readable and writable in both AIFComponent and SysteMoC actor. However, their synthesis is not that straightforward. Helper variables in the form of a carry variable and a boolean flag need to be synthesized in order to properly handle delayed assignments on a local variable. Basically, immediate assignments are done on a *direct* variable, while delayed assignments are done on *carry* variable. The flag is used to assign the delayed value in the next reaction.

It should be noted that the motivation behind variable synthesis procedure discussed previously might not be clear from the first glance. However, issues should be clearer when we discuss how variables and helper variables are actually utilized in the next section 3.5.2. We advise the reader to have a look their to get a better idea.

We arrive now to *state transitions* which is the last synthesis field to be discussed in our actor template of Listing 3.6. All state transitions of the EFSM are encoded in SysteMoC syntax in the field *state transitions* (line #33). Remember that guards of the transitions have been previously defined in *guard definitions* (line #14). States themselves have been defined in *state definitions* (line #8). Specification of state transitions should make sure that they satisfy the following conditions:

- **Triggered at clock edge only**: this is done by prefixing every transition with the statement `TILL(clkevent)` where `clkevent` is the SyteMoC event triggered by a SystemC clock (see section 3.2).

- **Input validity**: transitions can't be triggered unless there is at least one valid token on **all** input ports. For example, for an actor with input ports `i1`, `i2`, and `i3`, all transitions should have the condition `i1(1)&&i2(1)&&i3(1)`.

- **Output buffer space**: similar to the previous condition, transitions can't be triggered unless there is exactly one buffer space on **all** output ports. That is, for an actor with output ports `o1`, `o2`, and `o3` the output port condition of all transitions should be `o1(1)&&o2(1)&&o3(1)`

- **Guard validity**: if a guard exists on a transition it should also be satisfied for the transition to trigger.

3.5.2 Synthesis of actor actions

We have discussed previously almost all the fields of the actor code generation template of Listing 3.6. We left the field of *action definitions* (line #11) to be discussed here. Basically, for every state in the EFSM, an actor action is generated in this field. An actor action should *execute* the attached data-flow guarded actions of its corresponding state. Obviously, the execution should preserve the original synchronous semantics. We are going to discuss action code generation based on the template of Listing 3.7. What complicates code generation is the following semantic mismatch issues:

- **Event variables**: while the storage of *memorized* variables is consistent with variable semantics of C++, *event* variables of Quartz programs need to be handled in a special way. Value of event variables is set per-reaction and a default reaction to absence should take place if the value is not set.
- **Read-only output ports**: output variables are both writable and readable in Quartz, whereas output ports are read-only in SysteMoC. Therefor, a *carry* variable should be generated for each output port. Read and writes are done to the carry variable. Value of the carry variable is propagated to its output port when it has been identified for the current reaction.
- **Delayed assignments**: the major source of semantic mismatch is arguably the delayed assignments of Quartz. Delayed assignment set the value of a variable in the next reaction based on evaluation of the variable environment in current reaction.

```
01   void [[ Action Name ]]() {
02
03         [[ Reaction to absence ]]
04         [[ Execution of delayed assignments of local variables ]]
05         [[ Immediate guarded actions ]]
06         [[ Propagate output values ]]
07         [[ Delayed action on local variables ]]
08         [[ Delayed actions on output variables ]]
09
10         clkevent.reset();
11   }
```

Listing 3.7: Template of an actor action

Before continuing further the reader is advised to have a look at Table 3.1 to check how variables are actually handled in the generated code. The process of synthesizing an action starts by splitting DGAs of a given state to *immediate* actions and *delayed* actions. Delayed actions are then split to delayed actions on local variables and delayed actions on output variables. Immediate action field (and fields above it) define the current variable environment of a reaction. After defining reaction variable environment, the values of carry output variables should be propagated to their corresponding output ports (line #06). Remember that all DGAs should be synthesized to read and write to output carry variables instead of output ports.

The variable environment of the next reaction is computed in the delayed actions fields (lines #7 and #8). Note that delayed action on local variables (line #7) are rewritten to write to the *carry* variable and not the *direct* local variable. The value of *direct* local variables should be preserved since it might be used by transition guards of the next state transition. Direct local variable are written to *only* in the current variable environment of the reaction i. e. before propagating output values.

Template fields of reaction to absence and delayed assignments execution (lines #3 and #4) are generated together. Basically, when a delayed assignment is executed on carry local variable a boolean flag should be set (line #07). That flag signals that the carry value should be assigned to the direct local variable in the next reaction. The execution of this delayed assignment is done in the field of line #4. Note that *memorized* output variables do not need a boolean flag since there next reaction value is already computed in line #08.

Reaction to absence code is generated by default at the beginning of the reaction for all variables of type *event*. However, it is handled somewhat differently between local variables and output variables. If the flag of a delayed assignment is set for a local variable that denotes that a delayed value is available to be copied from carry to direct variable. Thus the reaction to absence should not takes place. For event output variables, a set delayed assignment flag means simply that a reaction to absence shouldn't take place.

We discuss now the synthesis of guarded actions themselves (remember definition 2.3). A single guarded action can be synthesized as single if statement in C++. However, things are not that straightforward when synthesizing a list of guarded actions due to dependencies between them. Template fields immediate DGA (line #5), delayed DGA on local variables (line #7), and delayed DGA on output variables (line #8) are all generated by synthesizing a list of DGAs. In order to discuss dependencies between DGAs we define the partial order $(\mathcal{G}, \leq)$ where $\mathcal{G}$ is a set of DGAs and the operation $\leq$ is defined on DGA in Definition 3.5.

Definition 3.5. We define the operation $<$ (less than) on two guarded actions $G1$ and $G2$ and say that $G1 < G2$ iff $wr(G1) \in FV(G2)$. The definition can be extended to $\leq$ (less than or equal) and say that $G1 \leq G2$ iff $wr(G1) \in FV(G2) \vee wr(G1) = wr(G2)$

Note that DGAs of the field **delayed DGA on local variables** (line #7) have no dependencies between them. Remember that $wr(G)$ of all field's DGA is a local carry variable. That leaves us with discussing the synthesis of the other two fields in the following:

- **Immediate DGAs**: the dependency that can exist between DGAs here is WAR (true dependency). Basically, for two DGAs $G1$ and $G2$, if $G1 < G2$ then $G1$ should appear **before** $G2$ in the synthesized code. Note that a DGA writing to a variable defines the value of the variable in the **current** reaction. All other DGAs reading that variable should therefore be synthesized after that writing DGA.

- **Delayed DGAs on output variables**: the dependency that can exist between DGAs here is RAW (anti-dependency). Basically, for two DGAs $G1$ and $G2$, if $G1 < G2$ then

$G1$ should appear **after** $G2$ in the synthesized code. Note that a DGA writing to a variable here defines the value of the variable in the **next** reaction. All other DGAs reading that variable should therefore be synthesized before that writing DGA.

Note that, thanks to Quartz semantics, two DGAs writing to the same variable do not need to be ordered since it can't happen that both of them write to the same variable in the same reaction. Actually, this is true for immediate actions only. Write conflicts can happen between two delayed actions or between a delayed and an immediate action. Write conflicts are detected in our simulation and assertion violation error is generated if they happen.

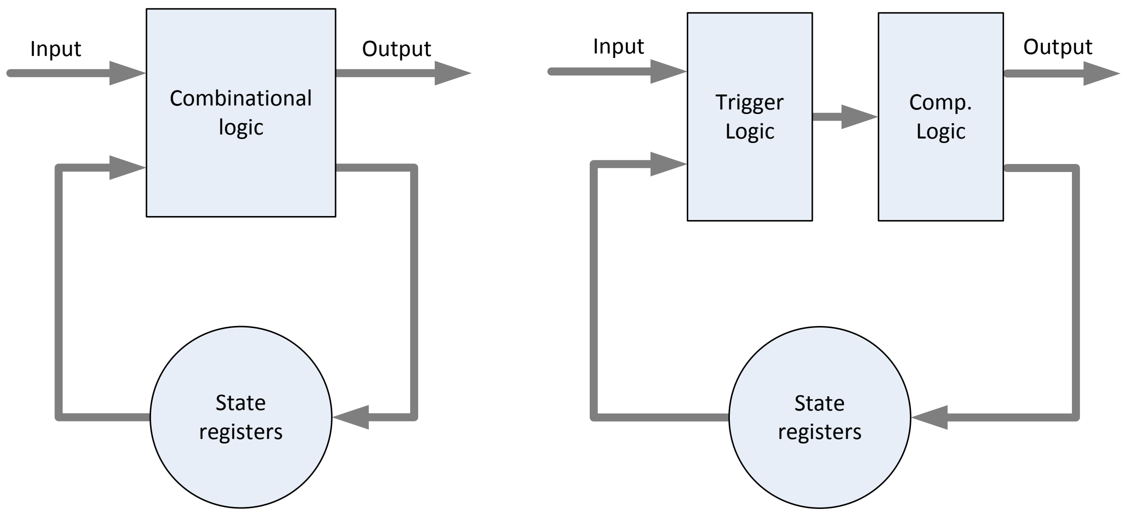
Finally, having the partial order $(\mathcal{G}, \leq)$ defined, we need to convert this partial order to a total order. That is done by means of topological sorting on guarded actions. We refer the reader to any standard text on computer algorithms like [93] to see how that can be done. We know that the dependency graph of guarded actions has no cycles due to the causality analysis done by Quartz compiler. Therefore topological sorting will succeed and result in a totally order list of DGAs. That list can be simply synthesized with a sequential iteration over its items.

Chapter 4

Results and Future Directions

4.1 Model of an elastic hardware module

We have implemented in this thesis a library to synthesize an elasticized synchronous system to a SysteMoC actor network. That enables rapid simulation of different partitioning strategies at a high level of abstraction. Metrics such as throughput can be effectively measured and compared between different partitioning strategies. The ultimate future goal of based on this work is to synthesize the partitioned synchronous system to actual hardware. That allows new optimization metrics to be measured e. g. area and power consumption. To this end, we try in this section to bridge the gap between the synthesized actor model and the hardware to be synthesized.



(a) A traditional hardware module (b) An elastic hardware module

FIGURE 4.1: A model of an elastic hardware module

Figure 4.1 depicts a model of an elastic hardware module and compares it to the traditional model of hardware modules. We adopt this model as our hardware synthesis target. As opposed to a traditional module that communicates with plain wires, an elastic module does input and output on *ports* using a communication protocol. The assumed protocol in our approach is a simple protocol that has forward (valid) and backward (back-pressure) channels e. g. SELF [59]. Note how the combinational logic of the traditional module has been replaced by Trigger Logic (TL) and Computation Logic (CL). In its simplest forms, TL should wait for all inputs ports to have valid values and all output ports to have buffer space before triggering the computation, which is exactly what we do in our actor model. Note that the output can be propagated to an elastic or a traditional module. Therefore, we omit output assumptions from the following discussion.

Despite being a simple model, it still have a significant potential not only from delay tolerance perspective but also from power saving one. Obviously, TL should always be powered-on since it should wait for valid inputs. However, it is also the right place to take fine-grained (per-computation) power saving decisions to control (1) clock-gating of state registers to implement *patience* and (2) power-gating of CL. Note that the usually larger CL part is required to be powered-on *only* when there is something useful to do. Power gating CL would save large amount of static power if computations are relatively rare. Actually, only the part of CL that is doing the particular computation relevant to the current reaction is needed. That part would execute the current actor action in our actor model.

Our elastic hardware module of Figure 4.1 beers similarity to LIP of Carloni et al. [53, 54]. The difference is that we do not think that elasticity can be handled efficiently using a black-box approach. We think that design decisions related to elasticity should be thought of from early design stages. Experience has show so far [94–96] that optimizations are a must for a viable elasticity approach. We discuss in the following possible optimizations to our basic model which helps in identifying directions for future work:

- **Early evaluation**: instead of waiting for a valid value on all input ports, TL may start a computation as soon as *sufficient* input is available to gain better throughput. Implementation requires specifying *mandatory* input ports per reaction i. e. input ports that must have *valid* value before starting early evaluation. After reading the *value* in those input ports, we can sequentially identify and wait for a valid value on other input ports (if required) that are sufficient for the current reaction to evaluate. For example, the key is the mandatory input for a simple If-Then-Else multiplexer. After reading the key, it is sufficient to wait for valid input on the selected port among the two ports. Implementing early evaluation requires having a state machine internal to the TL beside the module's state machine which complicates its design.

- **Speculation**: goes one step further than early evaluation in the quest for more throughput. TL doesn't wait for all *sufficient* inputs to arrive. Some inputs can be speculated (guessed) for the computation to start earlier. The cost of speculation can be significantly high in terms of hardware complexity especially for handling the case of miss-guessed inputs. If the miss-guessed input arrives late, all of the subsequent hardware modules need to be contacted to organize a fall-back.

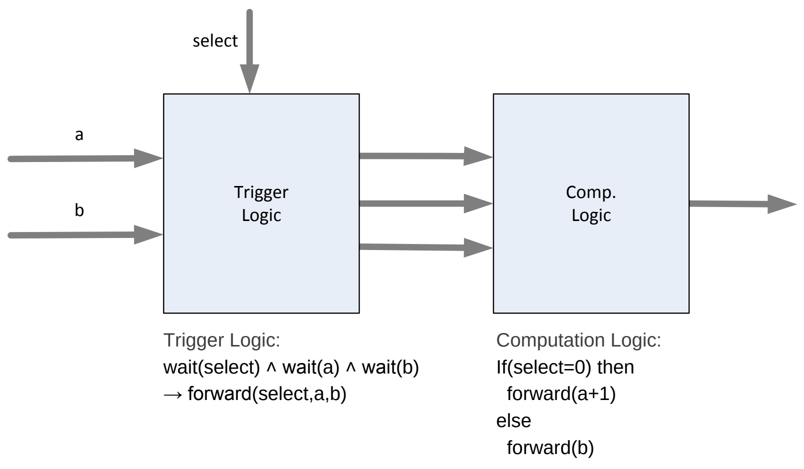
- **Scheduling**: if the interaction between two module can be analyzed, we will be able of identifying repetitive communication patterns and therefore schedule their communication. For example, by trading more buffering capacity on an elastic channel we might be able to remove the back-pressure communication mechanism since it won't be required anymore.

One little secret not mentioned before about the overhead of early evaluation is that it typically requires the elastic channel to support *anti-tokens* [63]. Anti-tokens are special type of tokens that are sent back on the input ports that were not involved in the reaction i. e. their input was not valid when early evaluation triggered. Anti-tokens should "clash" with valid input tokens to cancel each other. That is because the alignment of the original synchronous reaction should be preserved. Otherwise, the late coming input tokens could mistakenly be considered as fresh input. Obviously, supporting anti-tokens in the elastic channel adds more overhead to the already complicated TL design.

For elastic systems to be viable we need to achieve as much optimization as possible while keeping overhead to a minimum. That requires a synchronous design flow that starts from a semantically precise modeling language as opposed to HDL languages. A modeling language that provides as much insight as possible into the interaction between various module. Ideally, we should be able to send *only* what is actually needed for a computation to the other modules. Therefore, trigger part specification in early evaluation is simplified since anti-tokens won't be needed anymore. Unfortunately, synchronous languages with a single clock like Quartz [70] and Esterel [77] falls short on achieving that due to their MoC. An interesting future work would be to equip those languages with polychrony features similar to SIGNAL [79, 97].

We will continue with discussing trigger specification and design issues of elastic hardware modules. The discussion will be based on Figure 4.2 where we consider a simple ITE module that has the simple computation of adding 1 to its first input "`a`" while forwarding the second input intact "`b`". Figure 4.2(a) depicts the module with a trigger that waits on all inputs. Note that all inputs to the trigger need to be forwarded to the computation logic to do the computation.

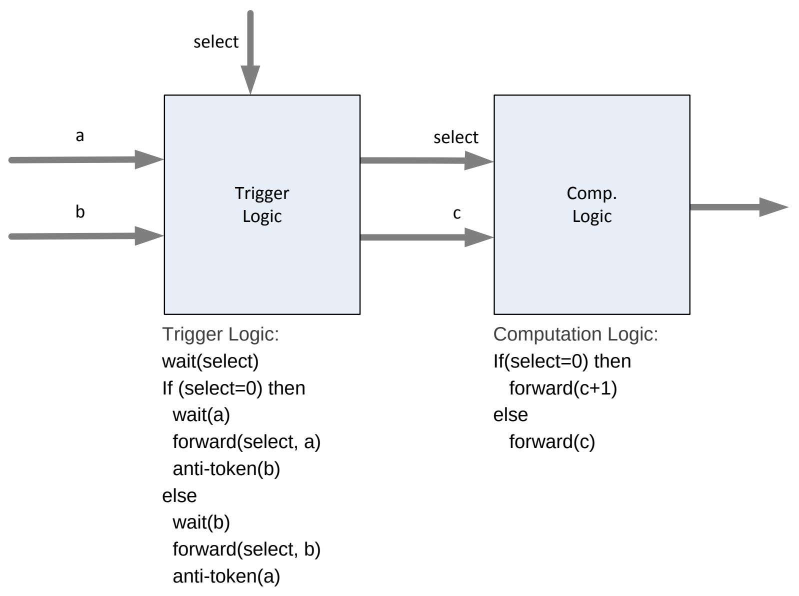
In Figure 4.2(b) the situation is a bit different. TL implements early evaluation and therefore doesn't need to wait on all inputs. Anti-token should be sent on the input port that wasn't involved in the reaction. We used a simple sequential language to specify trigger's functionality. Note that the computation logic doesn't need more than two inputs in this case. We have changed the design further in 4.2(c). There, TL also implement early evaluation. However, we used firing rules in 4.2(c) to specify essentially the same functionality of the trigger in 4.2(b).



```
Trigger Logic:
wait(select) ∧ wait(a) ∧ wait(b)
→ forward(select,a,b)
```

```
Computation Logic:
If(select=0) then
  forward(a+1)
else
  forward(b)
```

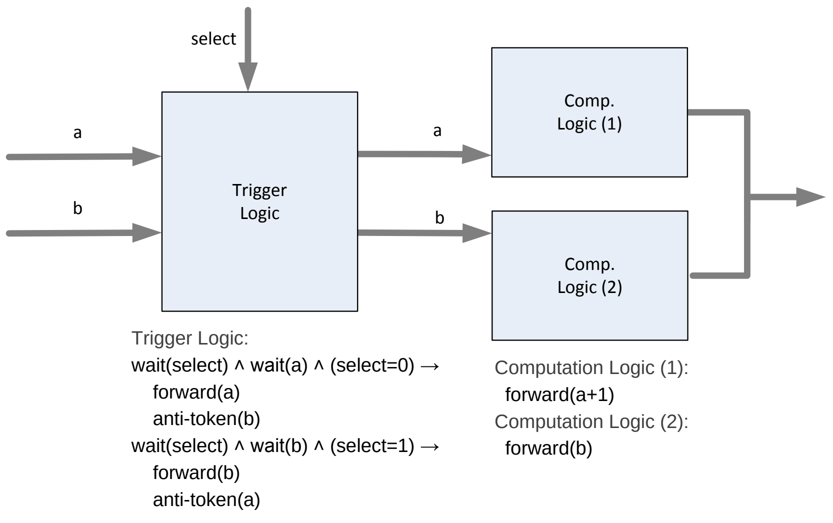
(a) Without early evaluation, waiting on all inputs



```
Trigger Logic:
wait(select)
If (select=0) then
  wait(a)
  forward(select, a)
  anti-token(b)
else
  wait(b)
  forward(select, b)
  anti-token(a)
```

```
Computation Logic:
If(select=0) then
  forward(c+1)
else
  forward(c)
```

(b) Early evaluation trigger specified using sequential function



```
Trigger Logic:
wait(select) ∧ wait(a) ∧ (select=0) →
   forward(a)
   anti-token(b)
wait(select) ∧ wait(b) ∧ (select=1) →
   forward(b)
   anti-token(a)
```

```
Computation Logic (1):
  forward(a+1)
Computation Logic (2):
  forward(b)
```

(c) Early evaluation trigger specified using firing rules

Figure 4.2: Elastic module design and trigger specification

Comparing the synthesizability and expressiveness of the two methods of trigger specification is considered as future work. We add to that the synthesis of trigger specification to hardware. Note also that computation logic in 4.2(c) has been separated to two blocks where trigger logic acts as a micro-network switch choosing the right computation logic based on input values. Dividing computation logic can have many applications in practice e. g. static power saving. Finally, note that by having better model analayzability through polychrony, we may eliminate the need for anti-tokens and can have better communication scheduling.

4.2 Supported features of Quartz

Our synthesis library supports the majority of the features of the Quartz language. We will begin our discussion of supported features with Quartz data types depicted in Table 4.1. We support all data types except arrays when defined in actor interface and unbounded bitvectors. Actually, it's questionable whether arrays should be ever supported as input/output ports of an actor. Working on large data structures should be typically done locally since communication is almost always more expensive than computation. Programmers of what will be elasticized models, should be aware of that and thus encouraged to use arrays only locally. Tuples can be used instead for small tightly coupled collection of items.

It should be noted that there are differences between array operations in Quartz and C++. Array variables should be defined first in Quartz and then they can be initialized with an initializer list. However, initializer lists in C++ can be used only at array definition. Additionally, Quartz allows assigning array variables to each other, e.g. `array1 = array2`, that would result in copying all the elements of the second to the first. As for unbounded bitvectors, it doesn't seem to us that they are useful in practice. Thus, they are not supported in our implementation. We continue our feature discussion in the following:

- **Type conversion operators**: supported type conversion operators are depicted in Table 4.2. Operator `tup2bv` for converting a tuple of booleans to bitvector is currently not supported and is left as future work.

- **Numeric and boolean operators**: all of them are supported including equality (==), inequality (! =), less than ($<$ and $\leq$), and greater than ($>$ and $\geq$). Boolean operators are supported including conjunction, disjunction, negation and implication.

- **Arithmetic operators**: all of them are supported both for integer arithmetic e. g. addition, subtraction, multiplication and modulo, and also for floating point arithmetic operators sinus ($sin(\pi)$), cosinus ($cos(\pi)$), power to an integer ($exp(\pi, \gamma)$), and logarithm to base 2 ($log_2(\pi)$).

- **Bitvictor operators**: the majority of those operators are supported including bit access, concatenation, reverse, and bit range access.

Beside those features, we make sure that the simulation doesn't violate the specification. Specification is violated for example when assigning to an element outside of array bound or when the value assigned to a bounded `nat`is outside its bounds. Specification violations are detected by inserting specification conditions as `assert` statement in synthesized actions. Additional `assert` statements are inserted to make sure that the action doesn't cause write conflicts. Write conflicts are detected by checking the value of the delayed assignment flag associated with the left-hande side variable. This value should always be false before executing an action.

Quartz data type	Synthesized data type	Remarks
`bool`	`bool`	direct synthesis of booleans.
`nat`	`unsigned int`	direct synthesis of naturals.
`nat{n}`	`unsigned int`	bound assertion is synthesized for every action with condition $(value < n)$.
`int`	`int`	direct synthesis of integers.
`int{n}`	`int`	bound assertion is synthesized for every action with condition $(-n \leq value < n\ )$.
`real`	`double`	direct synthesis of reals.
`bv`	N/A	unbounded bitvectors are not supported.
`bv{n}`	`sc_bv<n>`	Implementation is based on SystemC's template class `sc_bv`. An XML serialization function should also be overloaded.
$[n]\alpha$	`C++ array`	array of n elements of type α is supported for local variables only.
$\alpha_1 * \alpha_1 * ...\alpha_n$	`struct definition`	a wrapper `struct` is defined for every tuple. An XML serialization function should also be overloaded.

Table 4.1: Supported Quartz data types

Quartz operators	Remarks
`nat2bv(x,n)/int2bv(x,n)`	Supported directly by `sc_bv` constructor.
`arr2bv(x)`	Supported by converting the bool array to int.
`tup2bv`	Not supported.
`bv2nat(x)/bv2int(x)`	Supported directly by `sc_bv` class methods.
`nat2real(x)/int2real(x)`	Supported using C++ type casting.

Table 4.2: Supported type conversion operators

4.3 Limitations of considered approach

4.3.1 Limitations of perfect synchrony and Quartz

Obviously, we need to put our synthesis library "in action" in order validate its functionality. Since synchronous system partitioning is out of our scope in this work, we had to find a *manual* partitioning method that is simple to implement. The result of the partitioning should be a SysteMoC actor network graph that should *eventually* produce the same simulation output of the original synchronous system simulated with Averest's simulation tool `aif2trc`. However, we have discovered that there exist certain features in Quartz that are simply not synthesizable. Additionally, the synchronous model of computation mandates that individual synchronous components are generally not amenable to composition. We discuss those issues in the following:

- **Input of storage type event**: this feature of Quartz doesn't "make sense" in an elastic system that should presumably tolerate delay variations. That is due to the reaction to absence that needs to take place in case a valid input is not available. Reaction to absence is a *system-wide* decision where the synchronous system can decide at a given instance whether an event input is present or not. In our settings, synchronous components can't individually discriminate between an *absent* input and a *delayed* input. Workaround for this limitation is simple, just ignore it! We treat *event* input variables as if they were of type *memorized.* Therefore, actors should wait for all inputs to be available and no reaction to absence is synthesized. Note that our workaround is based on the assumption that (1) output and local variables of type event still have reaction to absence, which is the case in our synthesis process, and (2) the partitioning process is correct in the sense that an output port of type event will be connected to the considered input port of type event.

- **Input/Output variables**: Quartz supports interface variables of type I/O. That is, the variable value can be read as well as written to in a Quartz module. This feature is not straightforward to synthesize since SysteMoC actors can have ports either for input or output only. We need a transformation such that given an AIF component with I/O variables, each IO variable shall be replaced with two (or maybe more) variables. DGAs should be rewritten in terms of those variables in a way that preserves the original semantics. The case when DGAs only reads or only writes to an I/O variable is trivial to transform, however we didn't consider the general transformation in this thesis. Actually, we do not think that this problem can be generally solved by a transformation to the current *flattened* AIF system format. Instead, the compiler of Quartz is the right place to address this issue.

- **System partitioning**: any Quartz program of descent size should, obviously, be built by dividing functionality of the main module to several submodules. Synthesizing submodules to corresponding SysteMoC actors is a straightforward partitioning strategy. Unfortunately, the synchronous model of computation limits our ability to use such partitioning. Basically, each synchronous component does the cycle of waiting for valid value on *all* input ports, computing results, and writing output value to *all* output ports. The resulting actor network topology of this naive partitioning resembles a SDF [74] which is a restricted form of DPNs as discussed earlier.

Partitioning of a synchronous system to a set of synchronous components is more challenging than what it seems at a first glance. We will discuss in the following the partitioning of the synchronous system represented by module `Main` shown in Listing 4.1. The module itself is composed of two submodules running in parallel `M1` and `M2` shown in Listings 4.2 and 4.3 respectively.

```
module Main(event bool !out){
    event bool x;
    {M1(x);}
    ||
    {M2(x, out);}
}
```

Listing 4.1: `Main`

```
module M1(bool !x)
{
    x = true;
    pause;

}
```

Listing 4.2: `M1`

```
module M2( bool ?x
        , event bool !out){
  if (x) out=true; pause;
  if (x) out=true; pause;
  if (x) out=true; pause;
}
```

Listing 4.3: `M2`

Let's consider elasticizing module `Main` by partitioning it into its submodules. Each submodule in turn should be represented by an actor as depicted in Figure 4.3. We ignore delay on connection `x` and focus instead on the correctness of system execution in the first three cycles. The original synchronous system should produce values {`true`, `false`, `false`} on the output. That is because the value of event variable `x` can be resolved by the system to be `false` after `M1` has finished execution.

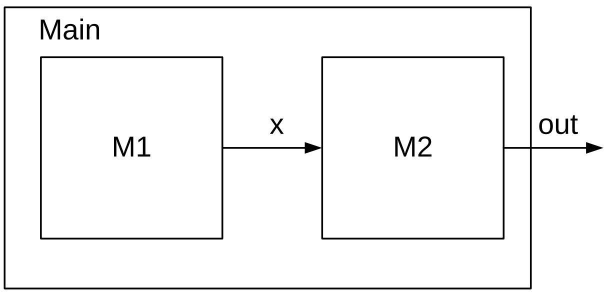


Figure 4.3: Composition of synchronous modules

That is not the case however in the elasticized system where actor M1 should finish execution after the first cycle. Therefore, only one output value is produced {true} and execution effectively terminates. Let's consider now modules M3 and M4 which are shown in Listings 4.4 and 4.5 respectively. Note that these modules run also in parallel like previous ones, however they have infinite lifetime which is typical for hardware modules. Note that M3 and M4 interact in a producer/consumer fashion where M3 set a value and signals flag start. In turn, M4 reads the value, act upon it, and then sets flag done.

```
module M3(event int ?computed, !value
        , event bool ?done, !start){
  value = 0;
  loop{
    next(start) = true;
    await(done);
    next(value) = computed;
  }
}
```

Listing 4.4: M3

```
module M4(event int ?value, !computed
        , event bool ?start, !done){

  loop{
    await(start);
    next(computed) = value + 1;
    next(done) = true;
  }
}
```

Listing 4.5: M4

Direct synthesis of each of those modules to an actor would preserve the original semantics. However, it is a bit inefficient since signal channels start and done are not required in that settings. Note that actors will only need the inputs value and computed to react. This example highlights an optimization opportunity to be considered when partitioning. To conclude, we have shown that system partitioning is an area where much future work is required. An aspect of this work should be on breaking perfect synchrony where all input and output ports are involved in every reaction. Polychrony [97] should allow a reaction to take place by reading values on a subset of inputs ports and producing values on a subset of output ports.

4.3.2 Limitations of SysteMoC

Limitations discussed so far are related to Quartz and the synchronous MoC. To this end, we would like to discuss also a SysteMoC issue that may arise in a future work that considers early evaluation. Early evaluation is a key technique for bringing synchronous systems to the average case performance of asynchronous ones. However, system specification in SysteMoC may need some improvements to better suite early evaluation simulation requirements. Basically, SysteMoC uses firing rules to describe trigger conditions. Specification of SysteMoC firing rules requires complete separation between *guard satisfaction* and *action execution*. In other words, a firing rule should specify necessary , and preferably sufficient, conditions for the firing action to execute.

That arrangement means that, firstly, firing rules and actions are tightly coupled and need to be considered combined for synthesis. The result is a simplified action the does only data transformations while much of the conditions on this transformation is transferred to the corresponding firing rule. Secondly, number of firing rules (and actions as a consequence) generated to handle different input combinations might increase exponentially. Thirdly, it forces us to separate what was a before a single firing action corresponding to a single synchronous reaction, to many sub-actions each matched with one or more firing rules.

To better illustrate the issue, let's consider an actor with input ports `a`, `b` and `c`. In our current settings, we didn't consider early evaluation and therefore the actor should wait for all inputs before firing any synchronous reaction. Now let's consider early evaluation for an action that we "know" before hand it involves `a` and `b` only. In that case, we will need one firing rule, and a corresponding action, for the case of only `a` and `b` have valid values and another firing rule and a firing action for the case when `a`, `b` and `c` have valid values. The former firing action should send an anti-token on port `c` and the later firing action should simply discard the value on port `c`.

The situation gets more complicated if we consider the more common case that a decision on what inputs are involved in a reaction can't be statically (compile-time) identified. That is, the decision depend on input values which is known only dynamically at run-time. If we consider our example again, that could mean that we need to know the value of `a` in order to determine if we will need `b` or `c` or maybe even both in our reaction. To adhere to SysteMoC's philosophy one would need many firing rule action pairs to handle all the different possibilities depending on the value of `a` and the presence of `b` and `c`.

The issue can be addressed by being a bit more *liberal*. That means that we do not need to specify a necessary and sufficient guard for a firing rule, but rather a guard that we think is sufficient to start executing firing action. Then, any other required inputs can be waited for *inside* the action. SysteMoC needs then to support a hybrid form of sequential waiting inside an action and firing rules (compare Figures 4.2(b) and 4.2(c)). It is understandable that waiting for inputs inside actions breaks SysteMoC's analayzability, since SysteMoC's developers depend on parsing FSM syntax for their analysis. However, if we consider using SysteMoC as a simulation platform and implement analysis at higher synchronous model level, then the idea can prove practical. Actually, supporting sequential waiting allow us to move swiftly from one end of the spectrum which is Kahn Process Networks [98] (an actor with a single state, no firing rules, and a single big action implemented with sequential waiting) to the other SysteMoC end of the spectrum (an actor with many firing rules and actions such that actions are small and implement data transformations only) with all the synthesis possibilities in between those extremes.

4.4 Experimentation settings

As part of this work, a synthesis library has been developed to generate SysteMoC actor network out of a partitioned synchronous system. Behavior of the actor network would be then simulated on a SystemC simulation kernel. The library has been developed in F#.NET which is a functional programming language based on the .NET framework. The decision was originally dictated by the fact that the API of Averest framework exposes F# special features such as *discriminated unions*. The use of those features is not possible from other .NET framework languages.

The library has about 3900 Lines of Code (LoC). It should be noted that learning and developing with F# was an enjoyable experience. Based on our previous experience with C#.NET, we can confirm that F# proved succinct and more natural in expressing concepts. It would have taken us much more effort in terms of LoC to achieve the same functionality in C# compared to F#.

The result of the synthesis is a single C++ file `synthesis.cpp`. File `synthesis.cpp` should be compiled and linked to SystemC and SysteMoC libraries among other libraries. The complete list of libraries required to link the executable successfully can be found in Table 4.3. Our test environment was an Ubuntu 12.04 virtual machine with `g++` version 4.6. Note that the libraries should be provided to the compiler in the given order, otherwise link may not be successful.

Library	Description
`libsystemoc`	SysteMoC library main library, found in systeMoC package.
`libboost_program_options`	One of Boost collection of libraries.
`libcosupport-systemc`	SysteMoC helper library, found in SysteMoC package.
`libsystemc`	SystemC main library
`libcosupport-tracing`	SysteMoC helper library, found in SysteMoC package.
`libcosupport-streams`	SysteMoC helper library, found in SysteMoC package.
`libboost_iostreams`	One of Boost collection of libraries.
`libboost_system`	One of Boost collection of libraries.
`libcosupport-smartptr`	SysteMoC helper library, found in SysteMoC package.
`libboost_thread`	One of Boost collection of libraries.
`libcosupport-math`	SysteMoC helper library, found in SysteMoC package.

Table 4.3: Libraries required to link `synthesis.cpp`

We will describe the procedure of obtaining libraries of Table 4.3 and set the build environment in the following:

- **SystemC library**: can be found at the download section of the Accellera website (http://www.accellera.org/home/). Downloaded source code needs to be configured and compiled. The compilation result in SystemC library and header files required for the compilation of our synthesized code.
- **SysteMoC library**: the official version of SysteMoC is available at (http://www12.informatik.uni-erlangen.de/research/scd/smoc_download.php). At the time of writing this thesis the official version was v0.9.3. Unfurtunately, this version doesn't support SyteMoC event registration with clocks. We had to use a development version that has been provided by SysteMoC team available at (http://www12.informatik.uni-erlangen.de/people/falk/systemoc-top--devel--1.0--20130618.tgz)
- **Boost libraries**: a collection of open source C++ libraries that provides variety of functionality. On our Ubuntu 12.04 machine, these libraries are available in the standard package repository.

4.5 Experimentation

The scope of this work was on the synthesis of an already elasticized synchronous system modeled in Quartz. To this end, a synthesis library has been developed. Our library supports the majority of Quartz features as discussed in section 4.2. To fully experiment with our synthesis library we had to look on ways to partition already existing Quartz benchmarks. Unfortunately, we haven't found existing work in the literature that addresses that issue. Moreover, the limitations discussed before in section 4.3.1 has left us with little room to maneuver, since almost all significant Quartz benchmarks have issues like IO variables. Transforming them to a form suitable for elasticization would require complete rewrite of the model.

Fortunately, the simulation of a Quartz module depends on writing a `drivenby` clause for it. Averest's simulation tool `aif2trc` uses that clause to provide input stimuli. We have exploited that in order to automatically partition a module/driver system to a SysteMoC network of two actors where the driver actor would provide input to the module actor. A tool named `aif2sysmoc` has been developed to do that automatic partitioning and feed the network topology to our library in order to generate the code. We have done many tests on small Quartz modules to validate our synthesis features. The output has been checked to match with the output of `aif2trc`.

The tests has been done on our own examples in addition to suitable examples already available at http://www.averest.org/examples/. To push the library to its limits, we have successfully used it to generate code for a SHA2-256 [99] core that we have developed in the previous semester. The core source and documentation is also available online at http://es.cs.uni-kl.de/research/applications/sha2/. The core has been developed in a basic version that matches the standard and in an optimized version. Both require sophisticated bitvictor operations.

	Gen. time	Gen. LoC	Boolexps	module states
Heron sqr root	0.1 s	462	11	3
CruiseControl	1 s	1266	335	11
SHA (Unoptimized)	3 s	5076	553	60
SHA (Optimized)	12 s	38012	7301	163

Table 4.4: Experimenting with `aif2sysmoc`

The representative result of the experimentations is given in Table 4.4. We list the example name along side the time required to generate code for it, the generated lines of code, the number of boolean expressions manipulated, and the total number of states in the example. The number of boolean expression has been listed since its the most expensive operation in code generation. Remember that building EFSM requires doing many case discrimination operations on boolean guards.

4.6 Conclusion

We discussed challenges faced by the synchronous paradigm in the modern nano-era and argued that it is inherently incapable of being adapted to address those challenges. Asynchronous design is a promising approach that could address those challenges. However, asynchronous design is a disruptive technology to established practices and design expertise. Additionally, it suffers itself from a different set of design challenges. To this end, elasticizing a synchronous design can provide an *evolutionary* path to adapt synchronous circuits to better tolerate delay variability.

In that regard, we have discussed the main approaches to synchronous elasticity, namely, *LIP* [53, 54] and *elastic circuits* [50, 51, 58, 59]. In LIP, it has been proposed to wrap IP modules (pearls) with communication wrappers (shells) that enables them to communicate using ports using a latency-insensitive protocol. LIP should be supported mainly by the place & route tool. In the later approach, elasticity has been addressed at a fine-grained RTL level. RTL transformations have been proposed in [51, 61] to a synthesized gate-netlist. These transformations should be considered in addition to well-known RTL transformations e. g. retiming [24]. This approach requires considerable support from the synthesis tool and therefore is more disruptive to the former one.

We argued that a elasticity should be addressed at a higher level of abstraction compared to those approaches. Basically, it should be considered as another motivation for the migration to model-based design where software as well as hardware design issues, including elasticity, of complex embedded systems can be analyzed early at system specification. Basically, we need a modeling methodology that enables synthesizing communication based on the required computation behavior e. g. removing feedback from channels ”known” not to get congested. To this end , the synchronous approach [68, 69] to system modeling has an appropriate formal foundation to be considered as the starting point for such a methodology.

Additionally, we have described the pros and cons of pure synchronous languages like Quartz. These issue can be partly addressed by extending pure synchronous languages with polychronous modeling features [97]. Nonetheless, research is still required in the area of partitioning synchronous specifications among. In a very recent survey on reliability in nano-era circuits by Henkel et al. [100], though it doesn’t claim to be complete, elastic systems has not even been mentioned, that gives an indication that system elasticity is still at its infancy and that much work is still required.

Finally, our discussion have included a novel classification model for elastic system in section 1.6. Also, we have set some future direction based on a model of an elastic hardware module in section 4.1. We have validated our approach by developing a library to synthesize synchronous components to a SysteMoC actor network and simulate it using SystemC. In our library, we have handled the issues arising from semantic mismatch between the two different MoCs in a seamless way.

Bibliography

[1] A. Davis and S. M. Nowick, “An introduction to asynchronous circuit design,” Tech. Rep. UUCS-97-013, University of Utah, Dept. of Computer Science, Sept. 1997.

[2] D. Matzke, “Will physical scalability sabotage performance gains?,” *Computer Magazine*, vol. 30, no. 9, pp. 37–39, 1997.

[3] E. Friedman, “Clock distribution networks in synchronous digital integrated circuits,” *Proceedings of the IEEE*, vol. 89, pp. 665–692, May 2001.

[4] L. Carloni and A. Sangiovanni-Vincentelli, “Coping with Latency in SoC Design,” *Special Issue on Systems on a Chip*, vol. 22, no. 5, pp. 24 – 35, 2002.

[5] T. Chaney and C. Molnar, “Anomalous Behavior of Synchronizer and Arbiter Circuits,” *IEEE Transactions on Computers*, vol. C-22, pp. 421–422, Apr. 1973.

[6] S. Yang and M. Greenstreet, “Computing Synchronizer Failure Probabilities,” in *2007 Design, Automation & Test in Europe Conference & Exhibition*, pp. 1–6, IEEE, Apr. 2007.

[7] J. Gubner and C.-P. Chen, “Correlation-preserved non-Gaussian statistical timing analysis with quadratic timing model,” in *Proceedings. 42nd Design Automation Conference, 2005.*, pp. 83–88, IEEE, 2005.

[8] M. Orshansky and K. Keutzer, “A general probabilistic framework for worst case timing analysis,” in *Proceedings 2002 Design Automation Conference (IEEE Cat. No.02CH37324)*, pp. 556–561, ACM, 2002.

[9] P. A. Beerel, J. Cortadella, and A. Kondratyev, “Bridging the gap between asynchronous design and designers (Tutorial),” in *17th International Conference on VLSI Design. Proceedings.*, pp. 18–20, IEEE Comput. Soc, 2004.

[10] M. Imai and T. Nanya, “A Novel Design Method for Asynchronous Bundled-data Transfer Circuits Considering Characteristics of Delay Variations,” in *12th IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC’06)*, pp. 68–77, IEEE, 2006.

[11] K. Bowman, S. Duvall, and J. Meindl, “Impact of die-to-die and within-die parameter fluctuations on the maximum clock frequency distribution for gigascale integration,” *IEEE Journal of Solid-State Circuits*, vol. 37, no. 2, pp. 183–190, 2002.

[12] L. Benini, E. Macii, M. Poncino, and G. De Micheli, “Telescopic units: a new paradigm for performance optimization of VLSI designs,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 17, pp. 220–232, Mar. 1998.

[13] L. Benini, E. Macii, and M. Poncino, “Telescopic Units: Increasing The Average Throughput Pipelined Designs By Adaptive Latency Control,” in *Proceedings of the*

34th Design Automation Conference, pp. 22–27, IEEE, 1997.

[14] D. Baneres, J. Cortadella, and M. Kishinevsky, "Variable-latency design by function speculation," in *2009 Design, Automation & Test in Europe Conference & Exhibition*, pp. 1704–1709, IEEE, Apr. 2009.

[15] R. Ho, K. Mai, and M. Horowitz, "The future of wires," *Proceedings of the IEEE*, vol. 89, pp. 490–504, Apr. 2001.

[16] P. Saxena, N. Menezes, P. Cocchini, and D. Kirkpatrick, "Repeater Scaling and Its Impact on CAD," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 23, pp. 451–463, Apr. 2004.

[17] D. Sylvester and K. Keutzer, "A global wiring paradigm for deep submicron design," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 19, no. 2, pp. 242–252, 2000.

[18] L. Benini and G. De Micheli, "Networks on chips: a new SoC paradigm," *Computer*, vol. 35, no. 1, pp. 70–78, 2002.

[19] W. Dally and B. Towles, "Route packets, not wires: on-chip interconnection networks," in *Design Automation Conference, 2001. Proceedings*, pp. 684–689, 2001.

[20] R. Marculescu, U. Y. Ogras, L.-s. Peh, N. E. Jerger, and Y. Hoskote, "Outstanding Research Problems in NoC Design: System, Microarchitecture, and Circuit Perspectives," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 28, pp. 3–21, Jan. 2009.

[21] V. Varshavsky and V. Marakhovsky, "GALA (Globally AsynchronousLocally Arbitrary) Design," in *Concurrency and Hardware Design* (J. Cortadella, A. Yakovlev, and G. Rozenberg, eds.), pp. 61–107, Springer Berlin Heidelberg, 2002.

[22] C. J. Myers, *Asynchronous circuit design*, vol. 6. John Wiley & Sons, Inc, 2001.

[23] J. Sparsøand S. Furber, "Principles of Asynchronous Circuit Design: A Systems Perspective," *Kluwer Academic Publishers*, 2001.

[24] C. E. Leiserson and J. B. Saxe, "Retiming synchronous circuitry," *Algorithmica*, vol. 6, pp. 5–35, June 1991.

[25] D. E. Muller and S. W. Bartky, "A Theory of Asynchronous Circuits," in *International Symposium of the Theory of Switching*, pp. 204–243, 1959.

[26] D. A. Huffman, "The synthesis of sequential switching circuits," Tech. Rep. 274, Research Laboratory of Electronics, Massachusetts Institute of Technology, Cambridge, Massachusetts, 1954.

[27] H. C. Brearley, "ILLIAC II: A short description and annotated bibliography," *IEEE Transactions on Computers*, vol. C-14, no. 6, pp. 399–403, 1965.

[28] W. A. Clark, "Macromodular computer systems," in *Proceedings of the April 18-20, 1967, spring joint computer conference on - AFIPS '67 (Spring)*, (New York, New York, USA), p. 335, ACM Press, Apr. 1967.

[29] C. L. Seitz, "System timing," in *An introduction to VLSI design*, ch. 7, Reading, MA: Addison-Wesley, 1980.

[30] A. Peeters, "The 'Asynchronous' Bibliography."

[31] J. Brzozowski and J. Ebergen, "On the delay-sensitivity of gate networks," *IEEE Transactions on Computers*, vol. 41, no. 11, pp. 1349–1360, 1992.

[32] I. E. Sutherland, "Micropipelines," *Communications of the ACM*, vol. 32, pp. 720–738, June 1989.

[33] A. Smirnov and A. Taubin, "Synthesizing asynchronous micropipelines with design compiler," in *SNUG'06: Synopsys User Group, Boston*, (Boston, USA), 2006.

[34] D. Edwards and A. Bardsley, "Balsa: An Asynchronous Hardware Synthesis Language," *The Computer Journal*, vol. 45, pp. 12–18, Jan. 2002.

[35] D. M. Chapiro, *Globally-Asynchronous Locally-Synchronous Systems.* Phd thesis, Standford University, Oct. 1984.

[36] J. Muttersbach, T. Villiger, and W. Fichtner, "Practical design of globally-asynchronous locally-synchronous systems," in *Proceedings Sixth International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC 2000) (Cat. No. PR00586)*, pp. 52–59, IEEE Comput. Soc, 2000.

[37] J. Muttersbach, T. Villiger, H. Kaeslin, N. Felber, and W. Fichtner, "Globally-asynchronous locally-synchronous architectures to simplify the design of on-chip systems," in *Twelfth Annual IEEE International ASIC/SOC Conference (Cat. No.99TH8454)*, pp. 317–321, IEEE, 1999.

[38] F. Gurkaynak, S. Oetiker, H. Kaeslin, N. Felber, and W. Fichtner, "GALS at ETH Zurich: Success or Failure," in *12th IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC'06)*, pp. 150–159, IEEE, 2006.

[39] T. Chelcea and S. M. Nowick, "Robust interfaces for mixed-timing systems with application to latency-insensitive protocols," in *Proceedings of the 38th conference on Design automation - DAC '01*, (New York, New York, USA), pp. 21–26, ACM Press, June 2001.

[40] T. Chelcea and S. Nowick, "Robust interfaces for mixed-timing systems," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 12, pp. 857–873, Aug. 2004.

[41] I. Sutherland and S. Fairbanks, "GasP: A Minimal FIFO Control," in *Proceedings of the 7th International Symposium on Asynchronous Circuits and Systems*, (Washington, DC, USA), pp. 46–, Mar. 2001.

[42] S. Tatapudi and J. Delgado-Frias, "A mesochronous pipelining scheme for high-performance digital systems," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 53, pp. 1078–1088, May 2006.

[43] A. Branover, R. Kol, and R. Ginosar, "Asynchronous design by conversion: converting synchronous circuits into asynchronous ones," in *Proceedings Design, Automation and Test in Europe Conference and Exhibition*, pp. 870–875, IEEE Comput. Soc, Feb. 2004.

[44] G. Hazari, M. Desai, A. Gupta, and S. Chakraborty, "A novel technique towards eliminating the global clock in VLSI circuits," in *17th International Conference on VLSI Design. Proceedings.*, pp. 565–570, IEEE Comput. Soc, 2004.

[45] N. Andrikos, L. Lavagno, D. Pandini, and C. Sotiriou, "A Fully-Automated Desynchronization Flow for Synchronous Circuits," in *Design Automation Conference, 2007. DAC '07. 44th ACM/IEEE*, pp. 982–985, 2007.

[46] K. Fant and S. Brandt, "NULL Convention Logic: a complete and consistent logic for asynchronous digital circuit synthesis," in *Proceedings of International Conference on Application Specific Systems, Architectures and Processors: ASAP '96*, pp. 261–273, IEEE Computer Soc. Press, 1996.

[47] I. Blunno, J. Cortadella, A. Kondratyev, L. Lavagno, K. Lwin, and C. Sotiriou, "Handshake protocols for de-synchronization," in *10th International Symposium on Asynchronous Circuits and Systems, 2004. Proceedings.*, pp. 149–158, IEEE, 2004.

[48] J. Cortadella, A. Kondratyev, L. Lavagno, K. Lwin, and C. Sotiriou, "From synchronous to asynchronous: an automatic approach," in *Proceedings Design, Automation and Test in Europe Conference and Exhibition*, pp. 1368–1369, IEEE Comput. Soc, 2004.

[49] L. Necchi, L. Lavagno, D. Pandini, and L. Vanzago, "An ultra-low energy asynchronous processor for Wireless Sensor Networks," in *12th IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC'06)*, pp. 78–85, IEEE, 2006.

[50] J. Carmona, J. Cortadella, M. Kishinevsky, and A. Taubin, "Elastic Circuits," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 28, pp. 1437–1455, Oct. 2009.

[51] J. Cortadella, M. Galceran-Oms, and M. Kishinevsky, "Elastic systems," in *Eighth ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE 2010)*, pp. 149–158, IEEE, July 2010.

[52] G. A. Agha, I. A. Mason, S. F. Smith, and C. L. Talcott, "A foundation for actor computation," *Journal of Functional Programming*, vol. 7, pp. 1–72, Jan. 1997.

[53] L. Carloni, K. McMillan, and A. Sangiovanni-Vincentelli, "Theory of latency-insensitive design," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 20, no. 9, pp. 1059–1076, 2001.

[54] K. L. M. Luca P. Carloni, "Latency Insensitive Protocols," in *Computer Aided Verification* (N. Halbwachs and D. Peled, eds.), vol. 1633, pp. 123–133, Springer Berlin Heidelberg, 1999.

[55] L. P. Carloni, "The Role of Back-Pressure in Implementing Latency-Insensitive Systems," *Electronic Notes in Theoretical Computer Science*, vol. 146, pp. 61–80, Jan. 2006.

[56] M. Singh and M. Theobald, "Generalized latency-insensitive systems for single-clock and multi-clock architectures," in *Proceedings Design, Automation and Test in Europe Conference and Exhibition*, pp. 1008–1013, IEEE Comput. Soc, 2004.

[57] M. R. Casu and L. Macchiarulo, "A new approach to latency insensitive design," in *Proceedings of the 41st annual conference on Design automation - DAC '04*, (New York, New York, USA), p. 576, ACM Press, 2004.

[58] J. Cortadella, M. Kishinevsky, and B. Grundmann, "Synthesis of synchronous elastic architectures," in *2006 43rd ACM/IEEE Design Automation Conference*, pp. 657–662, IEEE, 2006.

[59] J. Cortadella, "Self: Specification and design of synchronous elastic circuits," in *TAU 06: Proceedings of the ACM/IEEE International Workshop on Timing Issues*, 2006.

[60] T. Kam, M. Kishinevsky, J. Cortadella, and M. Galceran-Oms, "Correct-by-construction microarchitectural pipelining," in *Proceedings of the 2008 IEEE/ACM*

International Conference on Computer-Aided Design, (San Jose, California), pp. 434–441, IEEE Press, Nov. 2008.

[61] M. Galceran-Oms, J. Cortadella, D. Bufistov, and M. Kishinevsky, "Automatic microarchitectural pipelining," in *Proceedings of the Conference on Design, Automation and Test in Europe*, pp. 961–964, European Design and Automation Association, Mar. 2010.

[62] M. Galceran-Oms, A. Gotmanov, J. Cortadella, and M. Kishinevsky, "Microarchitectural Transformations Using Elasticity," *ACM Journal on Emerging Technologies in Computing Systems*, vol. 7, pp. 1–24, Dec. 2011.

[63] J. Cortadella and M. Kishinevsky, "Synchronous Elastic Circuits with Early Evaluation and Token Counterflow," in *Design Automation Conference, 2007. DAC '07. 44th ACM/IEEE*, pp. 416–419, IEEE Press, 2007.

[64] D. E. Bufistov, J. Cortadella, M. Galceran-Oms, J. Júlvez, and M. Kishinevsky, "Retiming and recycling for elastic systems with early evaluation," in *Proceedings of the 46th Annual Design Automation Conference on ZZZ - DAC '09*, (New York, New York, USA), p. 288, ACM Press, July 2009.

[65] J. Carmona, J. Julvez, J. Cortadella, and M. Kishinevsky, "Scheduling Synchronous Elastic Designs," in *2009 Ninth International Conference on Application of Concurrency to System Design*, pp. 52–59, IEEE, July 2009.

[66] D. Bertozzi, L. Benini, and G. De Micheli, "Error control schemes for on-chip communication links: the energy-reliability tradeoff," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 24, pp. 818–831, June 2005.

[67] D. Flynn, "AMBA: enabling reusable on-chip designs," *IEEE Micro*, vol. 17, no. 4, pp. 20–27, 1997.

[68] N. Halbwachs, *Synchronous Programming of Reactive Systems.* Kluwer Academic Publisher, 1993.

[69] A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone, "The synchronous languages 12 years later," *Proceedings of the IEEE*, vol. 91, pp. 64–83, Jan. 2003.

[70] K. Schneider, "The synchronous programming language Quartz," Tech. Rep. 375, Department of Computer Science, University of Kaiserslautern, Kaiserslautern, Germany, 2009.

[71] J. Falk, C. Haubelt, and J. Teich, "Efficient Representation and Simulation of Model-Based Designs in SystemC," in *Proceedings of Forum on Specication and Design Languages 2006 (FDL 2006)*, Proc. FDL\\\'06, Forum on Design Languages 2006, (Darmstadt, Germany), pp. 129–134, ECSI, Sept. 2006.

[72] J. Falk, C. Haubelt, and J. Teich, "Representing Models of Computation in SystemC," in *ITG/GI/GMM Workshop für Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen*, (Munich, Germany), Apr. 2005.

[73] J. Teich, "Hardware/Software Codesign: The Past, the Present, and Predicting the Future," *Proceedings of the IEEE*, vol. 100, pp. 1411–1430, May 2012.

[74] E. Lee and D. Messerschmitt, "Synchronous data flow," *Proceedings of the IEEE*, vol. 75, no. 9, pp. 1235–1245, 1987.

[75] J. Eker, J. Janneck, E. Lee, J. Ludvig, S. Neuendorffer, and S. Sachs, "Taming heterogeneity - the Ptolemy approach," *Proceedings of the IEEE*, vol. 91, pp. 127–144, Jan. 2003.

[76] D. Harel and A. Pnueli, "On the development of reactive systems," in *Logics and models of concurrent systems* (K. R. Apt, ed.), pp. 477–498, New York, NY, USA: Springer-Verlag New York, Inc., Feb. 1985.

[77] G. Berry, "The foundations of Esterel," in *Proof, language, and interaction: essays in honour of Robin Milner* (G. Plotkin, C. Stirling, and M. Tofte, eds.), pp. 425–454, Cambridge, MA, USA: MIT Press, July 1998.

[78] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice, "LUSTRE: a declarative language for real-time programming," in *Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages - POPL '87*, (New York, New York, USA), pp. 178–188, ACM Press, Oct. 1987.

[79] A. Benveniste, P. Le Guernic, and C. Jacquemot, "Synchronous programming with events and relations: the SIGNAL language and its semantics," *Science of Computer Programming*, vol. 16, pp. 103–149, Sept. 1991.

[80] K. M. Chandy and J. Misra, *Parallel program design: a foundation.* Austin, Texas, USA: Addison-Wesley, 1988.

[81] E. A. Lee, S. Neuendorffer, and M. J. Wirthlin, "Actor-Oriented Design Of Embedded Hardware And Software Systems," *Journal of Circuits, Systems and Computers*, vol. 12, pp. 231–260, 2003.

[82] Committee, *IEEE Standard 1666-2011 for SystemC Language Reference Manual.* New Jersey, USA: IEEE Standards Association, 2012.

[83] H. D. Patel and S. K. Shukla, "Towards a heterogeneous simulation kernel for system level models," in *Proceedins of the 14th ACM Great Lakes symposium on VLSI - GLSVLSI '04*, (New York, New York, USA), p. 248, ACM Press, Apr. 2004.

[84] J. Falk, C. Haubelt, and J. Teich, "Syntax and execution behavior of SysteMoC," Tech. Rep. 04-2005, University of Erlangen-Nuremberg, Department of CS 12, Hardware-Software-Co-Design, Am Weichselgarten 3, D-91058 Erlangen, Germany, Dec. 2005.

[85] C. Haubelt, M. Meredith, T. Schlichter, and J. Keinert, "SystemCoDesigner: Automatic Design Space Exploration and Rapid Prototyping from Behavioral Models," in *Proceedings of the 45th Design Automation Conference (DAC 08)*, (Anaheim, CA, USA), pp. 580–585, June 2008.

[86] S. S. Bhattacharyya, G. Brebner, J. W. Janneck, J. Eker, C. von Platen, M. Mattavelli, and M. Raulet, "OpenDF: a dataflow toolset for reconfigurable hardware and multicore systems," *ACM SIGARCH Computer Architecture News*, vol. 36, p. 29, June 2009.

[87] D. Baudisch, J. Brandt, and K. Schneider, "Translating Synchronous Systems to Data-Flow Process Networks," in *Parallel and Distributed Computing, Applications and Technologies (PDCAT)* (S.-S. Yeo, B. Vaidya, and G. A. Papadopoulos, eds.), (Gwangju, Korea), pp. 354–361, IEEE Computer Society, 2011.

[88] Y. Bai, J. Brandt, and K. Schneider, "SMT-based optimization for synchronous programs," in *Proceedings of the 14th International Workshop on Software and Compilers for Embedded Systems - SCOPES '11*, (New York, New York, USA), p. 11, ACM Press,

June 2011.

[89] Y. Bai, J. Brandt, and K. Schneider, "Data-Flow Analysis of Extended Finite State Machines," in *2011 Eleventh International Conference on Application of Concurrency to System Design*, pp. 163–172, IEEE, June 2011.

[90] J. C. Hoe and Arvind, "Synthesis of operation-centric hardware descriptions," pp. 511–519, Nov. 2000.

[91] M. Vijayaraghavan and Arvind, "Bounded dataflow networks and latency-insensitive circuits," pp. 171–180, July 2009.

[92] J. Brandt, M. Gemuende, and K. Schneider, "From Synchronous Guarded Actions to SystemC," in *Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen (MBMV)* (M. Dietrich, ed.), (Dresden, Germany), pp. 187–196, Fraunhofer Verlag, 2010.

[93] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, Second Edition*. The MIT Press, 2001.

[94] M. Casu and L. Macchiarulo, "Issues in implementing latency insensitive protocols," in *Proceedings Design, Automation and Test in Europe Conference and Exhibition*, pp. 1390–1391, IEEE Comput. Soc, 2004.

[95] R. Lu and C.-K. Koh, "Performance optimization of latency insensitive systems through buffer queue sizing of communication channels," in *International Conference on Computer Aided Design. ICCAD-2003*, pp. 227–231, 2003.

[96] E. Kilada, S. Das, and K. Stevens, "Synchronous elasticization: Considerations for correct implementation and MiniMIPS case study," in *2010 18th IEEE/IFIP International Conference on VLSI and System-on-Chip*, pp. 7–12, IEEE, Sept. 2010.

[97] P. Le Guernic, J.-p. Talpin, and J.-c. Le Lann, "Polychrony for System Design," *Journal of Circuits, Systems and Computers*, vol. 12, no. 3, pp. 261–304, 2003.

[98] G. Kahn, "The Semantics of Simple Language for Parallel Programming," in *Proceedings of IFIP Congress* (J. L. Rosenfeld, ed.), pp. 471–475, 1974.

[99] National Institute of Standards and Technology, "Federal Information Processing Standards (FIPS) Publication 180-4," 2012.

[100] J. Henkel, L. Bauer, N. Dutt, P. Gupta, S. Nassif, M. Shafique, M. Tahoori, and N. Wehn, "Reliable on-chip systems in the nano-era," in *Proceedings of the 50th Annual Design Automation Conference on - DAC '13*, (New York, New York, USA), p. 1, ACM Press, May 2013.