# Synthesis of Distributed Synchronous Specifications to SysteMoC

Mohamed Ammar Ben Khadra
TU Kaiserslautern
ammar@rhrk.uni-kl.de

Yu Bai
TU Kaiserslautern
bai@cs.uni-kl.de

Klaus Schneider
TU Kaiserslautern
schneider@cs.uni-kl.de

**Abstract**

Heterogeneous multi-core embedded systems are being increasingly used to meet performance requirements of modern applications. However, meeting the often stringent demands for correctness, resource utilization, and real-time response in a complex concurrent software is challenging. Model-based design is a widely accepted methodology to meet such requirements where a *functional* model of the system is developed independently from its *architectural* model. In that regard, functional models developed in synchronous languages have been successfully used for synthesis of hardware circuits and sequential software. However, significant work on distributed software synthesis from synchronous languages remains only theoretical. We discuss a semantics-preserving synthesis procedure for elastic networks (a common representation of distributed synchronous specifications) to SysteMoC which is an actor-oriented modeling library based on SystemC. Hence, we not only demonstrate a potential simulation platform that can be used to improve the existing theory, but we also make a step towards a *formal* model-based design flow that combines synchronous and actor-oriented Models of Computation (MoC). Additionally, we discuss some experimental results based on our implementation of the synthesis procedure.

## 1. Introduction

Meeting the ever increasing demands for more processing power in many embedded systems is challenging. Compared to traditional software, concerns of embedded software designers go beyond developing concurrent software that can take advantage of the available hardware parallelism. Actually, the real challenge lies in meeting non-functional system requirements including correctness, resource utilization, and real-time response. To this end, model-based design is regarded by many as a promising approach to meet such stringent requirements where a functional model of the system is developed independently from its target architecture. That allows designers to focus on core functional issues without being distracted by architecture specific issues. That also enables re-using the same functional model across different target architectures. After validating the functional model, designers proceed with the synthesis phase where the design space is explored to find

a suitable software mapping and scheduling strategy. Additionally, hardware synthesis should be considered too for a comprehensive model-based design approach.

We identify some key properties that need to be satisfied in an ideal functional model. (P1) formal verifiability which is mandatory for safety-critical applications. However, it is also increasingly important for other applications due to software complexity. (P2) composability where the model can be analyzed and composed in a modular manner from existing models. (P3) analyzability for early estimation of non-functional system properties. (P4) expressiveness in a wide range of application domains including data-dominated as well as control-dominated applications. Additionally, hybrid system modeling is increasingly important with prevalence of cyber-physical systems. Finally, (P5) synthesizability such that the generated software is efficient, semantics-preserving, and satisfies the non-functional requirements of the system. Unfortunately, improving P5 mandates improving P3. In turn, that typically comes at the cost of reducing support for one or more of the other properties. For example, adhering to a restricted MoC such as static dataflow networks [LM87] improves P3 at the cost of reducing model expressiveness in P4. Those limitations have been acknowledged in projects like Ptolemy II [EJL+03] which focused on formal composition of multiple MoCs to achieve higher expressiveness.

Synchronous languages [BCE+03] have emerged since early eighties to address the specification issues 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 w.r.t. internal system state. All micro-steps are executed in the same variable environment i. e. a variable can take only one value in a macro-step. Micro-steps are assumed to consume no time, and time advances to the next macro-step after all micro-steps are finished. They offer key advantages over traditional programming languages including deterministic concurrency and explicit modeling of time. Time is essential for improved analyzability [Lee09] and it can be used for, e. g. , WCET analysis [LS03]. Moreover, they enable designers to associate different computation costs with different macro-steps through clock-refinement [GBS13]. Hence, early estimations of resource utilization can be conducted.

Simplicity of the synchronous hypothesis provides easier reasoning about system behavior. Additionally, the sound formal basis and lock-step composition of synchronous modules enable more efficient formal verification since asynchronous concurrent behaviors should not be considered. Hence, synchronous languages already satisfy properties P1 and P2 to a large extent and provide many interesting properties for P3. However, their main application domain was memory-bounded control-dominated applications. That is demonstrated with commercial successes for Esterel [Ber98] and Lustre [HCRP91] in efficient synthesis of real-time uni-processor software as well as hardware. In that respect, work on distributed software synthesis, also known as desynchronization, for multicore processors and distributed environments like automotive software, is still an active area of research.

All events in a synchronous model are synchronized and totally ordered w.r.t a global clock and communication takes zero-time. Obviously, that is an ideal assumption that cannot be efficiently realized in practice. Hence, many desynchronization approaches have been proposed in the literature to relax perfect synchrony while preserving the original semantics (see [Gir05] for a good survey). Among the proposed techniques are endo/isochronous distribution [BCL00] and efficient communication through polychrony [LTL03]. In that regard, there is a lack of suitable simulation tools that help in exercising and improving that body of theoretical knowledge.

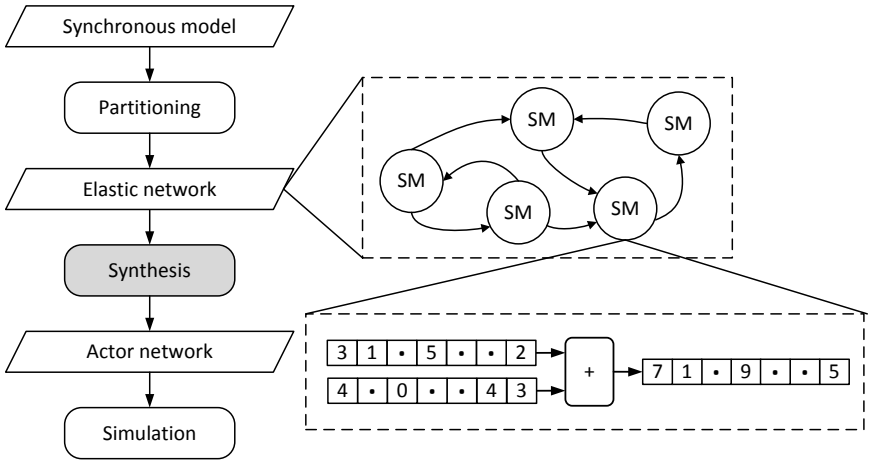



**Figure 1:** Our simulation flow where the considered synthesis procedure is highlighted. The communication model of synchronous modules (SM) using synchronous channels (SC) is emphasized.

We discuss in this paper a semantics-preserving synthesis procedure for elastic networks, an abstract distributed synchronous specification that consists of synchronous modules (SM) that communicate over synchronous channels (SC). Elastic networks are the first refinement step of perfect synchrony where computations are still synchronized while communication over SC takes time in the form of an integer number of clock cycles. The term has been adopted from elastic circuits [CCKT09] since they both share the same 'view' of the system. Hence, elastic network optimizations can be directly transferred to their hardware counterparts. We avoid the term synchronous data-flow since it is often confused with static dataflow networks [LM87]. Formally,

**Definition 1.** *A **synchronous module** is a tuple $(\mathcal{P}, \mathcal{L}, \mathcal{CF}, \mathcal{DF})$ where $\mathcal{P}$ 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 module. $\mathcal{CF}$ and $\mathcal{DF}$ define the list of control-flow and data-flow guarded actions respectively.*

**Definition 2.** *A **synchronous channel** is a tuple $(sP, dP, \mathcal{D}, \mathcal{C}, \mathcal{T})$ where $sP$ and $dP$ are SM that are the source and destination port, respectively. $\mathcal{D}$ is minimum token delay. $\mathcal{C}$ is the maximum capacity of the channel. $\mathcal{T}$ is the supported token type which can be basic, e. g. integer, or aggregate.*

We used guarded actions to characterize SMs. Basically, a guarded actions consists of an action, usually an assignment operation, that is not executed until its associated boolean guard is satisfied. More on guarded actions has been provided in Section 2. An SC is a FIFO capable of holding tokens (valid data) and bubbles (empty). At each clock tick, a token can advance iff the item in the next FIFO cell is a bubble. That creates back-pressure on producing SM to stop if a consuming SM has stalled. $\mathcal{D}$ determines the minimum number of clock cycles needed by a token through an SC. That number may increase if a receiving SM stalls. Additionally, an SM cannot fire until there are tokens on **all** of its input ports and bubbles on **all** of its output ports (patience property). Upon firing, exactly one token is consumed/produced on every input/output port.

The entire simulation flow is depicted in Figure 1 where we obtain our elastic network representation from a partitioning stage that is beyond our scope. We focus in this work on the synthesis stage of elastic networks only. Our synthesis target is SysteMoC [FHT06], a SystemC based

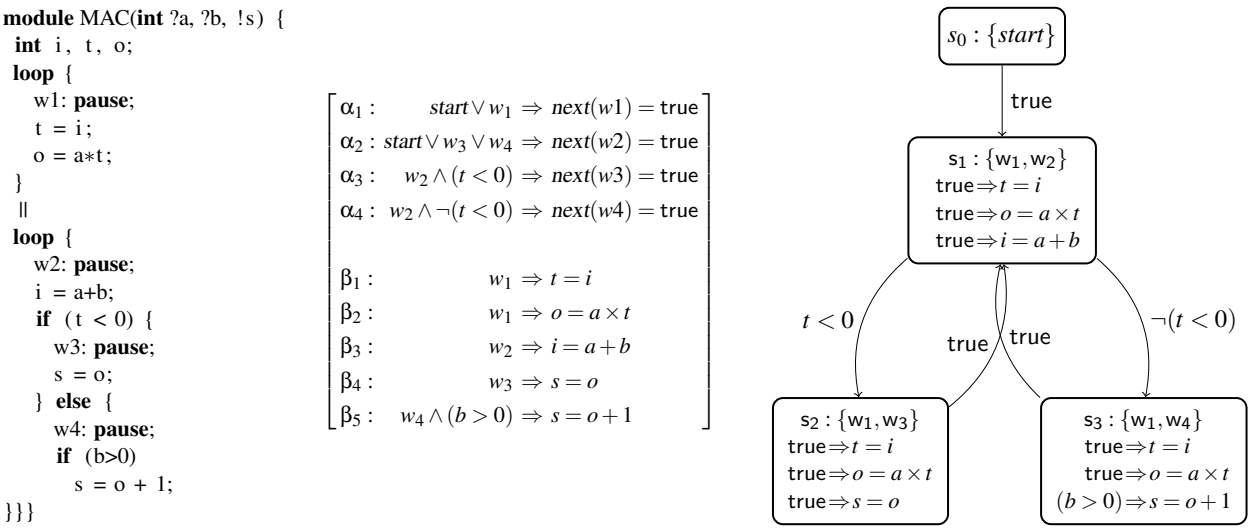



**Figure 2:** (left) Quartz module MAC, (middle) compiled guarded actions where $\mathcal{CF}$ ($\alpha_i$) and $\mathcal{DF}$ ($\beta_i$), and (right) generated EFSM.

actor-oriented modeling library. SysteMoC provides both time-triggered and asynchronous data-triggered actor firing options. Hence, it enables gradual refinement of a synchronous model from perfect synchrony to a completely asynchronous actor network model. Our discussion of the synthesis procedure is organized as follows: First, we give in Section 2 the necessary background. Then, we discuss the details of synthesizing SMs and SCs in Section 3. Later, some experimental results based on our implementation are described in Section 4. Finally, we discuss some related work in Section 5.

## 2. Background

We give in this section the necessary background on synchronous guarded actions and later on actor-oriented modeling using SysteMoC.

### 2.1. Synchronous guarded actions

Figure 2 depicts an example synchronous module MAC written in the imperative synchronous language Quartz [Sch09] which is the specification language of our Averest framework[1]. Module MAC has input variables **a** and **b**, output variable **s**, and local variables **i**, **t**, **o**. Macro-steps are determined using **pause** statements which has been assigned *labels* to show the control-flow of the module. Basically, MAC consists of two infinite loops running in parallel. However, they are running in lock-steps synchronizing at each **pause**. Quartz syntax of MAC is shown only to provide an intuitive description of how a synchronous model would look like. Compiling MAC using Averest's Quartz compiler would yield a set of control-flow guarded actions (CGA) and data-flow guarded actions (DGA).

[1] Available at `http://www.averest.org`

A guarded action (GA) has the form $\langle \gamma \Rightarrow \alpha \rangle$, where guard $\gamma$ is a boolean expression that needs to be satisfied in order for action $\alpha$ to be executed. We are mostly concerned with assignment actions. An *immediate assignment*, denoted by $\langle x = e \rangle$, assigns $x$ to the evaluated result of expression $e$ in the current macro-step. A *delayed assignment* denoted by $\langle \mathit{next}(x) = e \rangle$, assigns $x$ to the value of $e$ in the current macro-step. However, the assignment is executed in the next macro-step. Note that most synchronous languages can be compiled to guarded actions which makes our characterization of SM behavior in terms of set of CGAs ($\mathcal{CF}$) and set of DGAs ($\mathcal{DF}$) a common intermediate format. CGAs differ from DGAs in that their actions are assignments to control-flow *labels* whereas actions of DGAs assign values to module variables only.

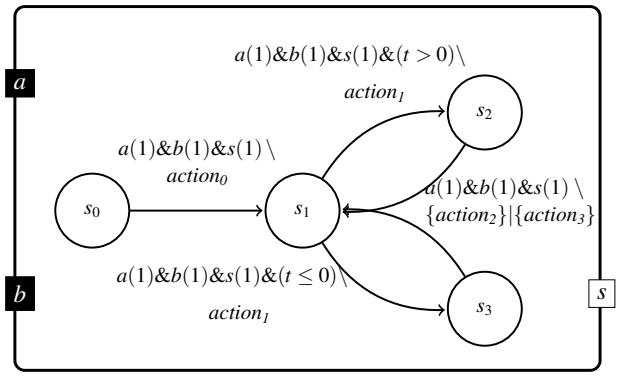
One can represent the behavior of synchronous system with a state machine that has only a single state with $\mathcal{DF}$ attached to it. At each clock tick, all guards $\gamma$ of DGAs in $\mathcal{DF}$ are evaluated. Then, only the assignments $\alpha$ that have their guards $\gamma$ evaluated to **`true`** will be executed. However, we are interested in a more efficient representation, in terms of computation, of the system by only evaluating DGAs that belong to the current synchronous reaction. To this end, we generate an Extended Finite State Machine (EFSM) from the given $\mathcal{CF}$ of the system. Then, each DGA is attached only to the state(s) where it could possibly be executed. An EFSM is formally defined in Definition 3. Figure 2 depicts $\mathcal{CF}$, $\mathcal{DF}$ and the generated EFSM of module MAC. The default starting state is $s_0$ where only label $start$ is set. Each state has been annotated with its control flow labels and its attached DGAs. Note that each state represents a synchronous reaction where all system variables should have a unique value. A *reaction to absence* should take place for variables that have not been assigned a value by DGAs of the current synchronous reaction. In that respect, variables in Quartz can be either of type *memorized* or type *event* depending on their required reaction to absence behavior. *Memorized* variables are assigned their same value in the previous reaction, whereas *event* variables take the default value for their data type e. g. **false** for booleans.

**Definition 3.** *An **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 $T \subseteq (S \times G \times S)$ is a finite set of transition relations where $G$ is the set of transition guards. $D$ 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 $s$.*

### 2.2. Actor-oriented modeling in SysteMoC

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 supports the same token type. The connection is a FIFO that has a configurable size. Actor's internal states are defined by a set of local variables $\mathcal{L}'$ that are readable and writable only inside the actor. The behavior of the actor is defined by a Firing-FSM (FFSM). Formally,

**Definition 4.** *An **actor network graph** is a directed bipartite graph $(A, C, P, E)$ 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 $n \in N^{\infty} = \{1, 2, 3, \ldots \infty\}$, 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 constrained 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.*




```
class MAC : public smoc_actor {
    SC_HAS_PROCESS(MAC);
  smoc_firing_state s0, s1, s2, s3;
  // Other definitions omitted.
public:
  smoc_port_in<int> a, b; smoc_port_out<int> s;
  MAC(sc_module_name name)
    : // Constructor instantiations omitted.
  {
  // Only firing transitions of s1 are shown.
    s1=TILL(clk)>>(a(1)&&b(1)&&(guard1))>>s(1)
        >>CALL(MAC::action1)>>s2|
        TILL(clk)>>(a(1)&&b(1)&&(guard2))>>s(1)
        >>CALL(MAC::action1)>>s3;
 } };
```

**Figure 3:** MAC actor model, (left) a graphical model where each $action_i$ executes DGAs corresponding to $s_i$, patience property is observed by guarding all firing transitions such that **one** token (bubble) should be available on all input (output) ports before firing, e. g. a(1). (right) textual C++ code generated for the actor where **guard1** and **guard2** are functions $(t > 0)$ and $(t \leq 0)$ respectively.

**Definition 5.** *An **actor** is a tuple $(\mathcal{P}, \mathcal{L}', \mathcal{F}, \mathcal{R})$ where $\mathcal{P}$ 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 actor, $\mathcal{F}$ is a set of functions, and $\mathcal{R}$ is the firing FSM.*

Note that $\mathcal{F} = \mathcal{F}_G \cup \mathcal{F}_A$ where $\mathcal{F}_G$ is a set of boolean functions used to guard state transitions, and $\mathcal{F}_A$ is set of firing actions executed upon firing and able to change values of $\mathcal{L}'$. Note also that an EFSM is a restricted form of an FFSM where the relation between FSM states and $\mathcal{F}_A$ is bijective. Therefore, we can describe the behavior of actors using EFSM. Transition guards of an EFSM are described by $G \subseteq \mathcal{G}_{clk} \times \mathcal{G}_I \times \mathcal{G}_O \times \mathcal{F}_G$, where $\mathcal{G}_I$ ($\mathcal{G}_O$) are used to guard that sufficient number of tokens (bubbles) are available on input (output) ports to be consumed (produced), and $\mathcal{G}_{clk}$ is a clock event condition used if clock synchronization is required. Figure 3 depicts the actor model and the generated code for module MAC. Clocks in SysteMoC are defined per actor. Therefore, transitions can be synchronized by setting an equal clock period (main clock) for all actors in the network and setting all firing transitions to have guard $\mathcal{G}_{clk}$. It is simple to model actors that need multi-cycles for their computation by setting their clock to an integer number of the main clock period. It is even possible to configure delay time for individual firing transitions by using Virtual Processing Component (VPC) which is a supporting framework to SysteMoC.

## 3. Synthesis details

We discuss in this section the synthesis details of an SM and an SC separately.

### 3.1. Synthesis of a Synchronous Module

Given an SM defined by $(\mathcal{P}, \mathcal{L}, \mathcal{CF}, \mathcal{DF})$, we need to generate its corresponding SysteMoC actor $(\mathcal{P}, \mathcal{L}', \mathcal{F}, \mathcal{R})$. To this end, we need to generate and synthesize the EFSM $(\mathcal{R})$ based on the given $(\mathcal{CF}, \mathcal{DF})$. Additionally, we synthesize $\mathcal{F} = \mathcal{F}_G \cup \mathcal{F}_A$ where each firing action in $\mathcal{F}_A$ is generated from DGAs of a corresponding state in $\mathcal{R}$, while $\mathcal{F}_G$ are generated from transition guards of $\mathcal{R}$.

Finally, actor variables have to be synthesized such that $\mathcal{L}' = \mathcal{L} \cup \mathcal{L}_E$ where $\mathcal{L}_E$ are extra variables required to handle the semantic mismatch between the two different MoCs e. g. output variables are writable only in actors whereas they are both readable and writable in synchronous guarded actions. Details of SM synthesis have been briefly discussed here. We refer to [Ben13] for complete details including code generation templates.

Our EFSM generation algorithm proceeds by evaluating (to a boolean canonical form) all guards of $\mathcal{CF}$ based on the label variable environment of current state. Then, a subset $\mathcal{CF}'$ is defined such that its guards didn't evaluate to **false** in the previous step. Later, $\mathcal{CF}'$ goes through *case discrimination* to identify the variable environment of each of the next state(s) to be visited. The algorithm starts with the label environment of the initial state, i. e. , only label *start* is assigned value **true**. Attaching DGAs to a state is done by evaluating all guards of $\mathcal{DF}$ based on the state's label environment and attaching the corresponding subset $\mathcal{DF}'$ with guards that were not reduced to **false**. As for actor variable $\mathcal{L}'$, we provide in Table 1 a summary of the generated variables. The rationale behind that will get clearer next in our discussion of firing action generation.

**Table 1:** Synthesizing variables of a synchronous module.

| Variable flow | Variable type | Generated variables |
|---|---|---|
| Input | Memorized | Port variable only. |
| | Event | Port variable only. |
| Output | Memorized | Port variable and local carry variable . |
| | Event | Port variable, a local carry variable, and a flag variable. |
| Local | Memorized | A local direct variable, a local carry variable, and a flag variable. |
| | Event | A local direct variable, a local carry variable, and a flag variable. |

An actor firing action proceeds in two phases, namely, immediate and delayed. In the immediate phase, (1) the reaction to absence for all variables of type *event* are executed and flags of delayed assignments are checked to be executed. To this end, all local variables and output variables of type *event* should have a boolean flag that is set when a delayed value is available from previous reaction to be assigned. (2) immediate guarded actions are ordered and executed, (3) by this time, all variable values in the current reaction are known and output values can be propagated on ports. The delayed phase can now proceed where (1) DGAs writing to local variables are executed first. However they are rewritten such that writing is done to a *carry* variable and not to the actual *direct* variable. That is because the current value of local variables might be used by transition guard functions $\mathcal{F}_G$ to determine the next transition. Finally, (2) DGAs writing to output variables are ordered and executed.

**Definition 6.** ***Guarded action dependencies****: let $G = \langle \gamma \Rightarrow x = \tau \rangle$ be a guarded action where $\gamma$ is the boolean guard, $x$ is the variable assigned a value, and $\tau$ is an expression. Let $FV(\tau)$ be the set of free variables in expression $\tau$. We define the following:*

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

**Definition 7.** ***Guarded actions order****: for guarded actions $G1$ and $G2$, we say that $G1 < G2$ iff $wrVars(G1) \subseteq rdVars(G2)$.*

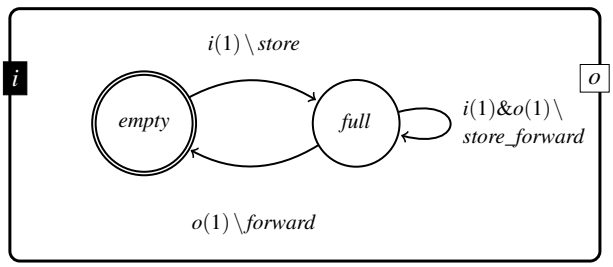



**Figure 4:** Actor model of a Channel Buffer (CB). Channel buffers are chained with smoc_fifo to build a synchronous channel where each CB contributes to a delay of one clock cycle.

We provide a partial order relation on guarded actions in Definitions 6 and 7. Ordering of guarded action in immediate and delayed phases requires finding a total (sequential) order for DGAs based on the analyzed partial order. In that regard, a topological sorting is done such that for each two DGAs where $G_1 < G_2$: $G_1$ should appear before $G_2$ in the immediate phase (RAW dependency) and $G_1$ should appear after $G_2$ in the delayed phase (WAR dependency). Consider for example DGAs of state $s_1$ in Figure 2. DGA (RAW) ordering in the immediate phase should be $\beta_3$, $\beta_1$, and $\beta_2$. However, no delayed assignments exist in $s_1$. Therefore, the delayed phase shall be empty in its firing action.

### 3.2. Synthesis of a Synchronous Channel

We now consider the synthesis of an SC defined by tuple $(sP, dP, \mathcal{D}, \mathcal{C})$. Note that it is important for us to make SMs completely independent of their communication over their SCs. In that way, we can reuse generated SM code and simulate them with different SC configuration of $\mathcal{D}$ and $\mathcal{C}$. Unfortunately, SysteMoC supports only one FIFO type for communication between actors which is defined in the class **`smoc_fifo`**. A **`smoc_fifo`** can be considered as an SC that has zero delay i. e. $\mathcal{D} = 0$. Therefore, we had to simulate the clock cycle delay on an SC by introducing Channel Buffers (CB). Basically, a CB is a basic SysteMoC actor that has the sole purpose of delaying its input by one clock cycle. Therefore, to model an SC with delay of $\mathcal{D}$ clock cycles, we need to generate a chain of $\mathcal{D}$ number of CBs connected by **`smoc_fifo`**.

The actor model of a CB is depicted in Figure 4. Required storage capacity $\mathcal{C}$ of the SC can be distributed among **`smoc_fifo`** in the chain. Our arrangement separates SC delay $\mathcal{D}$ from SC capacity $\mathcal{C}$ which are implemented by CB and **`smoc_fifo`** respectively. Note that an CB is essentially a single C++ template class in the generated source code. Hence, the C++ compiler would instantiate as many classes as needed depending on the declared SC token type $\mathcal{T}$. Note that it is also possible to implement different $\mathcal{D}$ and $\mathcal{C}$ of an SC using a single complex CB instead of our chain of simple CBs. However, the C++ compiler will then need to instantiate a different C++ class for each different combination of $\mathcal{D}$, $\mathcal{C}$ and $\mathcal{T}$ which may result in a larger executable. Note also that asynchronous communication can be modeled using a single CB that has its clock period set to arbitrary time period rather than to the main clock period as in synchronous channels.

## 4. Experimental results

We have developed a synthesis library that is capable of generating code for most features of Quartz including all of its data types. The library was developed in F#.NET and it has about 3900 Lines of Code (LoC). We took advantage of the data types already supported by SystemC, e. g. , bitvectors (**`sc_bv`**) were elegantly mapped to their Quartz counterparts. Synthesis of aggregate data types (tuples) was also straightforward by mapping them to C++ structs. We also supported **`assert`** statement generation to make sure that the original specifications are not violated at run-time, e. g. , out-of-bound array accesses. We consider here a simple partitioning strategy where the synchronous system is partitioned to two modules, namely, one that provides input stimuli (driver) and another that reacts based on that input (main). Therefore, the resulting actor network consists of two actors representing two SMs. Example results of the experimentations conducted on different benchmarks is given in Table 2. We list the example alongside, the time required to generate code for it on a standard PC, the effective number of LoC, the number of canonicalized boolean expressions during EFSM generation, and the total number of states in the EFSM. Number of canonicalized boolean expressions is listed since its the most expensive operation in terms of required computation.

**Table 2:** Experimental results

| Model | Time | LoC | Boolexps | States |
|---|---|---|---|---|
| Heron Sqr Root | 0.1 s | 462 | 11 | 3 |
| Cruise Control | 1 s | 1266 | 335 | 11 |
| SHA (Basic) | 3 s | 5076 | 553 | 60 |
| SHA (Optimized) | 12 s | 38012 | 7301 | 163 |

The tested benchmarks were Heron (Newton) square root algorithm, a simple car cruise control model, and an implementation of the SHA2-256 hashing algorithm. Thanks to the robust support for bitvectors in SystemC, we have not had any problem in synthesizing the SHA2 model, although it utilizes some sophisticated bitvector operations. SHA2 model was implemented in two versions, a basic version that maps the standard directly and uses all 64 scheduling values $W_t$, and an optimized version that starts hashing a block as soon as a hash word has been received. It uses the last 16 scheduling values only which is typical for commercial cores. The optimized version has more states since it requires more control. The generated code is then compiled using g++ and linked against SysteMoC, SystemC and some Boost libraries. Note that generated LoC depends on the number of states in the EFSM and the number of DGAs attached to each state. Comparing basic and optimized SHA2 implementations reveals that LoC figure grows rapidly due to increased number of DGAs per state. That issue should be handled by a smarter scheme for DGA code sharing between different firing actions. We leave that for a future work.

## 5. Related work

Brandt et al. [BGS10] considered the synthesis of a non-partitioned synchronous system represented by guarded actions to SystemC directly. In contrast, we discussed the synthesis of distributed synchronous specifications to SysteMoC. Halbwachs et al. [HM06] discussed a method

for simulating asynchronous tasks in Lustre. Their implementation was based on SCADE, which is the commercial Lustre tool. This work is concerned instead with the refinement and simulation of synchronous models using open source libraries. Generally, there is not much work in the area of synchronous language synthesis for simulation purposes. We note that it is important to preserve the control states of SMs for better analyzability. Hence, system partitioning schemes that convert a synchronous model directly to a asynchronous one, e. g. Baudisch et al. [BBS10], are not suitable for elastic network representation.

## 6. Conclusion

We discussed in this work a semantics-preserving synthesis procedure for elastic networks. That enables a large body of theoretical work on synchronous system distribution, e. g. endo/isochrony, to be exercised and improved. It also makes it possible to gradually refine a synchronous model from perfect synchrony to a completely asynchronous actor network model. Additionally, analysis developed for actor networks like [FZK+10] can be utilized in the refined synchronous model. We think that our synthesis procedure is a step towards combining both MoCs in a formal model-based design flow that can potentially have better analyzability and expressiveness. For example, designers can benefit from the composability of synchronous models and combine it with the mapping and scheduling properties of data-flow actor models to generate safety-critical multi-processor software.

## Acknowledgment

We would like to thank Joachim Falk of the HW/SW Co-Design group at Erlangen-Nuremberg University for providing us with SysteMoC 1.0 beta version. This version supports clock synchronization and it is still not publicly released to the time of this writing.

## References

[BBS10] Baudisch, D., J. Brandt, and K. Schneider: *Multithreaded code from synchronous programs: Extracting independent threads for OpenMP*. In *Design, Automation and Test in Europe (DATE)*, pages 949–952, Dresden, Germany, 2010. EDA Consortium.

[BCE+03] Benveniste, A., P. Caspi, S. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone: *The synchronous languages twelve years later*. Proceedings of the IEEE, 91(1):64–83, 2003.

[BCL00] Benveniste, A., B. Caillaud, and P. Le Guernic: *Compositionality in dataflow synchronous languages: Specification and distributed code generation*. Information and Computation, 163(1):125–171, 2000.

[Ben13] Ben Khadra, M.A.: *A model-based approach to sychronous elastic systems*. Master's thesis, Department of Computer Science, University of Kaiserslautern, Germany, March 2013. Master.

[Ber98] Berry, G.: *The foundations of Esterel*. In Plotkin, G., C. Stirling, and M. Tofte (editors): *Proof, Language and Interaction: Essays in Honour of Robin Milner*, pages 425–454. MIT Press, 1998.

[BGS10] Brandt, J., M. Gemünde, and K. Schneider: *From synchronous guarded actions to SystemC*. In Dietrich, M. (editor): *Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen (MBMV)*, pages 187–196, Dresden, Germany, 2010. Fraunhofer Verlag.

[CCKT09] Carmona, J., J. Cortadella, M. Kishinevsky, and A. Taubin: *Elastic circuits*. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (T-CAD), 28(10):1437–1455, October 2009.

[EJL+03] Eker, J., J.W. Janneck, E.A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong: *Taming heterogeneity – the Ptolemy approach*. Proceedings of the IEEE, 91(1):127–144, January 2003.

[FHT06] Falk, J., C. Haubelt, and J. Teich: *Efficient representation and simulation of model-based designs*. In *Forum on Specification and Design Languages (FDL)*, pages 129–135, Darmstadt, Germany, 2006. Electronic Chips and Systems Design Initiative (ECSI).

[FZK+10] Falk, J., C. Zebelein, J. Keinert, C. Haubelt, J. Teich, and S.S. Bhattacharyya: *Analysis of SystemC actor networks for efficient synthesis*. ACM Transactions on Embedded Computing Systems (TECS), 10(2):18:1–18:34, December 2010.

[GBS13] Gemünde, M., J. Brandt, and K. Schneider: *Clock refinement in imperative synchronous languages*. EURASIP Journal on Embedded Systems, 3:1–21, August 2013. http://jes.eurasipjournals.com/content/2013/1/3.

[Gir05] Girault, A.: *A survey of automatic distribution method for synchronous programs*. In *Synchronous Languages, Applications, and Programming (SLAP)*, pages 1–20, Edinburgh, Scotland, UK, 2005. unpublished workshop proceedings.

[HCRP91] Halbwachs, N., P. Caspi, P. Raymond, and D. Pilaud: *The synchronous dataflow programming language LUSTRE*. Proceedings of the IEEE, 79(9):1305–1320, September 1991.

[HM06] Halbwachs, N. and L. Mandel: *Simulation and verification of asynchronous systems by means of a synchronous model*. In *Application of Concurrency to System Design (ACSD)*, pages 3–14, Turku, Finland, 2006. IEEE Computer Society.

[Lee09] Lee, E.A.: *Computing needs time*. Communications of the ACM (CACM), 52(5):70–79, May 2009.

[LM87] Lee, E.A. and D.G. Messerschmitt: *Synchronous data flow*. Proceedings of the IEEE, 75(9):1235–1245, September 1987.

[LS03] Logothetis, G. and K. Schneider: *Exact high level WCET analysis of synchronous programs by symbolic state space exploration*. In *Design, Automation and Test in Europe (DATE)*, pages 10196–10203, Munich, Germany, 2003. IEEE Computer Society.

[LTL03] Le Guernic, P., J. P. Talpin, and J. C. Le Lann: *Polychrony for system design*. Journal of Circuits, Systems, and Computers (JCSC), 12(3):261–304, June 2003.

[Sch09] Schneider, K.: *The synchronous programming language Quartz*. Internal report 375, Department of Computer Science, University of Kaiserslautern, Kaiserslautern, Germany, December 2009.