A Unified Programming Model for Time- and Data-Driven Embedded Applications

Gabriela Breaban*, Sander Stuijk*, Kees Goossens*†
*Eindhoven University of Technology, The Netherlands
{g.breaban,s.stuijk,k.g.w.goossens}@tue.nl
†Topic Embedded Products, The Netherlands

***Abstract*—Modern embedded systems encompass a fast-increasing range of applications, spanning from automotive to multimedia, and industrial automation. To tackle the increasing design complexity, the model-based design paradigm promotes the use of Models of Computation (MoCs) to capture the essential application properties. Existing MoCs are split between the event/time-triggered paradigm and the data-driven paradigm. However, time and data are two inter-related dimensions that are essential for defining the correct application behavior.**

In this paper we advocate a unified MoC that integrates the notions of time and data while accounting for imperfect clocks. We present the formal properties of our model and show how the Synchronous Data Flow (SDF) MoC can be used to analyze the time performance guarantees.

I. Introduction

Embedded applications are found today in an increasing number of fields, from the cyber-physical domain to multimedia and Internet of Things (IoT). Embedded applications interact with the surrounding environment from which they receive inputs, they process them and produce outputs that are used by the environment. In the digital domain, the inputs and outputs take the form of data samples.

Traditionally, embedded systems have been seen as event-triggered [1] due to the use of mechanisms such as interrupts for interacting with internal as well as external components. Following the introduction of the Time-Triggered Architecture [2], that relies on a shared notion of time between the interconnected processors, the time-triggered paradigm emerged as a viable option for safety-critical systems [3]. On the other hand, applications that do not rely on a global notion of time are described using a data-driven paradigm [4]. Hence the primary concepts that form the foundations for describing embedded systems are events, time and data.

Model-based design [5] advocates the use of abstract models to achieve a correct-by-construction design methodology. A large number of MoCs have been defined in the embedded systems literature. Since event- and time-triggered MoCs offer a total order of the events that occur in the system, they are classified as timed models. On the other hand, Dataflow models do not incorporate a notion of time and provide a partial order of the events, and are therefore classified as untimed models [6]. Although later extensions of the original Dataflow model included actor execution times for timing analysis purposes [7], [8], the execution semantics maintained data as the only trigger mechanism for the actors. Time and data, as possible execution trigger mechanisms, need to be reconciled in a meaningful manner. While the use of time triggers ensures a uniform and predictable timing, data triggers optimize the throughput and latency. Another important observation is that the explicit inclusion of time in the semantics of time-triggered MoCs came together with the assumption that a clock synchronization mechanism is in place to offer the common notion of time. Since precise clock synchronization can be costly to implement, alternative architectures have been defined that tolerate bounded clock imperfections [9].

In this paper we propose a unified model of computation that subsumes the time-triggered and dataflow paradigms. It extends the time-triggered notion of a period by allowing offsets that enhances pipelining, it supports multi-rates, and it offers a protocol that tolerates clock jitter, while ensuring the preservation of the semantics. Due to its extended view of time, the model naturally supports a range of HW architectures from Time-Triggered Architecture (TTA), Loosely Time-Triggered Architecture (LTTA) to Globally Asynchronous Locally Synchronous (GALS). The present paper is intended as a formalization of the proposed model, and the proof of concept using real applications is part of future work. The remaining part of the paper is organized as follows: Section II presents the related work, Section III introduces the structure of the model and the operational semantics, Section IV proves the equivalence between our model and SDF and Giotto, respectively, Section V shows how the model can be analyzed and Section VI concludes the paper.

II. Related work

Eker et al. propose in [10] a hierarchical modeling framework that tackles the heterogeneous nature of cyber-physical systems at subsystem level. The system to be designed is seen as a composition of subsystems residing in different domains, modeled by distinct MoCs, that interact via communication interfaces. The authors address the problem of semantic compatibility of the composed MoCs at the communication level to ensure a well-defined model behavior. Our work, by contrast, does not aim at combining existing MoCs but rather at defining a model that is a union of a time-based MoC and a data-based MoC, while also tolerating imperfect clocks.

Arumi et al. propose in [11] an extension of the SDF MoC for the so-called callback-based architecture in which the application receives its input data through time-triggered

interrupts. The extension consists of explicitly distinguishing between time-triggered input actors, deadline-based output actors and untimed actors. Simillarly, our model distinguishes between sources, sinks and intermediate tasks. However, it is not restricted to a given architecture, and it allows an extended view of time (including offsets and support for clock jitter).

Benveniste et al. propose in [9] two protocols to tolerate clock jitter while preserving synchronous execution semantics. The first protocol is based on elastic circuits and the notion of back-pressure and the second one is time-based and allows for non-blocking communication. The protocol we propose in this paper is inspired by the back-pressure protocol, as it uses finite FIFO channels for communication between components whose clocks experience individual jitter. However our model operates at a higher granularity as it allows for multi-rates and different periods of the communicating components.

Ali et al. try to bridge the gap between traditional real-time applications and Dataflow applications by proposing in [12] an algorithm for extracting real-time parameters from Homogeneous Synchronous Data Flow (HSDF) graphs in order to use them in real-time scheduling algorithms. The task real-time parameters, that comprise offset, period, deadline and execution time, are derived based on end-to-end latency constraints. In comparison, our model allows the coexistance of all the previously mentioned real-time parameters together with the Dataflow properties in a unified manner, so that either real-time scheduling algorithms or static Dataflow scheduling can be applied, depending on the specific structure of the application. However, we advocate the explicit support of different types of hardware platforms such as TTA, LTTA and GALS, as the notions of time and data are not equally enforced on each one of them.

III. Model Structure and Operational Semantics

A. Model Components

The components of our MoC are: clocks, tasks, sources, sinks and communication channels. We will exemplify each component using the example application in Figure 1.

A clock is a time source, that provides an infinite series of clock ticks, separated by the clock period. When a system comprises multiple clocks, the duration of each clock period will experience variation over time due to physical phenomena such as jitter or drift. This can be a problem when a common notion of time is required by the application semantics. To counteract it, a clock synchronization algorithm can be used to obtain a single time unit.

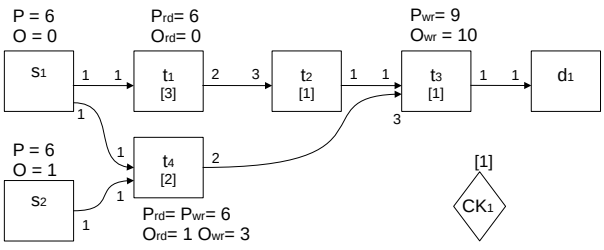

Fig. 1. Running Example

We allow for two types of clocks. The application can have a set of clocks that are perfectly synchronized with each other (e.g. using clock synchronization) effectively providing a single time unit reflected by the clock period. Alternatively, we consider the case when the clocks are affected by jitter (and not drift) and are not synchronized. Jitter causes a bounded variation of the clock period, so the clock can be characterized by a minimum and a maximum clock period. In our model we only consider the maximum clock period, since the aim is to provide worst-case guarantees for throughput and latency. These two types of clocks cannot coexist in a single model: either all the clocks are synchronized or they all experience jitter. In order to measure and compare distinct clock periods and their jitter, we need an external precise time reference. In this paper, we assume that this reference is the wall time.

Definition 1. *(Synchronized Clock) A synchronized clock c is a component that has a period p such that $t_c(n+1)-t_c(n) = p$, $\forall n \in \mathbb{N}$, where n is the clock tick index and $t_c(n)$ is the instant of the n'th tick.*

When all the clocks in a system are synchronized, the actual period p can be abstracted away, as all the remaining time-related notions can be expressed as a multiple of the same time unit that is equal to p. Figure 1 shows a synchronized clock CK_1 with a period of 1 time unit.

Definition 2. *(Jittery Clock) A jittery clock c has a maximum period $p_{max} = p+j$ such that $p-j \leq t_c(n+1)-t_c(n) \leq p+j$, $\forall n \in \mathbb{N}$, where n is the clock tick index, $t_c(n)$ is the instant of the n'th tick, p is the nominal clock period and j is the jitter.*

Given a clock c of type synchronized or jittery, let $crtTime(c,t) \in \mathbb{N}$ be a function that returns the total number of clock ticks at time $t \geq 0$.

A third concept that we include in our model to express time is the offset. An offset represents an initial delay with respect to a time instant.

Definition 3. *(Offset) Given a clock tick at time instant t_i, an offset o will produce a tick at time instant $t_o = t_i + o$.*

The processing components in our model are called tasks.

Definition 4. *(Task) A task is a tuple $\tau = (I, U, e, p_r, o_r, p_w, o_w)$, where I is a finite set of input ports, U is a finite set of output ports, e is the execution time, p_r and p_w are read and write periods, o_r and o_w are read and write offsets. $e, p_r, o_r, p_w, o_w \in \mathbb{N}$ are expressed as a multiple of a given time unit p, where p is the (maximum) period of a clock c. Let $taskClk : T \rightarrow C$, where C is a set of clocks (synchronized or jittery) and T is a set of tasks, be a function that returns the clock c associated with task τ.*

Each input/output port has a positive integer number associated with it called *rate*. Given a set of ports P, let $r : P \rightarrow \mathbb{N}_{>0}$ be a function that returns the rate of a port $p \in P$. Further,

we will write $p \in a$ to mean that port p belongs to a task/source/sink a.

The read/write period and offsets are optional and they only need to be specified when the task is required to read or write data periodically. We write $P_x = \varnothing,\ O_x = \varnothing,\ x \in \{w, r\}$ to mean that the read/write period and offset are not defined.

The example application in Figure 1 contains four tasks, t_1 to t_4. Task t_1 has a read period of 6 and read offset of 0. Task t_3 has a write period of 9 and write offset of 10, and task t_4 has a read and a write period, both equal to 6 and read and write offset of 1 and 3 respectively. Note that when both a read and a write period are defined for the same task, they have to be equal. The time unit is given by clock CK_1. Task execution times are shown between square brackets.

The application receives data in the form of samples from sources, it processes it and sends the resulting output samples to data sinks. We assume that all sources are periodic and their period and offset are known.

Definition 5. *(Source) A source is a tuple $s = (U, p_s, o_s)$ where U is a set of output ports, p_s is the source period and o_s is the offset. Let $srcClk : S \rightarrow C$, where C is a set of clocks (synchronized or jittery) and S is a set of sources, be a function that returns the clock c associated with source s, that gives the time unit for p_s and o_s.*

The period and offset (potentially 0) are mandatory properties for a source.

Definition 6. *(Sink) A sink is a tuple $d = (I, p_d, o_d, e)$ where I is a set of input ports, p_d is the sink period, o_d is the offset and e is the execution time. Let $sinkClk : D \rightarrow C$, where C is a set of clocks (synchronized or jittery) and D is a set of sinks, be a function that returns the clock c associated with sink d, that gives the time unit for p_d and o_d.*

The sink period and offset are not mandatory properties.

Figure 1 shows a source s_1 with a period of 6 and offset 0, source s_2 with the period of 6 and offset 1, and a sink d_1.

The tasks, sources and sinks communicate data via channels. A channel expresses a data dependency between a task/source and another task/sink.

Definition 7. *(Channel) A channel is a tuple $\gamma = (u, i)$, $\gamma \in \Gamma$ where u is an output port and i is an input port, such that $\forall \gamma_i = (u_i, i_i) \in \Gamma,\ \gamma_i \neq \gamma \implies u_i \neq u \ \wedge \ i_i \neq i$. Given a channel γ, let $q(\gamma, t)$ be a function that returns the number of data samples present in channel γ at time instant t.*

The above channel definition allows an input/output port to be connected to only a single communication channel. In our model the communication channels are assumed to be First In First Out (FIFO) memories characterized by destructive reads and non-destructive writes. A destructive read means that after reading a sample, the space occupied by the sample is freed. A non-destructive write means that a write operation will block until there is sufficient free space in the FIFO. In Figure 1, the channels are shown as arrows. The FIFO size is assumed to be infinite by the model. However, at design time, the finite size has to be properly dimensioned based on the known write and read frequencies at the output and input ports.

Definition 8. *(Application) An application is a tuple $A = (T, \Gamma, C, S, D)$, where T is a set of concurrent tasks, Γ is a set of communication channels, C is a set of clocks, S is a set of sources and D is a set of sinks.*

We define two criteria for an application to be valid. The first criterion is called *data consistency* and the second one is called *time consistency*.

Data consistency is a concept inherited from synchronous dataflow [4]. We present it here briefly. For in depth details we refer the reader to [13].

Definition 9. *(Balance Equations) A balance equation is defined for a communication channel (u, i) connecting an output data port u with an input data port i as follows:*

$$r(u) \cdot f(a) = r(i) \cdot f(b),\ p_o \in a,\ p_i \in b \tag{1}$$

where a is a task/source and b is a task/sink, $r(u)$, $r(i)$ are the rates of ports u, i.

After writing the balance equations for all the data channels, at least a solution has to be found for all the quatities denoted $f(k)$, where k denotes the task/source/sink connected to the corresponding port. We will call the minimum solution for $f(k)$ the repetition value of k. When a solution cannot be found, we say that the aplication specification is inconsistent and it cannot be analyzed.

Definition 10. *(Data consistency) Given an application (T, Γ, C, S, D) and a set of balance equations for all channels $c \in C$, we say that the application is data consistent if a solution can be found to the set of balance equations.*

When the application includes read/write periods for the tasks, sources and sinks, the periods require also a consistency check called time consistency. Let us define this concept for the case of synchronized clocks.

Definition 11. *(Time consistency) Given an application (T, Γ, C, S, D) and $\exists\ T_p \subset T$ such that $\forall \tau \in T_p$, τ has a read period p_r and/or a write period p_w defined and $\exists\ D_p \subset D$ such that $\forall d \in D_p$, $p_d \neq \varnothing$, then time consistency is defined as follows:*

$$p_{u_1} \cdot f(w_1) = .. = p_{u_n} \cdot f(w_n), \forall w_i \in \{S, T_p, D_p\} \tag{2}$$

In the equation above w_i is either a source, a periodic task or sink, p_{u_i} is its (read/write) period, and $f(w_i)$ represent the repetition value of component w_i, as obtained in the data consistency check. When a task has both a read and a write period, they both need to be included in the equality.

Data and time consistency ensure that the application execution consists of a repetitive state with respect to time and data. The strict periodicity of tasks can only be obtained when a single notion of time exists. For jittery clocks, time consistency is explained in Subsection III-C.

B. Operational Semantics for Synchronized Clocks

A clock provides a time unit for the tasks and sources. When all the clocks of an application are synchronized they provide the same time unit. This subsection presents the operational semantics for the tasks, sources and sinks in this specific case. Further, the following set of assumptions have to hold:

Assumption 1: Every data sample in a communication channel is written and read exactly once.

This assumption is in fact a consequence of having FIFO communication channels in which a data sample can only be read after it has been written and a new sample can be writen in the same location after the previous sample has been read.

Assumption 2: For every task τ with a read period p_r and offset o_r, sufficient data samples are present on each of its input channels at the time instants $o_r + k \cdot p_r,\ k \geq 0$, with the time unit given by a synchronized clock $c = taskClk(\tau)$.

Definition 12. *(Task Invocation) The execution of a task $\tau = (I, U, e, p_r, o_r, p_w, o_w)$ is called* task invocation*. A task invocation $\tau[k],\ k \geq 0$ consists of the following steps:*

1) ***Evaluate the read condition**:*
   a) ***Data-driven Read**: if $p_r = \varnothing \wedge\ o_r = \varnothing$ then, given a set of channels Γ_i such that $\forall p_i \in I,\ \exists\ \gamma = (u, i) \in \Gamma_i,\ p_i = i$ we say that the read condition of $\tau[k]$ evaluates to true if $q(\gamma, t) \geq r(p_i)$.*
   b) ***Time-driven Read**: if $p_r \neq \varnothing \wedge\ o_r \neq \varnothing$ then the read condition of $\tau[k]$ evaluates to true when $crtTime(c, t) = k \cdot p_r + o_r,\ c = taskClk(\tau)$.*
2) ***Read phase**: If the read condition of $\tau[k]$ evaluates to true, then given the set of channels Γ_i connected to task input ports such that $\forall p_i \in I,\ \exists\ \gamma = (u, i) \in \Gamma_i,\ p_i = i$, the read phase of $\tau[k]$ will remove a number of samples equal to $r(p_i)$ from each $\gamma \in \Gamma_i$.*
3) ***Process phase**: in this phase the task uses the read input data and executes for a time duration equal to e.*
4) ***Evaluate the write condition**:*
   a) ***Data-driven Write**: if $p_w = \varnothing \wedge\ o_w = \varnothing$ then the write condition of $\tau[k]$ always evaluates to true*
   b) ***Time-driven Write**: if $p_w \neq \varnothing \wedge\ o_w \neq \varnothing$ then the write condition of $\tau[k]$ evaluates to true when $crtTime(c, t) = k \cdot p_w + o_w,\ c = taskClk(\tau)$.*
5) ***Write phase**: If the write condition of $\tau[k]$ evaluates to true, then given the set of channels Γ_o connected to task output ports such that $\forall p_u \in U,\ \exists\ \gamma = (u, i) \in \Gamma_o,\ p_u = u$, the write phase of τ will add a number of samples equal to $r(p_u)$ into each $\gamma \in \Gamma_o$.*

Both the read and write conditions are blocking, meaning that the read and write phase only take place after the corresponding condition evaluates to true.

Also note that the read condition in step 1) represents a necessary condition for the next step, and not a sufficient condition. In other words, when an application is executed on a platform, the scheduler can always postpone a task invocation for which the read condition evaluates to true. Hence the sufficient condition for a executing a task invocation is ultimately given by the scheduler.

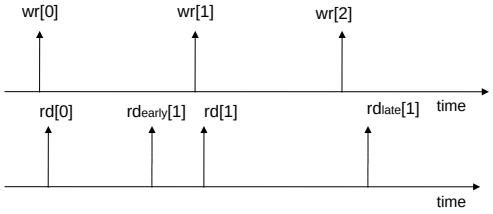

Fig. 2. Jitter Effects

The execution of the sources and the sinks is also called invocation. A source will write on each output port a number of samples equal to the port rate, as specified by Definition 5. A sink, on the other hand, will read on each invocation a number of samples equal to the port rate, when the corresponding read condition evaluates to true, and if an execution time is defined, then the read phase will be followed by a process phase.

C. Accounting for clock jitter

A common notion of time ensures deterministic communication. This means that for two communicating time-driven tasks, τ_1 and τ_2, for which the j-th invocation of τ_2 reads the data written by the i-th invocation of τ_1, $i \geq j$, the time-driven write phase of the i-th invocation of τ_1 always precedes the time-driven read phase of the j-th invocation of τ_2. This precedence can be achieved when both tasks share the same notion of time. Clock jitter determines random variation of the clock period. Hence, when the two tasks are driven by distinct jittery clocks, the above mentioned precedence can be violated due to independent period variation of the individual clocks, leading to non-deterministic communication.

When the time-driven communication is realized via FIFOs, clock jitter can lead to two cases: the reader might either attempt to read the data before it has been written (when the reader's clock is too fast) or, in the opposite case, the writer might attempt to overwrite data that hasn't been read yet (when the writer's clock clock is too fast). The former case can lead to FIFO underflow and the latter to overflow. Figure 2 illustrates these two cases for a one-place FIFO. The k-th invocation of the reader, rd[k] is supposed to access the data written by the k-th invocation of the writer, wr[k], k = 0,1,2. The arrows labelled wr[1] and rd[1] show the case when none of the clocks experiences jitter and the reader can access the data right after it was written. rd$_{\text{early}}$[1] shows the situation when the reader attempts to read the data before it was written and rd$_{\text{late}}$[1] shows the opposite case when the writter overwrites the data from invocation wr[1] before rd$_{\text{late}}$[1] occurs.

To overcome the effects of jitter for unsynchronized clocks on time-triggered architectures, we propose a protocol inspired by [14] that prevents FIFO under- and overflow. The protocol is called back-pressure-based protocol and it is based on the use of elastic circuits together with a skipping mechanism. An elastic circuit is, effectively, a FIFO, while a skipping mechanism allows for the FIFO to be polled by a local clock.

At the reader side, polling ensures that the read phase will only occur after the writer produced the necessary amount of data, therefore preventing underflow. At the writer side, back-pressure ensures that the writer will only attempt to write new samples after the reader has freed enough space in the FIFO, therefore preventing overflow.

To support this protocol we need to enhance the semantics presented in the previous subsection. Given a channel $\gamma = (p_o, p_i)$, where $p_o \in \tau_1$, $p_i \in \tau_2$ the back-pressure protocol extends the set of input ports of τ_1 with an additional port p'_i, $r(p'_i) = r(p_o)$, the set of output ports of τ_2 with an additional port p'_o, $r(p'_o) = r(p_i)$ and the set of application channels Γ with $\gamma' = (p'_o, p'_i)$. Further, for the reader, as well as for the writer, when it has a time-driven read, the time-driven read condition is extended to add a condition that checks the presence of data as follows:

Definition 13. ***Enhanced Time-driven Read****: If $p_r \neq \varnothing \wedge o_r \neq \varnothing$ then, given a jiterry clock c with a maximum period p_{max}, $c = taskClk(\tau)$ and a set of channels Γ_i such that $\forall p_i \in I,\ \exists\ \gamma = (o, i) \in \Gamma_i,\ p_i = i$ we say that the read condition of $\tau[k]$ evaluates to true when $crtTime(c, t) \geq k \cdot p_r + o_r \wedge q(\gamma, t) \geq r(p_i),\ \forall \gamma \in \Gamma_i$, where the time unit of p_r, o_r is p_{max}.*

Using this extended condition, the reader task will first wait until the start of the current period, $k \cdot p_r + o_r$, as given by its own jittery clock. Afterwards, it will poll until sufficient data is present on each of its input channels ($q(\gamma, t) \geq r(p_i)$). The writer, on the other hand, will check that sufficient space is available for the current write phase during the evaluation of the read condition preceding the current write phase.

The extended semantics effectively transform the time-driven read into a time- and data-driven read to counteract the effects of clock jitter. When an application including jittery clocks and finite channels is executed as soon as possible, according to the specified operational semantics, it will reach a periodic state. This result follows from the SDF theory [7], which we use for timing analysis (see Section V). Then, the periods p_{u_i} in Definition 11 represent average periods rather then strict period. An average period is computed by dividing the application period (from the periodic phase) by the repetition value, $f(w_i)$.

IV. Formal Properties of the Proposed Model

We claim that our MoC unifies the notions defined by the timed SDF MoC, the single-mode Giotto MoC and the LTTA protocol. We will prove this formally by showing that there exists a structural as well as an operational correspondance between each of these models and a subset of our model.

A. Equivalence Relation to Timed SDF

A timed SDF model consists of a set of actors and a set of edges [7]. An actor is defined by a set of input ports, a set of output ports and an execution time. An edge is defined by a source port and a destination port.

To show the equivalence between timed SDF graphs and our model, we will distinguish between three types of actors: actors that contain both input as well as output ports, actors with no input ports and actors with no output ports.

Lemma 1. *There is a one-to-one relation between a SDF actor $a = (I, U, e)$ where I is a set of input ports, O is a set of output ports and e is an execution time, such that $I \neq \emptyset \wedge U \neq \emptyset$ and a task $\tau = (I, U, e, p_r, o_r, p_w, o_w)$ for which $p_r = o_r = p_w = o_w = \varnothing$.*

Proof. Structurally, an actor port is identical to a task port, as both are essentially a communication interface. The concept of actor execution time and task execution time are also identical.

In terms of execution semantics, the execution of a SDF actor (i.e. a firing) is the same as the invocation of a task that has a data-driven read and a data-driven write phase. Indeed, according to Definition 12, when $p_r = o_r = p_w = o_w = \varnothing$, task τ has a data-driven read and write condition. □

Lemma 2. *There is a one-to-one relation between a non-auto-concurrent SDF actor $a = (I, U, e)$ where I is a set of input ports, O is a set of output ports and e is an execution time, such that $I = \emptyset \wedge U \neq \emptyset$ and a source $s = (U, p_s, o_s)$ for which $p_s = e$ and $o_s = 0$.*

Proof. Structurally, the same considerations as for Lemma 1 apply here.

In terms of execution semantics, a non-auto-concurrent actor is an actor that does not have multiple concurrent executions. Such an actor executes sequentially and each execution takes an amount of time equal to e. Moreover, when the actor has no input ports and thus no input data dependencies, then the executions will be back-to-back, with no intermediate delays. Then a sequence of actor executions is identical to a sequence of periodic source invocations, that consist of write phases separated by the period. □

Lemma 3. *There is a one-to-one relation between a SDF actor $a = (I, U, e)$ where I is a set of input ports, O is a set of output ports and e is an execution time, such that $I \neq \emptyset \wedge U = \emptyset$ and a sink $d = (I, p_d, o_d, e)$ for which $p_s = o_s = \varnothing$.*

Proof. Structurally, the same considerations as for Lemma 1 apply here.

In terms of execution semantics, the SDF actor will start as soon as sufficient data is present on each input port and then execute for an amount of time equal to e. Similarly, the sink will execute the read phase when sufficient data is present on each input port and then proceed with the process phase for e time units. Hence, the execution semantics are equivalent. □

Theorem 1. *All valid timed SDF application models are valid models in our formalism.*

Proof. The proof follows directly from the Lemmas 1, 2 and 3 and the equivalence between SDF edges and the channels in our model, both assumed to be FIFO memories. □

B. Equivalence Relation to Single-Mode Giotto

A single-mode Giotto model consists of a set of concurrent tasks, sensors and actuators [3]. The tasks read data from sensors and write data to actuators. The mode definition specifies a mode period and frequencies for task invocations and actuator updates. A frequency represents the number of invocations/updates of a task/actuator during the mode period. All invocations/updates are equally spaced in time such that the task/actuator period is equal to the mode period p_m divided by the frequency. The communication is realized through Giotto ports, using drivers. A Giotto port is a persistent memory location that retains the latest written value. Each sensor and actuator has its own Giotto port. Finally, a driver transports data between a set of source ports (from sensors or tasks) to a set of destination ports (from actuators or tasks).

To distinguish between Giotto ports and the ports in our model, we refer to the latter as enhanced ports. Structurally, a Giotto port acts as a memory location which allows multiple successive read or write operations, while an enhanced port acts as a FIFO location which only allows single write and read operations to occur after each other, prohibiting thus the loss of data. To properly adress the equivalence, we will consider the known write and read periods of the corresponding tasks and show how an equivalent behavior can be achieved through adequate mechanisms (see Lemmas 7 and 8).

Let $t = (In, Out, f_t)$ denote a Giotto task t with frequency f_t containing a set In of input Giotto ports and a set Out of output Giotto ports and $drv = (Src, Dst)$ denote a driver drv between a set Src of source ports and a set Dst of destination ports, $sens(p, f_s)$ is a sensor with port p and frequency f_s and $act(p, f_a)$ an actuator act with a port a and frequency f_a.

For the remaining part of this subsection, we will assume that given a Giotto model, there exists a time unit $\in \mathbb{Q}^+$ such that $\frac{p_m}{lcm(f_i)}$ for all specified frequencies f_i, where lcm is the least common multiple.

Definition 14. *(Sensor frequency) Given a sensor with a Giotto port p and a set of drivers D such that $p \in Src,\ \forall drv = (Src, Dst) \in D$ and T is a set of tasks such that $\forall t = (In, Out, f_t) \in T,\ \exists\ p' \in In \wedge \exists\ drv' = (Src', Dst') \in D,\ p' \in Dst'$, then the sensor frequency f_s is equal to $lcm(f_t)$, $\forall t \in T$.*

Intuitively, this definition says that a sensor frequency is given by the lcm of the frequencies of the reading tasks.

Lemma 4. *Given a sensor $sens(p, f_s)$ and a set of drivers D such that $\forall drv = (Src, Dst) \in D,\ p \in Src$, then for each $drv \in D$ define a new enhanced port output p_o with rate 1. There exists a one-to-one correspondance between a sensor $sens(p, f_s)$ and a source $s = (U, p_s, o_s)$ where U is a set comprising all previously defined enhanced output ports p_o, $p_s = \frac{p_m}{f_s}$, $o_s = 0$ and p_m is the Giotto mode period.*

Proof. Structurally, the Giotto port of the sensor, p, is transformed into multiple enhanced ports, one for each driver connected to it. Essentially, a Giotto port shared between multiple connections is transformed into multiple identical enhanced ports connected to point-to-point connections. Thus, the contents of the enhanced ports are all equal to the content of p. Further, Giotto ports have an implicit rate of 1.

In terms of execution semantics, the source writes periodically its output ports with period p_s, which is equivalent to the periodic update of the Giotto sensor port with period p_s. □

Lemma 5. *Given an actuator $act(p, f_a)$ and a set of drivers D such that $\forall drv = (Src, Dst) \in D,\ p \in Dst$, then for each drv in D define a new enhanced input port p_i with rate 1. There exists a one-to-one correspondance between an actuator $act(p, f_a)$ and a sink $d = (I, p_d, o_d, e)$ where I is a set comprising all previously defined enhanced input ports p_i, $p_d = \frac{p_m}{f_a}$, $o_d = 0$ and p_m is the Giotto mode period.*

Proof. The proof is the same as for Lemma 4. □

Lemma 6. *Given a Giotto task $t(In, Out, f_t)$ and a set of drivers D_i such that $\forall drv = (Src, Dst) \in D_i,\ p \in Dst$, then for each drv in D_i define a new enhanced input port p_i with rate 1. Similarly, for a set of drivers D_o such that $\forall drv = (Src, Dst) \in D_o,\ p \in Dst$, for each drv in D_o define a new enhanced output port p_o with rate 1. There exists a one-to-one correspondance between the task $t(In, Out, f_t)$ and a task $\tau = (I, U, e, p_r, o_r, p_w, o_w)$ where I is a set comprising the defined enhanced input ports p_i, U is a set comprising the defined enhanced output ports p_o, $p_r = p_w = o_w = \frac{p_m}{f_t}$, $o_r = 0$, and p_m is the Giotto mode period.*

Proof. Structurally, the transformation between the Giotto ports and the enhanced ports follows the same principals as in the previous lemmas. In terms of execution semantics, the task will read and write data with the period $\frac{p_m}{f_t}$. The read offset is 0 since in Giotto, all tasks read at the beginning of the mode period, while the write offset is equal to the period, since the first write operation occurs after one invocation. □

Definition 15. *(Sampling Ratio) Given a driver drv, a sampling ratio between a source port $p_s \in v$ and a destination port $p_d \in w$ is a pair (p, q) where $p = \frac{f_v}{gcd(f_v, f_w)}$ is the frequency of the sensor/task v connected to port p_s and $q = \frac{f_w}{gcd(f_v, f_w)}$ is the frequency of the actuator/task connected w to port p_d. We write $p : q$ to denote a sampling ratio (p, q).*

To define the correspondance between Giotto drivers and the channels in our model, we will consider each pair of source and destination ports (p_s, p_d) connected by a driver drv and distinguish between two cases, based on the value of the sampling ratio $p : q$ between the two ports.

Lemma 7. *Given a driver drv from the source port p_s to the destination port p_d such that the sampling ratio between p_s and p_s is $1 : 1$, then there exists a one-to-one relation between the tuple (drv, p_s, p_d) and a channel $\gamma = (p_s, p_d)$ where the rates of p_s and p_d are both 1.*

Proof. A connection between a source and a destination Giotto ports that are written/read with the same frequency f is equivalent to having a FIFO channel between two ports with rate 1 that are written/read with the frequency f. □

Lemma 8. *Given a driver drv from the source port p_s to the destination port p_d such that the sampling ratio between p_s and p_d is $p : q$, such that $(p = 1 \wedge q \neq 1) \vee (p \neq 1 \wedge q = 1)$ then there exists a one-to-one relation between the tuple (drv, p_s, p_d) and the tuple $(\gamma_1, \tau_{aux}, \gamma_2)$ where $\gamma_1 = (p_s, p'_s)$, $\gamma_2 = (p'_d, p_d)$, $\tau_{aux} = (p'_s, p'_d, 0, \varnothing, \varnothing, \varnothing, \varnothing)$, $r(p_s) = r(p_d) = 1$, $r(p'_s) = p$ and $r(p'_d) = q$.*

Proof. To ease the understanding of this lemma, we illustrate the relation in Figure 3. The auxiliary task τ_{aux} ensures the data consistency between p_s and p_d. Recall that given two components v and w such that $p_s \in v$ and $p_d \in w$, data consistency requires that $r(p_s) \cdot f(v) = r(p_d) \cdot f(w)$ where the repetition values $f(v)$ $(f(w))$ are equivalent to the frequencies f_v (f_w). Since $r(p_s) = r(p_d) = 1$ and $f(v) \neq f(w)$, the equation has no solution. By introducing τ_{aux}, the previous equality is transformed into the following set of equalities: $r(p_s) \cdot f(v) = r(p'_s) \cdot f(\tau_{aux})$ and $r(p'_d) \cdot f(\tau_{aux}) = r(p_d) \cdot f(w)$ for which the solution is $f(\tau_{aux}) = \frac{f(u)}{p} = \frac{f(w)}{q}$. □

The last two lemmas present two cases for the sampling ratio $p : q$. One third case that we don't cover in the current paper is when p and q are relatively prime numbers.

Theorem 2. *All valid single-mode Giotto application models are valid models in our formalism.*

Proof. The proof follows directly from the lemmas 4 to 8. □

C. Correctness of the Back-Pressure Protocol for Clock Jitter

Theorem 3. *Given a channel $\gamma = (p_o, p_i)$ between tasks τ_1 and τ_2 such that τ_1 has write period $p_w = n_1 \cdot p_1$ and offset $o_w = m_1 \cdot p_1$ and τ_2 has read period $p_r = n_2 \cdot p_2$ and offset $o_r = m_2 \cdot p_2$, where p_1 is the maximum period of c_1, $c_1 = taskClk(\tau_1)$ and p_2 is the maximum period of c_2, $c_2 = taskClk(\tau_2)$, let γ' denote γ augmented with the back-pressure protocol defined by condition 1) b)'. Then γ' prevents the under- and overflow of the FIFO associated to γ.*

Proof. Let us first address the underflow case. By augmenting c with the back-pressure protocol, the consumer's read condition becomes $crtTime(c, t) \geq k \cdot p_r + o_r \wedge q(\gamma, t) \geq r(p_i)$. Let us assume that the k-th invocation of τ_2 consumes $r(p_i)$ tokens produced by invocations $k + p$ to $k + p + n$ of τ_1, $p, n \geq 0$, $(n + 1) \cdot r(p_o) \geq r(p_i)$. FIFO underflow would occur if the k-th invocation of τ_2 would read the FIFO before the $k+p+n$-th invocation of τ_1 finishes. This is not possible

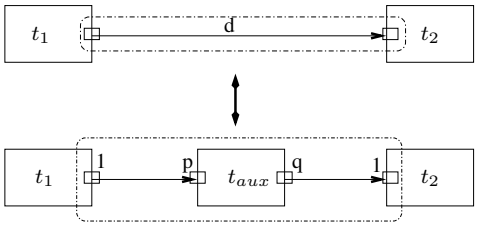

Fig. 3. Equivalence Relation for Lemma 8

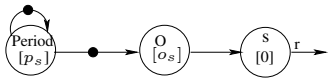

Fig. 4. SDF model for a source

due to the condition $q(\gamma, t) \geq r(p_i)$ and the destructive read of the FIFO, which ensures that when the amount of tokens in γ reaches $r(p_i)$, then they can only be produced by invocations $k + p$ to $k + p + n$ of τ_1.

To address the overflow case, let us consider the two possible types of read condition for τ_1. If τ_1 has a data-driven read, then the back-pressure protocol extends the set of input ports of τ_1 with port p having a rate $r(p) = r(p_o)$. This will ensure that invocations $k + p$ to $k + p + n$ of τ_1 will only finish if at least $(n + 1) \cdot r(p_o)$ locations are available in the FIFO. Hence the producer will never overwrite the buffer if it becomes full. If τ_1 has a time-driven read, then the back-pressure protocol extends the set of input ports of τ_1 with port p having a rate $r(p) = r(p_o)$ and it extends the read condition to $crtTime(c, t) \geq k \cdot p_r + o_r \wedge q(\gamma, t) \geq r(p)$. As in the previous case, the added condition $q(\gamma, t) \geq r(p)$ prevents τ_1 from overwriting the FIFO. □

V. SDF Analysis Model

To provide worst-case timing guarantees, we use the SDF model of computation. In this section, we will present the modeling techniques that allow us to translate an application written in our model to an analyzable SDF graph. We use the timed SDF formalism, in which actors have an allocated worst-case execution time (WCET). Within the SDF MoC, the data samples are called tokens, the computational components that correspond to the tasks in our model are called actors and their execution is called firing. SDF actors include input and output ports, thus our component ports can be directly translated to actor ports. The communication channels in our model correspond to edges.

A. Analysis Model for Synchronized Clocks

Let us start with the simplest components, sources and sinks. A source having a period p_s and offset o_s is modeled by three actors, as shown in Figure 4. The Period actor models the source period and the O actor models the offset. The token on the channel between Period and O will enable the source to start its first firing at the time instant o_s. The following firings will start at times $o_s + k \cdot p_s$, $k \geq 1$, according to the source definition. At each firing, the source will produce on each of its output ports, a number of tokens equal to the port rate r.

A sink $d = (I, p_d, o_d, e)$ is modeled similarly, with the difference that it has an execution time of e instead of 0.

A task is modeled by a chain of three actors: Rd, Exec and Wr. Figure 5 shows the model for an example task containing two input ports with the rates m and n, respectively and two output ports with the rates p and q, respectively, and no read or write period and offset. Actor Rd models the first two steps of the task invocation. The evaluation of the read condition is

Fig. 5. SDF model for a data-driven task

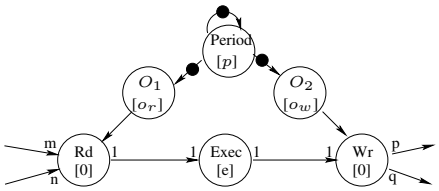

Fig. 6. SDF model for a time-driven task

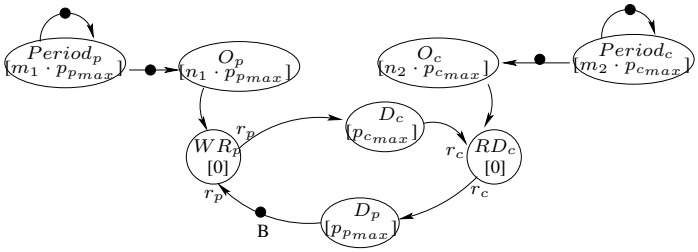

Fig. 7. SDF model for jitter

implicitly realized by the SDF firing rules, that allow the actor to fire as soon as a sufficient number of tokens are present on each input channel. By default, the Rd actor has an execution time of 0, but it can be modified to reflect a higher read time. Next, the Exec actor models the process phase and finally, the Wr actor models the last two execution phases.

Figure 6 shows the SDF model for a time-driven task with a write and read period of p, read offset o_r, write offset o_w.

Note that the Period actors included in the source and task models can be either shared or distinct, depending on the exact values of the periods. However, for the case of synchronized clocks, there is a single time unit which is reflected in the execution time of the clock and offset actors.

For a given application $A = (T, \Gamma, C, S, D)$, its associated SDF graph $G(A) = (V, E)$ is constructed by representing each source, task and sink by its designated SDF model and each communication channel by a graph edge. V represents the set of resulting SDF actors and E the resulting edges.

B. Analysis Model for Jittery Clocks

Figure 7 presents the SDF model for the back-pressure protocol presented in subsection III-C. The model shows the communication between a producer and a consumer task driven by two distinct jittery clocks. The maximum period of the producer clock is $p_{p_{max}}$ while for the consumer clock it is $p_{c_{max}}$. The write period and offset of the producer task is a multiple of the local clock period and the same holds for the read period and offset of the consumer task. Since the SDF model is used for worst-case performance analysis, the maximum clock period is used to obtain conservative performance bounds. The communication is realized via a FIFO that the consumer polls using its local clock. The actors D_c, D_p model the worst case arrival time of the tokens when the last one of the r_c tokens arrives right after the local clock tick. The consumer then has to wait for an extra clock period.

To model back-pressure, we add a backwards path from the consumer's read actor to the producer's write actor. The token labelled B on the edge towards WR_p models the size of the FIFO, equal to B locations.

C. Performance Analysis

The SDF analysis model allows us to compute the minimum throughput [7] and the minimal latency between a source and a sink [15] for a given application and also check that the defined periods and offsets are met. The analysis method that enables this is called state-space analysis and it consists of simulating the SDF graph that results by modeling the application according to the techniques described previously.

VI. Conclusions

In this paper we propose a unified model of computation that combines time and data execution semantics, while accounting for clock jitter. We formalize the structure of the model and the operational semantics. We prove the correspondance between a single-mode Giotto model and our model, between an SDF model and our model. Finally, we show how to compute performance guarantees using SDF.

Acknowledgment

This research is partially supported by EU grants CT217 RESIST, ECSEL 692455-2 ENABLE-S3, 737422 SCOTT, 737453 I-MECH, NL grant STW ZERO.

References

[1] E. A. Lee *et al.*, "Discrete-event models," in *System Design, Modeling, and Simulation using Ptolemy II*, 2014.

[2] H. Kopetz *et al.*, "The time-triggered architecture," *IEEE Proc.*, 2003.

[3] T. A. Henzinger *et al.*, "Giotto: a time-triggered language for embedded programming," *IEEE Proc.*, vol. 91, no. 1, 2003.

[4] E. A. Lee *et al.*, "Synchronous data flow," *IEEE Proc.*, vol. 75, no. 9, 1987.

[5] T. A. Henzinger *et al.*, "The embedded systems design challenge," in *FM*, 2006.

[6] E. A. Lee *et al.*, "A framework for comparing models of computation," *TCAD*, vol. 17, no. 12, 2006.

[7] A.-H. Ghamarian *et al.*, "Throughput analysis of synchronous data flow graphs," in *ACSD*, 2006.

[8] S. Stuijk *et al.*, "Scenario-aware dataflow: Modeling, analysis and implementation of dynamic applications," in *SAMOS*, 2011.

[9] A. Benveniste *et al.*, "A unifying view of loosely time-triggered architectures," in *EMSOFT*, ser. EMSOFT, 2010.

[10] J. Eker *et al.*, "Taming heterogeneity-the ptolemy approach," *IEEE Proc.*, vol. 91, no. 1, 2003.

[11] P. Arumi and X. Amatriain, "Time-triggered Static Schedulable Dataflows for Multimedia Systems," *SPIE The International Society for Optical Engineering*, vol. 7253, no. 1, 2009.

[12] H. I. Ali *et al.*, "Generalized extraction of real-time parameters for homogeneous synchronous dataflow graphs," in *PDP*, 2015.

[13] E. A. Lee *et al.*, "Dataflow," in *System Design, Modeling, and Simulation using Ptolemy II*, 2014.

[14] A. Benveniste *et al.*, "A unifying view of loosely time-triggered architectures," in *EMSOFT*, 2010.

[15] A. H. Ghamarian *et al.*, "Latency minimization for synchronous data flow graphs," in *DSD*, 2007.