A Model-based Design Flow for Asynchronous Implementations from Synchronous Specifications

Yu Bai
Hebei University of Science annd Technology
Shijiazhuang, China
baiyu@hebust.edu.cn

Omair Rafique and Klaus Schneider
Department of Computer Science,
University of Kaiserslautern, Germany
{rafique,schneider}@cs.uni-kl.de

***Abstract*—The synthesis of distributed embedded systems from dataflow models like Kahn Process Networks (KPN) has to deal with particular problems like absence of deadlocks and buffer overflows. However, the verification of the absence of these problems for a KPN model is in general not decidable. Starting with synchronous models, desynchronization avoids such design difficulties by generating sound dataflow networks by correctness of construction. In this paper, we present a design flow following such an approach. Our design flow differs from previous work in the following aspects: The synchronous models are specified by an imperative synchronous language and are therefore better suited for control-intensive applications. Verification of desynchronization criteria is carried out efficiently with the help of model checking and SAT-solving, ensuring the compliance of the functional behavior. Qualified code is translated automatically into the KPN model. Finally, the KPN model is automatically synthesized to the open computing language (OpenCL) based implementation which is platform independent and can be executed on various commercial off-the-shelf target platforms.**

I. Introduction

A. Motivation

Synchronous models are deterministic and therefore provide in particular predictable temporal behaviors, which greatly simplifies many efforts in the verification and validation of hard real-time systems. They have proven their usefulness on both single-core and multi-core platforms in safety critical applications such as aviation [1] and other embedded system industries. However, when it comes to soft real-time applications such as streaming and signal processing [2], performance and design flexibility are dominant factors over safety, and commercial off-the-shelf (COTS) heterogeneous hardware platforms are preferred from both the perspective of marketing as well as performance. Over-synchronization caused by the synchronous semantics puts an unnecessary burden and is criticized for these applications. In IoT, embedded devices are moreover connected by various means of communication. Synchronization could exist at different levels of abstraction, but a global synchronization remains a great challenge [3].

For designing distributed applications, asynchronous dataflow models following KPN semantics are usually preferred. However, KPN suffers from undecidability of checking major correctness properties like boundedness of buffers and absence of deadlocks [4]. As an alternative, *desynchronization* of synchronous models [5], [6] has been recently developed that benefit both from the static analysis methods for synchronous systems and the performance of the finally synthesized asynchronous systems. These approaches start with a network of synchronous components which are triggered by the same clocks. Desynchronization means to remove the clocks so that the components have to decide on their own by the arrival of input data whether they can execute a new execution step. In general, it is known that this might introduce new unwanted behaviors for synchronous models which is not the case for the subclass of endochronous components (in an isochronous network). Hence, if synchronous components were verified for endochrony, one can simply generate one thread per component to obtain a multi-threaded asynchronous system that is completely driven by the arrival of input data. Desynchronization is beneficial for distributed system synthesis in the sense that starting from a synchronous model, system correctness can be easily verified, and a KPN can be generated while its correctness is maintained [7].

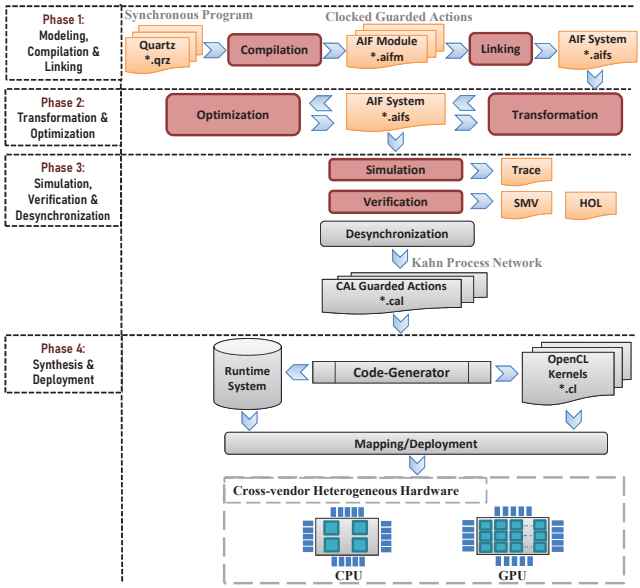

Fig. 1. The proposed design flow.

In this paper, we present a complete model-based design flow based on the *Averest* [8] tool to enable the design and automatic software synthesis of embedded systems by desyn-

978-3-9819263-5-4/DATE21/©2021 EDAA

chronization. The proposed design flow is organized in four phases as shown in Fig. 1. In particular, our design flow is based on the synchronous programming language Quartz [8] which is a variant of Esterel [9]. In phase 1, we specify the system behavior in Quartz which can then be compiled to the Averest intermediate format (AIF). In phase 2, with AIF we have the freedom and convenience of code optimization and transformations such as dead and passive code elimination. We then verify the desynchronization properties along with other properties of interest in phase 3. After successful verification, AIF code is translated into a KPN model. The underlying language of the target KPN is CAL [10]. In phase 4, the CAL KPN is finally synthesized using the open computing language (OpenCL) [11] so that it can be deployed and executed on any OpenCL abstracted target hardware.

The main contributions of the proposed design flow can be summarized as follows:

1. To the best our knowledge, we are the first to support model-to-model desynchronization for a synchronous imperative language. While dataflow synchronous languages share a similar structure with dataflow networks, and can utilize clock calculus to verify desynchronization criteria and synthesize communications, it is difficult to apply the same method for imperative synchronous languages.
2. We implemented a model transformation translating AIF into KPNs written in CAL which is a well-known dataflow modeling language. Such a model-to-model transformation increases flexibility. Thanks to the similarity between AIF and CAL guarded actions, the correctness of the translation is easily verified.
3. We further implemented a platform-independent code synthesis for CAL models. In particular, we offer a synthesis tool chain that automatically synthesizes CAL models into OpenCL code. This enables a more generalized synthesis not restricted to devices like multi-core processors.
4. We demonstrate the proposed design flow from specification to the final deployment using a case study of a building automation system (BAS).

B. Related Work

In synchronous program distribution, early methods tried to compile programs into automaton-like intermediate code and then distributed this code following the distributed dataflow principle [6]. Later, clock-based methods utilized clock-calculus to capture clock relations of different computations and instructed code distribution. Clock calculus is based on synchronous dataflow languages like Signal [12] and Lustre [13], and is usually not available for synchronous languages like Esterel [9] or Quartz [8] because of the different style of program semantics. Except for work by Girault et al. [6], no other work on desynchronizing imperative programs is clock-based. Their work focused on rate decomposition which starts from a single-rated specification, where we work with multi-rated specifications. Similar to work [5], [7], we need to verify that our specification satisfies the desynchronization criteria, i.e., to ensure an isochronous network of endochronous components. However, different from the classic clock-calculus based approach, we reduced the verification of endochrony into a SAT solving problem.

```
1.  module OccupancyDetect(
2.    event (bool*bool) ?change_IR, ?auto_mode, event (bool*bool) !movement){
3.     always {
4.       immediate abort
5.         every (val(change_IR)) {
6.           val(movement) = true;
7.           clk(movement) = true; // can omit this
8.         }
9.       when (!val(auto_mode));
10.    }
11. }
```

Fig. 2. The source code of module OccupancyDetect.

On the side of synthesis, a plethora of work has been done mapping synchronous dataflow programs (SDP) on multi-core architectures for better performance and power consumption. Most of these works focused on safety-critical applications in which timing predictability is of crucial importance, therefore they did not consider desynchronization, and are usually deployed on custom-made PRET architectures [1], [14]. Desynchronization has also been considered for multi-threaded code synthesis [15], [16]. These works are all based on clock calculus of SDPs and target multi-core platforms. Although it is possible to re-interpret the synchronous model asynchronously [16], no explicit model transformation is performed. Fewer works are done on code synthesis for imperative synchronous languages. ForeC [17] is a synchronous imperative language with C's style targeting PRET architectures. Although having a limited multi-rated version, it doesn't support desynchronization yet.

II. The Design Flow

A. Phase 1: Modeling and Compilation

The modeling language of our design flow is the imperative synchronous language Quartz, the core of the model-based design framework Averest. The syntax of Quartz is very similar to Esterel, which means it supports all the control flow statements like **abort, suspend**, etc. For the sake of space, we omit the language details. A small Quartz module `OccupancyDetect` is shown in Fig. 2, which reports room occupancy based on the motion detected from an IR sensor (`change_IR`), only under the auto mode (`auto_mode` is `true`). In order to support multi-clocked systems, we made a simple extension to Quartz such that each variable is modeled as a tuple of *(value, clock)*, and is denoted by $(val(x), clk(x))$. For convenience, we hereafter denote $val(x)$ as x.

Quartz supports modular design. A "`.qrz`" source file defines one module, which can be compiled into a "`.aifm`" file containing the **surface** and **depth** behavior. Modules can be linked to form a complete "`.aifs`" file, in AIF intermediate format, describing the complete system.

AIF describes a synchronous system $\mathcal{P}$ by a tuple $\langle \mathcal{V}, \mathcal{G} \rangle$ where the set of variables $\mathcal{V}$ is a disjoint union $\mathcal{V} = \mathcal{V}_{in} \cup \mathcal{V}_{loc} \cup \mathcal{V}_{out}$ of input, local and output variables, and $\mathcal{G}$ is a finite set of *synchronous guarded actions*. Guarded actions are pairs (γ, α) where α is an atomic action that is executed whenever its trigger condition (its guard) γ is true. Atomic actions are immediate assignments $\lambda = \tau$ and delayed assignments $\mathsf{next}(\lambda) = \tau$

```
Data Flow:
  G1: ((st==0) & change_IR & auto_mode) &
  (clk(change_IR) & clk(auto_mode)) => occ = True

Ctrl Flow:
  G2: ((st==2) & auto_mode & (clk(auto_mode)) => next(st) = 0
```

Fig. 3. Guarded actions and clocked guarded actions of OccupancyDetect.

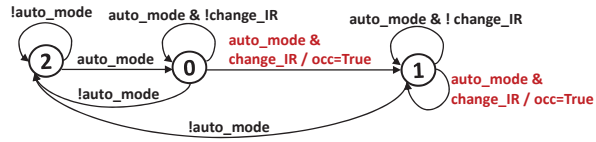

Fig. 4. The extended finite state machine of OccupancyDetect (clocks omitted).

which assigns the value of τ to λ either in the current or next cycle. In every reaction step, all the guarded actions (γ, α) are executed whose guard condition γ is true. The compiler helps to make sure that there are no causality cycles and write conflicts.

The guarded actions can be divided into control flow and dataflow groups. The control flow group encodes a finite state machine indicating how the control states evolve (Fig. 4), i.e., all their actions are assignments to state variables. Dataflow guarded actions in turn update values of data variables. Fig. 3 shows a dataflow and a control flow guarded action of `OccupancyDetect`, where $G1$ corresponds to the assignment during transition (st_0, st_1) and $G2$ encodes transitions (st_2, st_0) (which can be verified in the state machine of Fig. 4).

B. Phase 2: Transformation and Optimization

To specify a network of components, one needs to generate for each component a ".aifs" file as well as their interconnections. The augmentations are then performed in this phase to prepare for the desynchronization of this network. For supporting multi-clocked specifications, the compiler checks and augments the guarded actions in phase 1 in order to make sure that:

- For each data variable x, if it is required by the evaluation of a guarded or an assignment's expression, then $clk(x)$ should be conjuncted to the guard.
- For each data variable y, if it is produced by a guarded action, then $clk(y)$ should be set to true. If y is not updated, then $clk(y)$ should be set to false.

An augmented guarded action should look like $G1$ of Fig. 3 for the input variables, with an additional guarded action assigning $clk(occ)$ to true with the same guard as $G1$. Therefore, the programmer can omit the assignments of $clk(y) = \text{true}$ in the source code. Note that we only treat input and output event variables as clocked variables, as they are signals that may or may not be present in any cycle. The state variables and local variables are treated as memorized variables which do not have clocks. One can equally assume that their clocks are always true.

In KPN semantics each signal value is treated as a token which can only be produced/consumed once. For this reason, forks need to be translated to branching nodes duplicating the same values to the multi-casted receivers. For example in Fig. 5, we need to create a branch node for the fork $(M3 \rightarrow (M4, M5))$. Finally, optimizations like dead code and passive code elimination are also possible in this phase.

C. Phase 3: Simulation and Verification

In this phase, the system can be verified and simulated for various purposes. It is proven in [5] that if a synchronous network is *isochronous* and each node is *endochronous*, then clock signals can be safely removed, so that we can generate a KPN with the same functional behavior as the original synchronous network. In [18] this desynchronization criteria is reformed for synchronous guarded actions, stating that the synchronous system needs to be *clock consistent* and causally correct while each component is endochronous. In the following, we show how they are verified in our design flow.

1) Checking Clock Consistency: In multi-clocked specifications, a variable x is either present or absent in a cycle. If a producer generates a value for x then any consumer of x must be ready to read it. Equally, if x is expected by the consumer, the producer must produce its value. Therefore, the peer must agree on $clk(x)$ during each cycle, i.e., they need to be *clock consistent*. Assume that we have a set of synchronous components $\{\mathcal{M}_1, ..., \mathcal{M}_n\}$ and the network $\mathcal{P} := \mathcal{M}_1||...||\mathcal{M}_n$ is the composition of the components. We use $\mathcal{CC}(\mathcal{M}_i, \mathcal{M}_j)$ to denote $(\mathcal{M}_i, \mathcal{M}_j)$ are clock consistent. Since there is no clock calculus available for AIF, we utilize symbolic model checking for the verification of clock consistency. However, the verification can not be directly performed on the state transition system $\mathcal{L}_\mathcal{P}$ of $\mathcal{P}$. Since if there is any clock inconsistency, say between some state st_i of $\mathcal{M}_i$ and st_j of $\mathcal{M}_j$, then after synchronous composition, $st_i||st_j$ would evaluate to false, which wouldn't exist in the reachable state space of $\mathcal{L}_\mathcal{P}$.

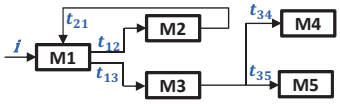

Fig. 5. A network of synchronous modules.

In order to model inconsistency inside the transition system, we add a *default behavior* to deal with the unspecified input conditions (including those clock-inconsistent ones). As for each component $\mathcal{M} := \langle \mathcal{V}, \mathcal{G} \rangle$ with $\mathcal{G} := \{(\gamma_1, \alpha_1), ..., (\gamma_n, \alpha_n)\}$, we add one more default guarded action to $\mathcal{G}$: $(\bigwedge_i (\neg\gamma_i), icc_\mathcal{M} := \text{true})$. Then the modified transition system of the network would be able to capture any module's inconsistent state. If there is any inconsistency, some component's default guarded action is triggered, and the corresponding icc signal is emitted.

It is pointed out that clock consistency is not compositional [18], i.e., $\forall i, j, \mathcal{CC}(\mathcal{M}_i, \mathcal{M}_j) \not\rightarrow \mathcal{CC}(\mathcal{M}_1, ..., \mathcal{M}_n)$. However we observed that if the network is *acyclic*, then clock consistency becomes compositional. Since the verification of clock consistency for any sub-network only puts weaker restriction on its I/O channels, when it is added by another component without loop, it will only get stronger restrictions from its input or output. For example in Fig. 5, assume $\mathcal{M}_2$ is away. If $CC(\mathcal{M}_3, \mathcal{M}_4, \mathcal{M}_5)$ holds, then the addition of $\mathcal{M}_1$ only puts

more constraints on the input of $\mathcal{M}_3$, which is no harm for the clock consistency of $\mathcal{M}_3||\mathcal{M}_4||\mathcal{M}_5$. Therefore, we only need to check if $CC(\mathcal{M}_1, \mathcal{M}_3)$ also holds. To deal with cycles in a network, we need to check additionally the clock consistency of the strongly connected components in the network.

For acyclic networks with stateless components, checking clock consistency of adjacent components is simpler. Take $\mathcal{M}_1, \mathcal{M}_3$ as an example. We only need to verify if the condition generating t_{13} implies the condition consuming t_{13}. To treat cycles, we simply specify that feedback loop signals like t_{21} are always present, so that we can ignore the signal and reduce the verification to acyclicity. This way, we can employ a state of the art SMT solver to get the job done efficiently.

2) Checking Endochrony: Endochrony is a local property of each component of a synchronous network required for desynchronization. Intuitively, a synchronous component is endochronous if there is a unique way to consume the values of the input streams for computing the reactions, once put in the asynchronous environment. Apparently, not every synchronous component is endochronous. For example, consider the component $\mathcal{M} : \langle\{x, y\}, \{g_1, g_2\}\rangle$ with g_1 : **clk**(x)=>y=x, g_2 : !**clk**(x)=>y=0 where $\mathcal{M}$ computes y based on the presence of x. Once put in the asynchronous environment, the clock information of x is gone, and there is no way to tell if a token of x is absent or on its way to $\mathcal{M}$. Endochrony however requires the component to infer deterministically the presence of the signals based solely on the arrival of data values, which $\mathcal{M}$ fails to satisfy.

Previous efforts on verifying endochrony basically all utilize clock calculus. In our design flow, we instead designed an efficient SMT verification based on AIF. In particular, our work is based on the work of [19] for the verification of one-buffered components. To verify endochrony of component $\mathcal{M}$, we check:

$$\forall s_1, s_2 \in \mathcal{R}.\forall x \in \mathcal{V}_{in}.$$
$$s_1(x) = s_2(x) \rightarrow s_1(\mathsf{clk}(x)) = s_2(\mathsf{clk}(x))$$

where $\mathcal{R}$ is the set of reachable states of the one-buffered version of $\mathcal{M}$, where each input is given a one-sized buffer. The formula states that if the input data values of two states are the same, then the clocks of the inputs should also be the same. Previous work assumed that $\mathcal{M}$ has no internal states. By exposing internal state variables as in-out buffers connected to themselves, the component can be made "stateless". If a component does not pass the verification, it does not yet mean that we cannot desynchronize it. We can identify the inputs that lead to the failure of the SMT solver, and then add the clock of inputs to the communication, so that it can finally pass the verification.

3) Generating CAL KPNs: We utilize the similarity between the guarded actions in CAL and AIF to generate the KPN model in CAL. In particular, we first generate for each control state a set of guarded actions (with a transformation in phase 2), so that a guarded action is triggered only under one state. Second, we group all guarded actions that share the same guards. Such a group of guarded actions can equally be seen as a single guarded action with an assignment to a vector of outputs. To check whether the component is endochronous, it must be

```
1.  actor OccupancyDetect() bool change_IRP, bool auto_modeP,
2.                  uint st_InP ==> bool movementP, uint st_OutP :
3.  ...
4.  Act5: action change_IRP:[change_IR], auto_modeP:[auto_mode],
5.            st_InP:[st_In] ==> movementP :[movement], st_OutP:[st_Out]
6.  guard ((st_In = 0) and (change_IR) and (auto_mode))
7.  do  movement := True; st_Out := 1; end
8.  ...
9.  end

1.  actor OccupancyDetect_St() uint st_OutP ==> uint st_InP:
2.  Act: action st_OutP:[st_Out] ==> st_InP:[st_In]
3.  do st_In := st_Out; end
4.  end
```

Fig. 6. The generated CAL actor code of module OccupancyDetect.

the case that all guards are mutually exclusive, i.e., different guarded actions cannot be triggered at the same time. This then fits the execution of the CAL guarded actions, where a CAL action is triggered only if there are sufficiently many tokens available at the input channels and the guard evaluates to true.

In this way, we can translate each guarded action group into a section of CAL guarded actions, e.g., the guarded actions from s_0 to s_1 of `OccupancyDetect` as shown in Fig. 6. Note also that we need an additional node in order to model its control state, which makes all CAL processes stateless. This is a requirement for the generation of OpenCL implementations.

D. Phase 4: Synthesis and Deployment

This phase provides a comprehensive tool chain that consists of various essential tools including the specialized code generator for the KPN MoC and the runtime system. Using OpenCL [11], it incorporates a standard hardware abstraction for cross-vendor heterogeneous hardware architectures. OpenCL offers a programming model consisting of a host and several kernels where the host is a centralized entity that is connected to one or more computing devices and is responsible for the execution of kernels. The tool chain adopts this idea of host and kernels for the synthesis as shown in Fig. 7. This section briefly discusses the essential tools of the tool chain that work together to finally implement and map the desynchronized CAL model from phase 3 on cross-vendor target hardware based on the underlying semantics of the KPN MoC.

1) Kernel Code Generation: The code generator is designed based on the underlying semantics of the KPN MoC and therefore generates an OpenCL kernel for each process (CAL actor) strictly based on the KPN MoC. In a KPN, a process is only triggered for execution if the exact information on inputs required to execute an action is available. Therefore, each time a process is triggered for an execution, a particular action is executed, mainly dependent on which guard is enabled. The guards of actions are always evaluated sequentially in the same order of their actions. The generated OpenCL kernel for the *OccupanyDetect* node as depicted in Fig. 6 is shown in Fig. 8. For brevity, we only illustrate the part of the code related to the action *Act5*:

First, the generated code peeks the tokens from all the guarded inputs into the local buffers (Lines 5-7). Next, the action *Act5* is evaluated for guard (Line 9). If the evaluation is true, the tokens are consumed from all the inputs of the action (Lines 10-12). Finally, the computations are performed (Lines 13-14) prior to writing the output tokens (Lines 15-16).

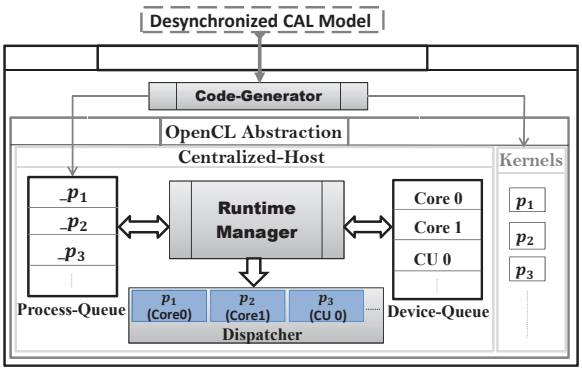

Fig. 7. The proposed synthesis tool chain.

```
1.  __kernel void OccupancyDetect(__global fifo_t* change_IRP, __global fifo_t*
2.   auto_modeP, __global fifo_t* st_InP,  __global fifo_t* movementP,
3.   __global fifo_t* st_OutP) {
4.  ...
5.     fifoPeek(change_IRP, buf_change_IRP, 1);
6.     fifoPeek(auto_modeP, buf_auto_modeP, 1);
7.     fifoPeek(st_InP, buf_st_InP, 1);
8.     /*Generate Code for Act5*/
9.     if((*change_IR_Act5) && (*auto_mode_Act5) && (*st_In_Act5 == 0)) {
10.        fifoRead(change_IRP, buf_change_IRP, 1);
11.        fifoRead(auto_modeP, buf_auto_modeP, 1);
12.        fifoRead(st_InP, buf_st_InP, 1);
13.        *movement_Act5 = true;
14.        *st_Out_Act5 = 1;
15.        fifoWrite(movementP, buf_movementP, 1);
16.        fifoWrite(st_OutP, buf_st_OutP, 1); }}
```

Fig. 8. The generated OpenCL kernel for OccupancyDetect.

2) The Runtime System: The runtime system is organized in a centralized host and kernels architecture, built under the OpenCL abstraction. The host accommodates different essential components along with the *Runtime-Manager*.

a) Process and Device Queues: The *Process-Queue* is generated by the code generator for the host. Each element of this queue provides the desired information about each process to the host such as the associated FIFO buffers, the process's status (idle, running or blocked), the associated kernel, etc. The queue, once generated, is maintained and updated by the host.

Apart from the *Process-Queue*, the host also creates a queue, namely the *Device-Queue*, using the OpenCL specification that lists all the available devices of the target hardware. Each element of this queue provides a command queue of a device, where the processes can be mapped for execution. Each command queue can represent a complete device (e.g., a CPU) or even a compute unit of that device (e.g., a CPU-core).

b) The Runtime-Manager: The heart of the centralized host around which the overall implementation circulates is the *Runtime-Manager*, as shown in Fig. 7. The Runtime-Manager is a part of the host that uses the *Process-Queue* and the *Device-Queue*, and provides the scheduler for scheduling processes based on the KPN MoC, the communication mechanism between the host and kernels, a dispatcher for mapping processes to devices, and a synchronization mechanism for processes.

Scheduling and Dispatching. The KPN Scheduler is designed for triggering processes for execution based on the underlying semantics of the KPN MoC. The KPN MoC only triggers a process for execution if the exact information on inputs/outputs required to fire an action is available. The scheduler iterates through the *Process-Queue* in a round robin fashion, and tests each process for execution based on the triggering semantics of the KPN MoC. Following the underlying semantics, the scheduler fetches a ready process from the list. The Runtime-Manager then examines the *Device-Queue* and finds the device with the least load, and dispatches the fetched process on that device. The generated kernel of the dispatched process is then executed based on the KPN MoC.

Communication and Synchronization Mechanism The data communication between the host (FIFO buffers) and the OpenCL device is carried out using OpenCL buffers. For each bounded FIFO buffer, an OpenCL buffer is created with the same design and size of the FIFO buffer. During the execution of a process (kernel), data is read/written from/to the associated OpenCL buffers. When all the dispatched instances of the kernel are executed, the Runtime-Manager is then automatically notified to update the components.

The Runtime-Manager generates a callback interface for every existing device in the *Device-Queue*. The evoked scheduler sets up a callback event for each fetched process and links it with the callback handler of the device where it is dispatched. Hence, the completion of the kernel of the dispatched process automatically invokes the callback handler of the used device. The callback handler performs a set of general tasks including: retrieving data from OpenCL buffers, updating all the FIFO buffers of the process, updating the *Process-Queue* as well as device's load and so on.

III. Case Study: Building Automation System (BAS)

We designed BAS in the spirit of [20] to validate our design flow. The purpose of BAS is to control the lamps in the rooms of a building and notify the security based on occupancy detection. Due to limited space, we demonstrate BAS with two rooms where each room has only one lamp, as shown in Fig. 9. There are three major components in the system: *OccupancyDetection*, *LampControl* and *AlarmControl*. *OccupancyDetection* takes as input the movements detected by a set of smart sensors involving infrared motion detectors and smart cameras. It receives data from these sensors and report the occupancy status of the rooms. *LampControl* takes the lamp configurations of the rooms, combined with the occupancy status, and the current and the desired illumination levels to switch the lamps. A lamp can be switched either automatically based on the occupancy or manually using a switch. *AlarmControl* collects occupancy signals and sets alarms accordingly.

We derive the guarded actions of each component and utilize the transformations in phase 2 and verification in phase 3 to generate their state transition systems. Clock consistency is verified for each directly connected set of components following Section II-C. We found an inconsistency in the first version of the system between *OccupancyDetection* and *LampControl*. The signals $occupancy1$ and $occupancy2$ were read by *LampControl* based on values of both $lampCfg$ and $occCfg$. However, *OccupancyDetection* produces the two signals only based on $occCfg$. A simple fix is to remove the influence of *lampCfg* on the reading of those signals. Once clock consistency is passed, we verified endochrony for each component where

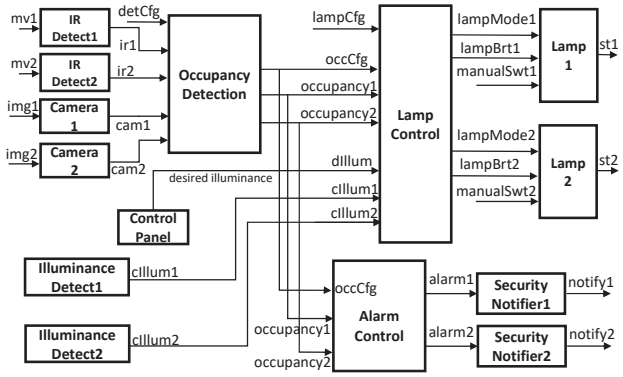

Fig. 9. The network architecture of BAS.

all modules passed the verification. This might be because with desynchronization in mind, we deliberately designed each component to be endochronous.

Once the desynchronization properties have been verified, the CAL KPN and OpenCL code are generated following phase 3 and phase 4, respectively. In order to validate our design flow, we first performed simulations for the synchronous system. Second, the automatically synthesized OpenCL implementation is executed separately on a quad-core CPU and a GPU. In particular, we run 100 cycles for the synchronous system and record the simulation trace. The trace includes the values of all the signal connections between components of the network, as well as values of the inputs and outputs of the network for all 100 cycles. The first eight cycles of the simulation with a subset of variables are shown in Table I, where emptiness of a location means that the variable is absent at that cycle.

TABLE I
FIRST TEN CYCLES OF THE SYNCHRONOUS SIMULATION.

Signals	Simulation Cycles									
	1	2	3	4	5	6	7	8	9	10
lampCfg	4	6	6	6	4	8	8	6	4	8
occCfg	3	3	2	3	3	3	3	3	3	3
occupancy1	T	T	T	T	T	T	F	T	T	T
occupancy2		F		F	F	F	F		F	T
dIllum1		2	2	2		2	2	2		2
cIllum1		1	1	0		0	0	0		1
state1	F	T	T	T	F	T	F	T	F	T
state2	F				F	F	F		F	T
notify1	T	T	T	T	T	T	F	T	T	T
notify2		F			F	F	F		F	T

This allows us to validate the functionality of our synchronous system. Furthermore, the same sequence of input values are taken out of the simulation trace and provided as the input for the synthesized OpenCL implementation. The implementation trace is collected and compared with the synchronous one, where both traces share the same sequences of values for all the variables. This validates that the generated OpenCL based KPN implementation functionally behaves the same as the original synchronous system.

IV. CONCLUSIONS

In this paper, we propose a complete design flow for the implementation of asynchronous dataflow networks from synchronous models. We developed efficient verification techniques for checking desynchronization properties and a synthesis of platform-independent OpenCL implementations. A case study validates the effectiveness of our design flow.

REFERENCES

[1] K. Didier, D. Potop-Butucaru, G. Iooss, A. Cohen, J. Souyris, P. Baufreton, and A. Graillat, "Correct-by-construction parallelization of hard real-time avionics applications on off-the-shelf predictable hardware," *ACM Trans. Archit. Code Optim.*, vol. 16, no. 3, pp. 24:1–24:27, 2019.

[2] W. Lund, S. Kanur, J. Ersfolk, L. Tsiopoulos, J. Lilius, J. Haldin, and U. Falk, "Execution of dataflow process networks on OpenCL platforms," in *Euromicro Intern. Conference on Parallel, Distributed, and Network-Based Processing*. Finland: IEEE Computer Society, 2015, pp. 618–625.

[3] F. Tirado-Andres, A. Rozas, and A. Araujo, "A methodology for choosing time synchronization strategies for wireless iot networks." *Sensors*, vol. 19, no. 16, 2019.

[4] T. Parks, "Bounded scheduling of process networks," Ph.D. dissertation, Department of Electrical Engineering and Computer Sciences, University of California, Dec 1995, phD.

[5] A. Benveniste, B. Caillaud, and P. Le Guernic, "From synchrony to asynchrony," in *Concurrency Theory (CONCUR)*, ser. LNCS, vol. 1664. Eindhoven, The Netherlands: Springer, 1999, pp. 162–177.

[6] A. Girault, "A survey of automatic distribution method for synchronous programs," in *Synchronous Languages, Applications, and Programming (SLAP)*, Edinburgh, Scotland, UK, 2005, pp. 1–20.

[7] D. Potop-Butucaru, Y. Sorel, R. de Simone, and J.-P. Talpin, "From concurrent multi-clock programs to deterministic asynchronous implementations," *Fundamentae Informaticae*, vol. 107, pp. 1–28, 2011.

[8] K. Schneider and J. Brandt, "Quartz: A synchronous language for model-based design of reactive embedded systems," in *Handbook of Hardware/Software Codesign*. Springer, 2017, ch. 2, pp. 29–58.

[9] D. Potop-Butucaru, S. A. Edwards, and G. Berry, *Compiling Esterel*, 1st ed. Springer Publishing Company, Incorporated, 2007.

[10] J. Eker and J. Janneck, "CAL language report," EECS Department, UC Berkeley, ERL Technical Memo UCB/ERL M03/48, December 2003.

[11] J. Stone, D. Gohara, and G. Shi, "OpenCL: A parallel programming standard for heterogeneous computing systems," *Computing in Science and Engineering*, vol. 12, no. 3, pp. 66–73, May-June 2010.

[12] P. Le Guernic, J.-P. Talpin, and J.-C. Le Lann, "Polychrony for system design," *Journal of Circuits, Systems, and Computers (JCSC)*, vol. 12, no. 3, pp. 261–304, June 2003.

[13] N. Halbwachs and P. RAYMOND, "A tutorial of lustre," 2001.

[14] A. Graillat, M. Moy, P. Raymond, and B. Dupont de Dinechin, "Parallel code generation of synchronous programs for a many-core architecture," in *Design, Automation and Test in Europe (DATE)*. Dresden, Germany: IEEE Computer Society, 2018, pp. 1139–1142.

[15] M. Nanjundappa, M. Kracht, J. Ouy, and S. Shukla, "A new multi-threaded code synthesis methodology and tool for correct-by-construction synthesis from polychronous specifications," in *Application of Concurrency to System Design (ACSD)*. Barcelona, Spain: IEEE Computer Society, 2013, pp. 21–30.

[16] K. Didier, A. Cohen, D. Potop-Butucaru, and A. Gauffriau, "Sheep in wolf's clothing: Implementation models for dataflow multi-threaded software," in *Application of Concurrency to System Design (ACSD)*. Aachen, Germany: IEEE Computer Society, 2019, pp. 43–52.

[17] A. Girault, N. Hili, E. Jenn, and E. Yip, "A multi-rate precision timed programming language for multi-cores," in *Forum on Specification and Design Languages (FDL)*. Southampton, United Kingdom: IEEE Computer Society, 2019, pp. 1–8.

[18] Y. Bai and K. Schneider, "Isochronous networks by construction," in *Design, Automation and Test in Europe (DATE)*. Dresden, Germany: IEEE Computer Society, 2014, pp. 1–6.

[19] Y. Bai, K. Schneider, N. Bhardwaj, B. Katti, and T. Shazadi, "From clock-driven to data-driven models," in *Formal Methods and Models for Codesign (MEMOCODE)*. Lausanne, Switzerland: IEEE Computer Society, 2014, pp. 32–41.

[20] M. Trapp, "Modeling the adaptation behavior of adaptive embedded systems," Ph.D. dissertation, University of Kaiserslautern, Germany, 2005.