A Refinement Theory for Timed-Dataflow Analysis with Support for Reordering

Joost P.H.M. Hausmans [§]
joost.hausmans@utwente.nl

Marco J.G. Bekooij [§] [¶]
marco.bekooij@nxp.com

[§] University of Twente, Enschede, The Netherlands
[¶] NXP Semiconductors, Eindhoven, The Netherlands

ABSTRACT

Real-time stream processing applications executed on embedded multiprocessor systems often have strict throughput and latency constraints. Violating these constraints is undesired and temporal analysis methods are therefore used to prevent such violations. These analysis methods use abstractions of the analyzed applications to simplify their temporal analysis.

Refinement theories have enabled the creation of deterministic abstractions of stream processing applications that are executed on multiprocessor systems. Prominent examples of such abstract models are deterministic timed-dataflow models which can be efficiently analyzed because they only have one behavior.

An important aspect of a stream processing application can be that it makes use of reordered data streams between tasks. An example is the bit-reversed ordered stream produced by a Fast Fourier Transform (FFT) task. However, existing abstraction/refinement theories do not support such reordering behavior or do not handle this type of behavior correctly. This is because existing refinement theories assume that the temporal behavior of applications is orthogonal to their functional behavior, whereas this orthogonality does not always hold in the case of reordered data streams.

In this paper we introduce a new refinement theory in which the potential interaction between temporal and functional behavior is taken into account. The introduced theory supports reordering of data and can therefore be used to validate existing systems with such reordering. Furthermore, the theory enables showing that deterministic dataflow models that do not apply reordering can be used as valid abstractions of systems in which reordering takes place.

The applicability of the refinement theory is demonstrated by creating deterministic timed-dataflow model abstractions of a Digital Video Broadcasting Terrestrial (DVB-T) application, and a communication network in which data is reordered. With these dataflow models the guaranteed throughput and buffer capacities of implementation options are compared.

EMSOFT'16, October 01-07, 2016, Pittsburgh, PA, USA
© 2016 ACM. ISBN 978-1-4503-4485-2/16/10…$15.00
DOI: http://dx.doi.org/10.1145/2968478.2968489

Categories and Subject Descriptors

C.4 [**PERFORMANCE OF SYSTEMS**]: Modeling techniques

General Terms

Performance, Theory, Verification

Keywords

Abstraction-Refinement Theory, Timed-Dataflow

1. INTRODUCTION

The design of modern real-time stream processing applications requires the use of temporal analysis methods to ensure that they meet their temporal constraints. Such analysis methods rely on temporally conservative abstractions of the application. Timed-dataflow models can be used as conservative abstractions. In these models actors introduce a delay [22, 26]. Refinement theories are used to ensure the conservativeness of abstractions. This conservativeness guarantees that the arrival times of data are overapproximated.

In current stream processing applications reordering of data can occur. For example, a buffer can be shared between concurrent writing tasks. The interleaving of such tasks is schedule dependent. Therefore also the order in which data is stored in the buffer depends on the schedule of the tasks. Another example is that the tasks do not produce data in the order that it is consumed. An example of such a task is one that performs a Fast Fourier Transform (FFT) and which produces data samples in a bit-reversed order while these samples are consumed in-order.

Implementations are made robust against this reordering. This is achieved by making use of indices that are associated with the data samples. Based on these indices the data can be retrieved from buffers in the required order [3].

Next to the implementation also the used refinement theory should treat this reordering correctly. However, this is not the case for existing refinement theories. Either they exclude reordering [14, 26] or they assume the temporal and functional behavior of tasks to be orthogonal [10, 23, 24]. This orthogonality does not always hold because an in-time arrival of data does not necessarily imply that values arrive in the required order.

This is illustrated with the example in Figure 1 which shows the response of two different components A and A' on a certain input. When only the times of events are considered it seems as if component A' generates an earlier output than component A. However, when we look at the indices of the output events we notice that A' produces the output events

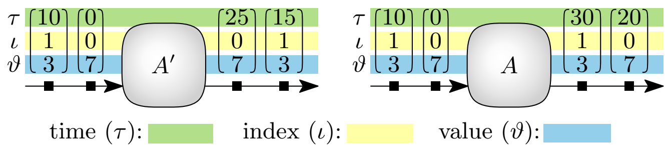

Figure 1: Two components illustrating that reordering couples functional and temporal behavior.

in a different order than A. Consider a component that consumes the produced events. When such a component requires the events in-order (first index 0), suddenly component A' does not generate an earlier output. Component A' is thus not a valid refinement of component A.

Note that the order of the output events can be important. Consider a consuming component which subtracts the values of subsequently arriving input data. If the events are read in the order of the indices then the result is equal for both components A and A' ($7 - 3 = 4$ for these two events). However, if data is read in the order of arrival then the result with component A would be different than the result with component A' ($3 - 7 = -4$).

Refinement theories should thus take the indexed order of event streams into account to support reordering correctly. However, existing refinement theories that do not exclude reordering, assume the temporal and functional behavior to be orthogonal, which prevents to use an indexed order of events.

In this paper we introduce a new refinement theory in which the functional behavior as well as the temporal behavior are taken into account. The introduced theory uses events that consist of a timestamp, an index and a value. Events in streams are related by matching their indices. This allows us to reason about consequences of reordering of events. Furthermore, we introduce a reorder-back component and the concept of receptivity which enables creation of in-order abstractions, of systems in which reordering takes place. We furthermore present a simplified refinement relation for a practically relevant subclass of components. This subclass allows to abstract from data-value-dependent input and output behavior, multiple inputs and outputs, and reordered input and output streams. The applicability of the approach is demonstrated by creating deterministic timed-dataflow model abstractions of a DVB-T receiver application, and a communication network in which reordering takes place. The timed-dataflow model can be used to compute the guaranteed throughput and minimum buffer capacities.

The organization of this paper is as follows. In Section 2 we discuss related refinement theories. In Section 3 we present the basic idea and intuition behind our approach which is formalized and evaluated in the remaining sections. In Section 4 we formalize the component model. Section 5 presents refinement of streams and components and discusses preservation of refinement when components are composed. We also define receptivity of ports and reorder-back components. In Section 6 we present the simplified refinement relation. The two case studies are presented in Sections 7 and the conclusions are stated in Section 8.

2. RELATED WORK

In this section we discuss existing refinement theories and their relation to ours.

The concept of creating deterministic abstractions of run-time scheduled systems with non-deterministic temporal behavior is introduced in [25, 26]. Deterministic timed-dataflow models are used as a temporally conservative abstraction of applications. The idea is that finish times of actors in the dataflow model are later than all possible finish times of the corresponding tasks in the application. However, a transitive refinement relation is not introduced in [25, 26]. Such a transitive refinement relation is required to have multiple levels of abstraction. Furthermore, this method does not support reordering of data.

In [14] a transitive refinement relation is introduced that considers both functional and temporal behavior of components. The inclusion of functional behavior enables to relate events of different abstraction layers. Based on this relation temporally conservative abstractions are defined. However, all event streams are compared by means of production times of events, which prevents a correct handling of reordering.

Another refinement relation is presented in [10]. This refinement relation allows to relate the non-deterministic temporal behavior of applications to the deterministic temporal behavior of analysis models. Such a relation can be established by making use of the the-earlier-the-better principle: a component is “better” when its production moments are earlier. An example of non-deterministic behavior is a varying execution time of a task in the application which can be deterministically abstracted in a dataflow model by using the Worst-Case Execution Time (WCET) of that task. The refinement relation in [10] does not forbid reordering of events. However, the assumption of the theory is that the temporal and functional behavior of abstractions is orthogonal. Therefore events only consist of timestamps, which prevents to distinguish a stream that produces events in-order and a stream that produces events at exactly the same timestamps but in which certain events are swapped and thus reordered. In contrast to [10], an event is in our paper a triple of time, index, and value. Therefore, our component model is an instance of the general tagged-signal model [20]. Because of the index in the triple, the dependencies between components remain unchanged even if events are reordered in time.

A refinement theory for modular real-time systems that are modeled as acyclic graphs of components is presented in [23]. This theory characterizes traffic and provided/remaining service by using arrival and service curves. An important difference with our approach is that we describe event arrivals in the time domain instead of the time-interval domain such that the correlation between events is maintained. This correlation improves accuracy and enables the analysis of cyclic graphs [15, 17]. Resource usage and resource sharing are not considered in our refinement approach, but the effects of resource sharing can be taken into account in the delays that components introduce between input and output streams. The delays for each component can be computed in isolation in case only schedulers from the class of starvation-free schedulers are applied [25, 26]. An example of a starvation-free scheduler is a Time-Division Multiplexing (TDM) scheduler. The delays can be computed using an iterative approach in case fixed priority preemptive scheduling is applied as described in [15, 17]. The approach in [23] considers only temporal behavior and therefore implicitly assumes that functional behavior is orthogonal and that no reordering of events takes place.

The work on relational interfaces [24] forms a basis for many refinement theories. The relational interfaces specify the relations between inputs and outputs of synchronous components. Such synchronous components communicate by assigning variables that are synchronized at the end of a synchronous round. The length of such a synchronous round forms a notion of implicit time. The refinement theory presented in this paper is suitable for components that synchronize on the arrival of events instead of such synchronous rounds. The arrival and production times of these events form an explicit notion of time. The functional and temporal behavior of the components in this paper can be non-deterministic and is described by relations.

There are a number of other Model of Computations (MoCs) that allow reordering of events. An example is the PTIDES MoC [27] which allows the handling of events in a different order than their time-stamp order. A difference is that our approach does not make use of time-stamps in the implementation and only requires the use of indices where reordering can take place.

3. BASIC IDEA

In this section we present the basic ideas behind our approach. First we discuss the relevance of a refinement theory for temporal analysis, after which we provide the intuition behind our refinement theory. A formal description of the theory is presented in Section 4.

3.1 Background

Timed-dataflow models [25, 26] are used to compute the guaranteed throughput of stream processing applications which are executed on run-time scheduled multiprocessor systems. These dataflow models are functionally and temporally deterministic because only actors with constant firing durations and sequential firing rules [19] are applied. Next to that, actors can only communicate using FIFO queues. The constant firing durations prevent reordering of tokens because tokens cannot overtake each other inside actors [1]. The sequential firing rules guarantee that the functional behavior of actors is independent of the arrival times of tokens. Efficient throughput and buffer size analysis techniques [5] have been defined for these deterministic dataflow models. Using the refinement theory of Hausmans [14] it can be shown that the guaranteed throughput of an application executed on predictable multiprocessor systems [8, 13] can be computed using such dataflow models, despite varying execution times of tasks as well as scheduling anomalies.

Unfortunately the refinement theory of Hausmans [14] is not applicable for applications and multiprocessor systems in which reordering of data occurs. Therefore it is currently not possible to derive the worst-case temporal behavior of such systems using timed-dataflow models.

In this work we present a new refinement theory which modifies and extends [10], such that the interaction between temporal and functional behavior as a result of reordering can be taken into account. For space reasons mainly the modifications and additions are presented in this paper. In the next subsection we present the intuition behind our refinement theory.

3.2 Intuitive description of refinement theory

To be able to reason about the interaction between temporal and functional behavior we associate each event with an index, a value and a timestamp. Events are part of event-streams. Each event is stored at a location in such a stream. We define that one stream a' refines another stream a when all events in these streams having the same index contain the same value and the timestamps in a' are not later than in a. This refinement is denoted as $a' \sqsubseteq a$. A variable x can also denote a trace, which is a set of streams. We define that $x' \sqsubseteq x$ in case x' and x consist of an equal number of streams

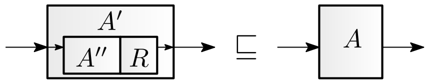

Figure 2: The serial composition of A'' and R results in a component A' that produces an in-order output trace.

and each individual stream that belongs to x is refined. The set of traces over the set of ports P of a component is denoted by $Tr(P)$. By adding a top and a bottom to $Tr(P)$ the lattice $(Tr(P), \sqsubseteq)$ is obtained.

Given the refinement of traces we define the refinement of components. The idea of component refinement is that if a refined component is used as a substitute of an abstract component then this does not result in later production moments of data. Components are similar to dynamic dataflow actors [19], but without a notion of firing rules. Components consume and produce traces and introduce a delay between the arrival of events in an input trace and the arrival of events in an output trace. We define that a component A' refines a component A if for every valid input trace of A and A' we have that A can produce events later than A'. Besides a later trace, component A can produce output traces that are earlier given the same input trace because the behavior of A can be non-deterministic. However, there must be at least one possible trace that A can produce whose events are never earlier than in all the possible traces that can be produced by A'.

The behavior of components is described by relations. If a is an input trace of component A and b a corresponding output trace then we write aAb. In our theory the components in the abstraction are required to be monotone (more precisely $\sqsubseteq$-monotone and $\sqsubseteq$-continuous). A component A is $\sqsubseteq$-monotone if for all traces a, a' with $a' \sqsubseteq a$ and $a'Ab'$, there also exists a b for which holds that aAb and $b' \sqsubseteq b$, or in words, an earlier input trace may not result in the fact that A can produce a later output trace. That refined components are not required to be monotone is needed to be able to make abstractions of systems with run-time scheduled tasks in which scheduling anomalies [11] occur.

We can lift the refinement of individual components to the refinement of graphs of components by proving that refinement is preserved by serial, parallel, and feedback composition. The number of components in an arbitrary graph of components can be reduced to a single component. This can be achieved by repetitive application of serial and parallel composition of two components, after which one component is obtained with potentially a feedback loop from its outputs to its inputs. Then this component can be compared to an abstract version of the component by making use of the fact that if two components refine each other without feedback loop, they also refine each other with feedback loop.

The proofs of refinement under serial composition, parallel composition, and given feedback loops are kept as similar as possible to the proofs in [10] by a careful redefinition of streams and the refinement of streams. There is however an important difference between our refinement theory and the refinement theory in [10]. The theory of [10] requires that if component B consumes a trace produced by component A that then an output trace of the refined component A' is a valid input trace for the component B. This requirement is not fulfilled for components that produce reordered streams in a trace and that are abstracted by components that do not allow reordered streams as input. To resolve this issue we introduce a so-called reorder-back component which produces by construction an in-order stream by delaying events. Such a reorder-back component R is placed at the outputs of component A'' and this serial composition results in a new component A' as shown in Figure 2. Furthermore, it is shown in [10] that the serial composition of two monotone components is a monotone component and therefore A' is monotone if A'' and R are monotone. In Section 5.3 it is shown that R is monotone.

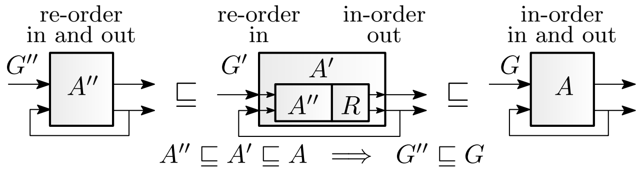

Figure 3: Creation of an abstract component model G without reordering.

We can show that a timed-dataflow model, e.g. a Synchronous Dataflow (SDF) model, in which no reordering takes place, can be a valid abstraction of a system in which components can produce reordered streams. This is shown by making use of an intermediate abstraction layer which consists of components that have internally reorder-back components R, as shown in Figure 3. By showing that each component A' refines A and A'' refines A' we conclude that G'' refines G' which refines G. By using the fact that refinement is transitive we conclude that G'' refines G, although in G'' streams can be reordered whereas the streams in G are in order.

The output of component A' in Figure 3 can be offered as input to component A'' inside A' because we consider the set of streams in which events are in-order a subset of the set of streams in which events are potentially reordered. The output of A' can also be offered as input to a component B which only accepts in-order streams. We define the concept of receptivity to indicate that the output stream of A' is accepted as input stream of B. By making use of this concept of receptivity we allow that $A'' \sqsubseteq A'$, despite that an in-order stream is produced by A' and a reordered stream is produced by A''. Therefore the usual requirement that a refined component must have a more restrictive output guarantee than the abstract component [6], is not satisfied.

However, the components in the intermediate layer G' in Figure 3 must be monotone. Otherwise, it cannot be shown that an abstract graph G, in which events do not reorder, can be created from a graph G'' in which events are potentially reordered. Therefore it must hold that the component A'' is monotone. This can be achieved by creating another layer in which we introduce monotonic components A'' which are valid abstractions of non-monotone components A'''. The components A'' and A''' can both reorder events.

These intermediate levels are modeling constructs that do not have to be implemented. They enable to create a deterministic abstraction G of systems in which the components have a non-monotone behavior, introduce a non-deterministic delay between arrival and production of events, and in which the components can produce and consume reordered streams. The deterministic abstraction G can be a timed-dataflow model that produces and consumes streams in FIFO order. These dataflow models can be used to derive the guaranteed throughput and minimum buffer sizes with existing algorithms if G is an analyzable model such as a dynamic dataflow model like SADF [9] and BPDF [2] or a static dataflow model like CSDF [4] and SDF [18].

4. TIMED COMPONENT MODEL

In this section we first give a formal definition of streams which is then used to formalize the refinement of timed components. We then prove that component refinement is preserved by serial, parallel, and feedback composition. Furthermore we introduce receptivity of ports and define the behavior of reorder-back components.

4.1 Streams

Events in a stream consist of a value, an index, and the production time of the event in the form of a timestamp. Streams are ordered using the timestamps of the events and different streams are compared by matching the indices of events. This allows to correctly compare streams with different orderings of events.

We use a continuous time domain $\mathcal{T}$ with an ordering $\leq$ and a minimal element 0. We also use a maximal element ∞ such that $\forall_{t \in \mathcal{T}} : t < \infty$. We use $\mathcal{T}^\infty = \mathcal{T} \cup \{\infty\}$. Next to that the values of events are chosen in a domain $\mathcal{O}$.

A stream is a total mapping $x : \mathbb{N} \to \mathcal{T}^\infty \times \mathbb{N} \times \mathcal{O}$. For a stream x we use $\tau_x : \mathbb{N} \to \mathcal{T}^\infty$, $\iota_x : \mathbb{N} \to \mathbb{N}$ and $\vartheta_x : \mathbb{N} \to \mathcal{O}$ to retrieve the timestamp, index and value of an event, respectively.

The considered streams are weakly monotone, i.e., for each stream x holds that $\forall_{k,l} : k \leq l \implies \tau_x(k) \leq \tau_x(l)$. Furthermore, each event in a stream x has a unique index: $\forall_{k,l} : k \neq l \implies \iota_x(k) \neq \iota_x(l)$.

Streams are formally defined as an infinite sequence of events. However, they can also be seen as finite. We define an event at location n to be absent when $\tau(n) = \infty$ which given monotonicity of streams implies that also later events are absent. The length of a stream can then be defined as $|x| = \min\{n \mid \tau_x(n) = \infty\}$. We denote with $\overline{\tau}_a(k)$ the timestamp of the event in stream a with index k, i.e., $\overline{\tau}_a(k) = \tau_a(i)$ with $\iota_a(i) = k$. The prefix, earlier-than, and refinement ordering relation for streams are defined as follows:

$$a' \preceq a \;\equiv\; |a'| \leq |a| \wedge \forall_{i<|a'|} : \; \tau_{a'}(i) = \tau_a(i) \wedge \iota_{a'}(i) = \iota_a(i) \tag{1}$$

$$a' \leq a \;\equiv\; |a'| = |a| \wedge \forall_{i<|a|} : \; \overline{\tau}_{a'}(\iota_a(i)) \leq \tau_a(i) \tag{2}$$

$$a' \sqsubseteq a \;\equiv\; \forall_i : \; \overline{\tau}_{a'}(\iota_a(i)) \leq \tau_a(i) \tag{3}$$

A stream ordering relation $\trianglelefteq$ can be lifted to traces $x', x \in Tr(P)$ as follows:

$$x' \trianglelefteq x \equiv \forall_{p \in P} : \; x'(p) \trianglelefteq x(p)$$

The presented stream/trace relations have the same properties as in [10], which implies that we can re-use the results of [10] that are based on these properties. The set of traces $Tr(P)$ forms a lattice with the $\sqsubseteq$-relation. The streams $\vec{0}$ and ϵ on all ports are the least and greatest elements, respectively. In $\vec{0}$, all timestamps are equal to 0 and in ϵ they are equal to ∞.

Next to refinement of timestamps also the values in refined and abstract streams should correspond. This can be checked separately. Similar to $\overline{\tau}$ we use $\overline{\vartheta}_a(k)$ for the value of the event in stream a with index k. For value correspondence in refined streams it can for example be defined that the values are required to be equal:

$$\forall_{i<|a|} : \; \overline{\vartheta}_{a'}(\iota_a(i)) = \vartheta_a(i) \tag{4}$$

Furthermore, in this paper the set of in-order streams is used which is a subset of reordered streams and is defined as follows:

Definition 1 (In-order stream). *A stream a is in-order if $\forall_i : \iota_a(i) = i$.*

4.2 Timed Components

The temporal and functional behavior of a component A is defined by the relation R_A between its input ports and output ports. Unlike in [10] we use the term component in this paper instead of actor to prevent confusion with actors from the timed-dataflow theory, of which SDF actors are an example. A component is defined as follows:

Definition 2 (Component). *A component is a tuple $A = (P, Q, R_A)$ with a set P of input ports, a set Q of output ports and $R_A \subseteq Tr(P) \times Tr(Q)$ with $Tr(P)$ and $Tr(Q)$ the set of all possible traces over the ports P and Q. We use xAy to denote $(x, y) \in R_A$ with $x \in in_A$ and $in_A = \{x \in Tr(P) \mid \exists y \in Tr(Q) : xAy\}$.*

A component A is $\sqsubseteq$-monotone if $\forall_{x,x' \in in_A, y'} : x' \sqsubseteq x \wedge x'Ay' \implies \exists y : xAy \wedge y' \sqsubseteq y$. Furthermore, a component A is $\sqsubseteq$-continuous, if for every pair of ordered sets of traces $\{x_k\}$ and $\{y_k\}$ with respect to $(Tr(P), \sqsubseteq)$ and $(Tr(Q), \sqsubseteq)$ for which it is the case that $x_k A y_k$, it holds that $(\bigsqcup_{\sqsubseteq}\{x^k\})A(\bigsqcup_{\sqsubseteq}\{y^k\})$.

4.3 Component Composition

Component interfaces can be composed to yield new component interfaces. These interfaces are defined as follows for parallel, serial, and feedback composition.

Definition 3 (Parallel Composition). *Let A and B be two components with disjoint sets of input ports P_A and P_B and disjoint sets of output ports Q_A and Q_B, respectively. Then the parallel composition $A||B$ is a component with input ports $P_A \cup P_B$, output ports $Q_A \cup Q_B$, and relation $R_{A||B} = \{(x_1 \cup x_2, y_1 \cup y_2) \mid x_1 A y_1 \wedge x_2 B y_2\}$.*

Definition 4 (Serial Composition). *Let A and B be two components with P_A and P_B the disjoint sets of input ports and Q_A and Q_B the disjoint sets of output ports, respectively. We have that Θ is a bijective function from the output ports of A, Q_A, to the input ports of B, P_B. The serial composition $A\Theta B$ is a component with input ports $P_\Theta = P_A$, output ports $Q_\Theta = Q_A \cup Q_B$ and the relation between input and output ports as follows:*
$R_{A\Theta B} = \{(x, y_1 \cup y_2) \in in_\Theta \times Tr(Q_\Theta) \mid xAy_1 \wedge \Theta(y_1)By_2\}$ with $in_\Theta = \{x \in in_A \mid \forall_y : xAy \implies \Theta(y) \in in_B\}$.

Note that in the definition of in_Θ we use, equal to [10], a "demonic" interpretation of non-determinism which states that an input x is allowed for $A\Theta B$ if any intermediate output of A is a valid input of B.

Next to parallel and serial composition we define feedback composition. We use $x \uparrow p$ for the trace x without the stream that is the input of port p.

Definition 5 (Feedback Composition). *Let A be a component with input ports P_A and output ports Q_A, and let $p \in P_A$ and $q \in Q_A$. The feedback composition of A on (p, q) is the component with input ports $P_{A(p=q)} = P_A \setminus \{p\}$, with output ports $Q_{A(p=q)} = Q_A$ and the relation between input and output ports as follows:*
$R_{A(p=q)} = \{(x \uparrow p, y) \mid xAy \wedge x(p) = y(q)\}$

5. TIMED COMPONENT REFINEMENT

In this section we present the timed component refinement relation and we prove conditions under which refinement is preserved over compositions. After that, we use the concept of receptivity as a weaker condition that still ensures that compositions preserve refinement. This weaker condition is especially important for timed-dataflow abstractions which are not input-complete as we will discuss in the last subsection. In that subsection we also introduce the reorder-back component.

5.1 Refinement

A component A can be replaced by component A' when A' refines A. Using A' instead of A will not worsen the worst-case behavior, i.e., A' may not produce its output data later than A. Component refinement is defined as follows:

Definition 6 (Component refinement). *Component A' refines a component A, i.e., $A' \sqsubseteq A$, if the following two conditions hold:*
(1) $in_A \subseteq in_{A'}$
(2) $\forall_{x \in in_A} \forall_{y'} : xA'y' \implies \exists_y : xAy \wedge y' \sqsubseteq y$

In the following lemmas it is shown that refinement is preserved by parallel, serial and feedback composition.

Lemma 1. *$A' \sqsubseteq A$ and $B' \sqsubseteq B$ implies $A'||B' \sqsubseteq A||B$.*

Proof. Follows trivially from the fact that parallel composition does not involve communication between the components and therefore the refinement, monotonicity, and continuity properties of the components are preserved by parallel composition. □

For refinement preservation of serial composition we intuitively sketch the proof presented in [10].

Lemma 2. *(I) If $A' \sqsubseteq A$ and B is input-complete and $\sqsubseteq$-monotone then $A'\Theta B \sqsubseteq A\Theta B$. (II) If $B' \sqsubseteq B$ then $A\Theta B' \sqsubseteq A\Theta B$.*

Proof. (I) Because $A' \sqsubseteq A$ it is possible to define intermediate traces z' and z, $z' \sqsubseteq z$, which are outputs of A' and A, respectively, and are both valid inputs of B. Because B is $\sqsubseteq$-monotone, we have that $A'\Theta B \sqsubseteq A\Theta B$. (II) Both serial compositions give the same intermediate traces and an intermediate trace that is accepted by B is also accepted by B'. Because $B' \sqsubseteq B$ there always exists an output trace of B which is not earlier than any output trace of B'. □

Given this lemma, it is possible to show that if B is input-complete and $\sqsubseteq$-monotone then $A' \sqsubseteq A \wedge B' \sqsubseteq B \implies A'\Theta B' \sqsubseteq A\Theta B$. This holds only if first the component A is refined and then the component B.

The proof that feedback composition preserves refinement under the conditions provided in [10] is not valid. It is erroneously assumed that an input of A' can be used as an input of A if $A' \sqsubseteq A$. We show that a similar proof as in [10] is valid if A is input-complete, i.e., it accepts all streams. We use $z = x[p \to y(q)]$ to denote that the trace z is equal to trace x in which the stream for port p is replaced by the stream at port q in trace y, i.e., $y(q)$.

Lemma 3. *If component A is input-complete, $\sqsubseteq$-monotone and $\sqsubseteq$-continuous then $A' \sqsubseteq A \implies A'(p = q) \sqsubseteq A(p = q)$.*

Proof. Requirement (1) of Definition 6: Because $A' \sqsubseteq A$ and A is input-complete we have that A' is also input-complete. Because of this also $A'(p = q)$ is input-complete, which implies that $in_{A(p=q)} \subseteq in_{A'(p=q)}$.
Requirement (2): We have to show that, for each allowed input, $A(p = q)$ can produce an output trace that is conservative ($\sqsupseteq$) to all possible output traces of $A'(p = q)$. Let $x \uparrow p \in in_{A(p=q)}$. Then there is an x such that $xA'y'$ with $x(p) = y'(q)$. Furthermore, because A is input-complete and $A' \sqsubseteq A$ there exists an y such that xAy with $y' \sqsubseteq y$. Because $x(p) = y'(q)$ and $y' \sqsubseteq y$ we know $x(p) \sqsubseteq y(q)$, but $x(p)$ is not necessarily equal to $y(q)$. Hence, $x \uparrow pA(p = q)y$ does not have to be a valid behavior. Nevertheless, we can construct a valid behavior $x''Ay''$ with $x'' \uparrow p = x \uparrow p$ and $x''(p) = y''(q)$ as follows. We define for $k \geq 0$ an x^k for which holds that $x^k \uparrow p = x \uparrow p$ and only $x^k(p)$ is modified. We start with $x^0 = x[p \to y(q)]$ and because $x(p) \sqsubseteq y(q)$ we have $x \sqsubseteq x^0$. Component A is $\sqsubseteq$-monotone, thus there exists an y^0 such that x^0Ay^0 and $y \sqsubseteq y^0$. Let us now define $x^{k+1} = x^k[p \to y^k(q)]$. We have $x^{k+1}(p) = y^k(q)$ and that if $x^k(p) \sqsubseteq y^k(q)$ then $x^k \sqsubseteq x^{k+1}$. If this is the case we have $x^{k+1}Ay^{k+1}$ with $y^k \sqsubseteq y^{k+1}$ given the $\sqsubseteq$-monotonicity of A. Then also $y^k(q) \sqsubseteq y^{k+1}(q)$ and thus $x^{k+1}(p) \sqsubseteq y^{k+1}(q)$ from which we can derive all the above properties for x^{k+2} and y^{k+2}. We thus have $\forall_{k\geq 0} : x \sqsubseteq x^k \ \wedge \ y' \sqsubseteq y \sqsubseteq y^k$ and given $\sqsubseteq$-continuity of A we know $x'' = \bigsqcup_{\sqsubseteq}\{x^k\}$, $y'' = \bigsqcup_{\sqsubseteq}\{y^k\}$ and $x''Ay''$. Then $x'' \uparrow p = x \uparrow p$, $y' \sqsubseteq y''$ and $x''(p) = y''(q)$. $\square$

5.2 Receptivity

We use receptivity of ports as another property for which component compositions preserve component refinement. We first define receptivity of ports and components. Then we prove that component refinement is preserved by serial composition and feedback composition if ports/components are receptive.

Definition 7 (Port receptivity)**.** *Port q of component B is receptive to port p of component A when $\forall_{x\in in_A}\forall_y : xAy \implies \exists_{s\in in_B} : s(q) = y(p)$.*

Definition 8 (Component receptivity)**.** *Component B is receptive to component A in the serial composition $A\Theta B$ when all connected ports are receptive, i.e., $\forall_{x\in in_A,y} : xAy \implies \Theta(y) \in in_B$. We then have $in_\Theta = in_A$.*

We now show that serial composition preserves component refinement when components are receptive.

Lemma 4. *When B is receptive to A' and B is $\sqsubseteq$-monotone then $A' \sqsubseteq A \implies A'\Theta B \sqsubseteq A\Theta B$.*

Proof. (1) B receptive to A' so $in_{A'\Theta B} = in_{A'}$ for which holds $in_{A'} \supseteq in_A \supseteq in_{A\Theta B}$. (2) Let $x \in in_{A\Theta B}$ then given property (1), there exists a z' for which we have $xA'z'$ and $\Theta(z')By'$. Furthermore, there exists a z for which xAz and $z' \sqsubseteq z$, due to $A' \sqsubseteq A$. Given the definition of $in_{A\Theta B}$ we know that $\Theta(z) \in in_B$ and because B is $\sqsubseteq$-monotone there exists an y such that $\Theta(z)By$ and $y' \sqsubseteq y$. $\square$

We also show that component refinement is preserved by feedback composition when feedback ports are receptive. We use the receptivity property twice. The first is in Lemma 5 where we use receptivity of port p of A' to port q of A' to show that the allowed input set is refined. This replaces the input-completeness of A' in Lemma 3. The second time that the receptivity property is used is that port p of A is receptive to port q of A' which is used in Lemma 6 instead of input-completeness of component A in Lemma 3.

Lemma 5. *If for feedback composition $A'(p = q)$ we have that port p is receptive to port q then $\forall_{x\in in_{A'}} : (x \uparrow p) \in in_{A'(p=q)}$*

Proof. Straightforward. $\square$

Lemma 6. *If ports p of both components A and A' are receptive to port q of A' and A is $\sqsubseteq$-monotone and $\sqsubseteq$-continuous then $A' \sqsubseteq A \implies A'(p = q) \sqsubseteq A(p = q)$.*

Proof. Requirement (1): From $A' \sqsubseteq A$ follows that $in_A \subseteq in_{A'}$ and given Lemma 5 we know that $in_{A'}$ is not changed, except than that port p is removed from the input set. Therefore also $in_{A(p=q)} \subseteq in_{A'(p=q)}$ holds. Requirement (2): Let $x \uparrow p \in in_{A(p=q)}$. Then there is an x such that $xA'y'$ with $x(p) = y'(q)$. Because port p of A is receptive to port q of A' we know that x with $x(p) = y'(q)$ is also a valid input for A. Because $A' \sqsubseteq A$ there exists an y such that xAy with $y' \sqsubseteq y$. Because $x(p) = y'(q)$ and $y' \sqsubseteq y$ we know $x(p) \sqsubseteq y(q)$. Again $x(p)$ is not necessarily equal to $y(q)$ but we can construct a valid behavior $x''Ay''$ with $x'' \uparrow p = x \uparrow p$ and $x''(p) = y''(q)$ as is shown in the proof of Lemma 3. $\square$

5.3 In-Order Temporal Dataflow Model

In timed-dataflow abstractions all streams must be in-order and actors are in general only monotone when they restrict their inputs to in-order streams. In the last proof of the previous section, receptivity is assumed between abstraction layers and therefore timed-dataflow models cannot directly be valid abstractions for models with reordering. As discussed in Section 3, we introduce reorder-back components in an additional abstraction layer to overcome this shortcoming. A reorder-back component delays the output streams to the earliest possible allowed in-order stream:

Definition 9 (Reorder-back component)**.** *The reorder-back variant of a $\sqsubseteq$-monotone component $A = (P, Q, R_A)$ is defined as $A' = (P, Q, R_{A'})$ with*

$$R_{A'} = \{(x, y) \mid xAy' \wedge$$
$$y = \min\{\bar{y} \mid y' \sqsubseteq \bar{y} \wedge \forall_{i\in\mathbb{N}} : \iota_{\bar{y}}(i) = i\}\}.$$

The reorder-back component is by construction $\sqsubseteq$-monotone because a later input stream can only delay the output stream further. This is a result of that it always produces the earliest allowed output stream.

6. SIMPLIFIED ACTOR REFINEMENT

In the previous sections refinement of components has been defined based on the properties of their input and output streams. However, showing that one component refines another component, requires knowledge about the internals of the component because the properties of the components must hold for every possible input and output stream. Showing that components refine each other can be complicated for a number of reasons. For example, the components can have a data-value-dependent input and output behavior, the components can have multiple input and output streams, and the components can consume and produce reordered streams. In this section we show that for a subclass of components it is possible to abstract from these complicating factors. This is achieved by reasoning in external enabling and finish conditions of components. Using the external enabling and finish condition a simplified refinement condition can be defined. This condition has been used in

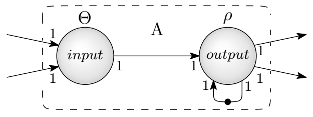

Figure 4: A latency-rate actor component A with latency parameter Θ and rate parameter ρ.

a number of papers [25, 26, 21] to show that an abstract component can be defined of a scheduled task, independent of the input-output behavior of the task. However, the simplified refinement condition differs from the component refinement conditions in Definition 6. We will derive in this section that the simplified refinement conditions are stronger than the component refinement conditions of Definition 6. This allows to conclude that if these simplified conditions hold also the properties from our refinement theory can be applied.

6.1 Simplified refinement condition

The simplified refinement relation is defined in combination with so-called actor components [25]. An actor component contains functionally deterministic dataflow actors [19] that are depicted as nodes and communicate using unbounded FIFO queues depicted as directed edges. An example of an actor component is shown in Figure 4 and is used in [25] as an abstraction of a task. On the internal queues of an actor component the actors produce one token and consume one token per firing. The actors produce these tokens on the internal queues in-order. The values in the tokens that the actors produce are a function of the values in the input tokens. The actors in the components have positive firing durations and the firing durations of the actors are either constant or the actors have a self-edge with one initial token to prevent that tokens can overtake each other inside an actor as a result of auto-concurrency [1]. An actor component has one input actor to which all incoming edges of the component lead and it has one output actor from which all outgoing edges originate, as depicted in Figure 4. An actor component must contain a directed path without initial tokens from its input actor to its output actor. Given such a path and because reordered production of tokens is excluded by construction, the i-th firing (also called execution) of the input actor will result in the i-th firing of the output actor.

An actor component is *externally enabled* when sufficient input tokens have arrived to enable its input actor, excluding inputs produced by actors inside the actor component. The *quantum* of an incoming (outgoing) edge is equal to the number of tokens consumed (produced) from (on) an edge per actor firing. All the quanta in Figure 4 are equal to 1.

By making use of the external enabling moment of the input actor we can abstract from multiple inputs, the consumption quanta, and the consumption order if it is ensured that the refined component uses the same number of inputs, consumption quanta, and consumption order. The production moment of an actor component is defined as the moment at which the output actor finishes its firing. Using the production moment we can abstract from the number of output queues, the production quanta, and also the production order of the tokens if it is guaranteed that they are the same for the refined component.

In [25, 26, 21] actor components have been presented to create a deterministic dataflow abstraction of a task scheduled on a processor. Refinement of these models has been shown by making use of Equation 5.

$$\forall_{i \in \mathbb{N}} : \; e'\langle i \rangle \sqsubseteq e\langle i \rangle \implies f'\langle i \rangle \sqsubseteq f\langle i \rangle \tag{5}$$

In this equation the notation $a'\langle i \rangle \sqsubseteq a\langle i \rangle$ is used for $\forall_{j \le i} : a'(j) \le a(j)$. Therefore Equation 5 states that if all external enabling moments of the refined component are not later than the external enabling moments of an actor component up to and including the i-th enabling that then also the production moments should not be later. Knowledge about external enabling moments of actors with a lower index than i are used in the proofs in e.g. [25, 21] that are based on mathematical induction and that refer to production moments of tokens during previous firings. However, Equation 5 is different from the conditions in Definition 6. Despite this, we show in the following lemma that if Equation 5 holds and under the assumption that $in_A \subseteq in_{A'}$ that then $A' \sqsubseteq A$.

Lemma 7. *If $\forall_{i \in \mathbb{N}} : \; e'\langle i \rangle \sqsubseteq e\langle i \rangle \implies f'\langle i \rangle \sqsubseteq f\langle i \rangle$ holds and the actors in the component A are deterministic then $A' \sqsubseteq A$.*

Proof. The actor component A is deterministic so there will be one stream f of finish moments of the output actor for a stream e of external enabling moments of the input actor. Next to that we have that $e'A'f'$ and eAf. Equation 5 can also be written as:

$$\forall_{e' \in in_{A'}} \forall_{e \in in_A} \forall_{f'} : e' \sqsubseteq e \wedge e'A'f' \implies eAf \wedge f' \sqsubseteq f$$

If $e' = e$ then $e' \sqsubseteq e$ holds and therefore the previous equation can be rewritten into:

$$\forall_{e \in in_A} \forall_{f'} : eA'f' \implies eAf \wedge f' \sqsubseteq f$$

Which implies $A' \sqsubseteq A$. □

7. CASE-STUDIES

In this section we apply our refinement theory to create an abstract (C)SDF model of a DVB-T channel decoder application and a communication network in which reordering takes place. These models can be used to compute the minimum FIFO capacities that are needed to satisfy a throughput constraint or to prevent deadlock.

7.1 DVB-T channel decoder

The task graph of a DVB-T channel decoder application is shown in Figure 5. The decoder is a functionally deterministic stream-processing application that produces one unique output stream given an input stream. The DVB-T application is a real-time application because it has to process a periodic data stream from an Analog-to-Digital Converter (ADC) that runs at a fixed frequency. The application is executed on a Software-Defined Radio (SDR) multiprocessor platform in which tasks execute data-driven and have non-constant execution times. Each task is executed on its own processor and tasks cannot start their execution when there is insufficient space in their output buffers.

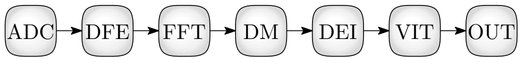

Figure 5: Task graph of a DVB-T decoder application.

All tasks in Figure 5 produce a stream that is consumed in the same order by the subsequent task, except for the Fast Fourier Transform (FFT) task. To demonstrate our refinement theory we assume that a pipelined FFT [12] is used which produces a stream in a bit-reversed order while the De-Mapper (DM) task consumes samples in-order. Therefore all communication buffers between the blocks can be FIFO buffers, except for the buffer between the FFT and the DM task which uses indices to preserve dependencies. All task are described as sequential programs, like in a Kahn Process Network (KPN) [16], and use reads and writes that block until the requested data/space becomes available.

Sufficiently large buffer capacities given a throughput constraint can be computed using a temporally and functionally deterministic SDF model of the decoder application. This SDF model should take into account that the FFT task produces samples in a different order than the DM task consumes the samples. Therefore we will focus on this communication between the FFT and DM tasks in the subsequent paragraphs.

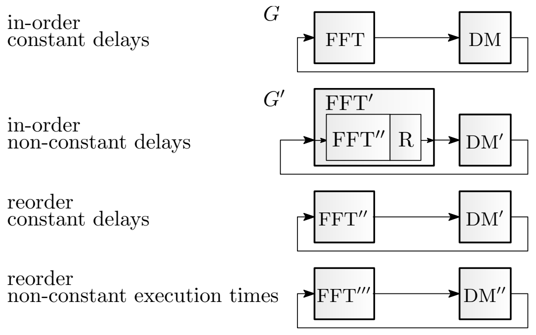

Figure 6: Component models of the FFT/DM subgraph.

To create a suitable SDF model we introduce two intermediate abstraction layers between the task graph and the dataflow graph G, as shown in Figure 6. At the first intermediate level varying execution times are replaced by constant delays that are equal to the Worst-Case Execution Times (WCETs) of the tasks such that a $\sqsubseteq$-monotone component model is obtained. Adding the reorder block R inside the FFT$'$ component results in an in-order stream. However, as discussed below, the delay of the FFT$'$ component is non-constant as a result of the reorder component.

The top-level model in Figure 6 is a valid abstraction if the components refine each other at lower abstraction levels. That the FFT component is refined at lower abstraction levels is shown as follows. We will assume for ease of understanding that a 4-point FFT is used instead of the 2048-point FFT in the real application. The location (i) and index (ι) of the input and output stream of the FFT$'''$ component are shown in Figure 7 as s_1 and s_2 respectively. This figure shows that the FFT component needs 3 input samples before it produces the first output sample. From the 4-th sample onwards it consumes and produces every execution a sample. The produced stream is not in-order. After adding a reorder component, the stream s_3 is obtained in which the production of some of the samples is delayed one execution compared to s_2.

To conclude refinement of the graph G' by G in Figure 6 it must be shown that the FFT$'$ and DM$'$ components are refinements of the corresponding FFT and DM SDF actors, despite that the FFT$'''$ component produces a reordered stream.

The behavior of the described FFT$'$ component cannot be modeled with an SDF actor because the FFT$'$ component has an aperiodic behavior. However, a conservative, i.e., pessimistic, abstraction in the form of the SDF model in Figure 8 can be created by making use of a negative number of initial tokens [7]. A negative number of initial tokens delays the enabling of an actor because more tokens need to be produced before the actor is enabled. The resulting input stream of the DM actor is s_4 in Figure 7 in which the first sample is one FFT execution delayed compared to s_3. The number of tokens in the SDF model is in this example -4 of which -3 are due to the pipelined behavior of the FFT task and -1 is added to obtain a stream in which one token is produced every execution after an initial phase.

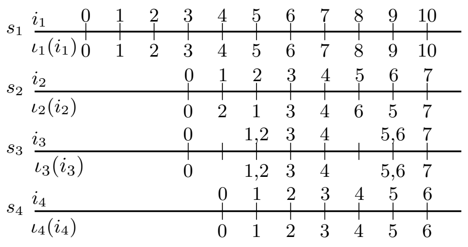

Figure 7: Input and output streams of the FFT component.

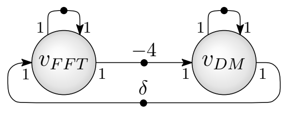

Figure 8: SDF model of the FFT and DM components.

Using the SDF model in Figure 8 we can compute buffer capacities. In this figure there are δ initial tokens on the edge (v_{DM}, v_{FFT}). To provide some additional insight we will explain the derivation of the minimum buffer capacity for deadlock-free execution. For deadlock-free execution we can make use of the fact that there should be at least one initial token on every cycle in an Homogeneous Synchronous Dataflow (HSDF) graph and that the considered subgraph in Figure 8 is actually an HSDF graph. Using this rule we find that at least 5 initial tokens are needed. From this we conclude that the buffer capacity should be at least 5 samples, because -4 initial tokens were used to model the effect that the FFT does initially not produce data and does also not produce the samples in order. Therefore we conclude that the capacity of the buffer should be 1 sample larger compared to the case that the FFT task would produce data in-order.

The refinement theory is also applicable for components that consume reordered streams. For example, also pipelined FFTs exist that consume data in a bit-reversed order and produce in-order streams. In this case samples need to be reordered in the buffer between the DFE and the FFT components because the DFE produces an in-order stream. As abstraction, an FFT SDF actor can be created that consumes in-order streams. In the SDF model there are -3 initial tokens introduced for modeling that the FFT actor produces tokens after the 4-th token is produced by the DFE. Therefore only 4 initial tokens are needed for deadlock-free execution instead of the 5 tokens in the previous example.

7.2 Communication network

In this subsection we consider the communication network depicted in Figure 9 in which there are two communication paths from a source to a destination for the transfer of data packets. There is also a network connection for the transfer of credits from the consumer C to the producer P, which is not shown in this figure. Which communication path is selected for the transfer of a packet is dependent on the traffic generated by other applications on the communication links of the network and can therefore vary per packet. As a result also the communication latency varies per packet and it can occur that a packet that is later injected in input buffer b_1 by task P arrives earlier in the output buffer b_2. The task C consumes the packets in index order from the buffer b_2 despite that the arrival order of the packets is unknown at design time. As soon as a packet is consumed by C, and removed from the buffer b_2, a credit packet is sent through the network to the producer to indicate that there is space in the buffers b_1 and b_2. The sizing of the buffer b_1 and b_2 is an important design decision because it affects the maximum throughput and the cost of the network. Furthermore we would like to derive the improvement in throughput if two instead of one connections are used.

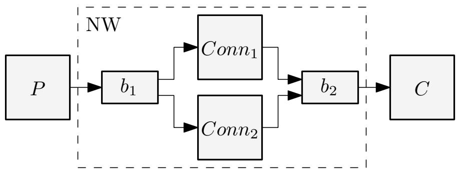

Figure 9: Block diagram of a communication network.

To compute the required buffer sizes a latency-rate model can be created for a communication path in network (NW) if the network provides guaranteed-throughput connections based on resource reservation [13]. Bandwidth for each network link can be reserved by making use of non-preemptive round-robin scheduling per packet in each network router given that the packets have a maximum length. The worst-case temporal behavior of each guaranteed-throughput network connection can then be modeled with a latency-rate model [25].

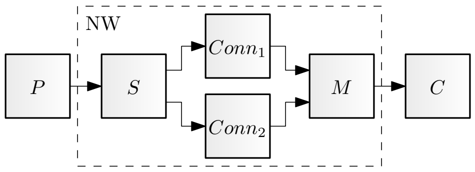

Figure 10: Component model of the communication network.

We first consider the case that inside the network the connection used for a packet transfer is selected randomly. Therefore it is theoretically possible that all packets are transferred by the same connection. A component model for the transfer of data packets through the network is shown in Figure 10. The selection component S in this figure models that a packet is transferred through one of the connections, and a merge component M receives packets from one of the two connections and provides them as input to the consumer component C. The packets arrive in an unknown order because packets can be transferred by different network connections, each introducing a varying, but bounded delay.

For throughput and buffer size analysis a deterministic model should be created in which no reordering occurs. However, unlike in the DVB-T example, the introduction of a more abstract component model in which all components introduce a constant delay does not result in a fixed arrival order of packets because of the non-deterministic selection of communication paths.

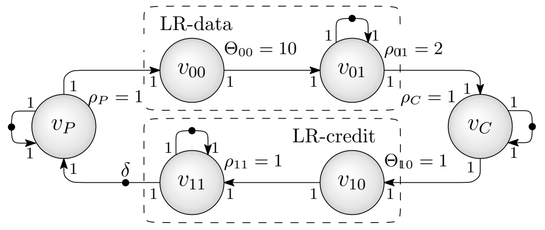

Figure 11: SDF model of the communication network.

An alternative is the creation of a deterministic latency-rate model of the whole network which abstracts from the individual connections, as well as the selection and merging. The two parameters in this latency-rate model are Θ and ρ, of which $1/\rho$ determines the minimum processing-rate and Θ is used to model the maximum communication latency of a packet. To determine a conservative model of the network we can make the assumption that all packets are transferred through the same connection, which causes a maximum delay as a result of queuing. Because it is unknown which connection is selected, we make $1/\rho$ in the latency-rate model equal to the minimum of the processing rates of both connections and set Θ equal to $L - \rho$ with L the maximum of the latencies introduced by the two connections. This results in the dataflow model in Figure 11 in which also the transfer of credits is modeled using a separate latency-rate model. With this model we can determine the number of initial tokens δ which is equal to the minimum required buffer capacity of b_1 as well as b_2.

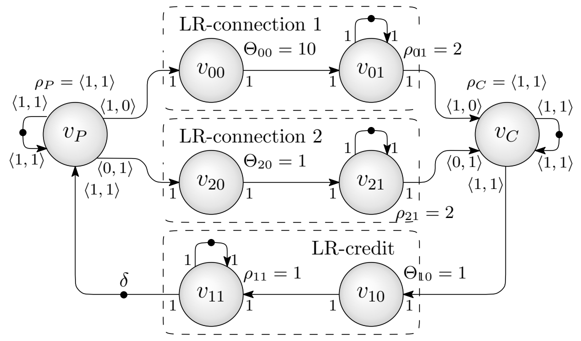

Figure 12: CSDF model of the communication network.

From the model in Figure 11 we can conclude that the use of multiple connections in the network does not improve the guaranteed throughput nor reduces the buffer capacities compared to the use of only one network connection. The guaranteed throughput could be improved if the connection for the transfer of packets are not selected randomly. For example, if packets with an even index are transferred through connection 1 and packets with an odd index through connection 2. This deterministic selection of connections for the transfer of packets does not guarantee that packets arrive in order. However, because packets are consumed in an order that is based on their indices we can abstract from the non-deterministic merge performed by the buffer b_2 and use the deterministic CSDF model in Figure 12. Using this model we can conclude that the guaranteed throughput of the network increases, compared to the previous example, if the capacities of the buffers b_1 and b_2 are made sufficiently large. Otherwise the latency of one of the connections could limit the maximum throughput of the network.

For example, consider that all phases of the actors have a firing duration of $1\,\mu$s, except for the actors v_{00}, v_{01}, and v_{21} which have a firing duration of respectively 10, 2, and $1\,\mu$s. Then with $\delta = 16$ a throughput of $0.5\,\text{token}/\mu$s is obtained. The network with the random connection selection achieves the same throughput given only 8 initial tokens ($\delta = 8$) according to the dataflow model in Figure 11. However, this is also its guaranteed throughput while the network with the alternating connection selection can achieve a guaranteed throughput of $1\,\text{token}/\mu$s according to its CSDF model in case the buffers b_1 and b_2 have a capacity of at least 32 packets.

8. CONCLUSION

In this paper a refinement theory for timed component models is presented that supports components consuming or producing reordered data streams. Using this refinement theory deterministic abstractions can be created in which no reordering occurs, of systems in which components introduce non-deterministic delays, have non-monotone temporal behaviors, and/or consume and produce reordered data streams. Such deterministic abstractions can be used to compute the guaranteed throughput and minimum buffer capacities using existing timed-dataflow analysis techniques.

Refinement is achieved by making use of an intermediate layer in which components are introduced that can reorder streams, but which have a temporally monotone behavior. Furthermore, a notion of receptivity is introduced to indicate that one component accepts an output stream of another component. The concept of receptivity enables that a component which produces a reordered stream can be a refinement of a component that produces an in-order stream.

Proving refinement of components can be difficult due to their complex input-output behavior, which for example includes reordered production of data. We show that for a practically relevant subclass of components it is possible to partially abstract from input-output behavior by reasoning in external enabling and production moments.

The relevance and applicability of the presented refinement theory is illustrated using a DVB-T receiver in which the FFT component produces a reordered stream. Furthermore, it is demonstrated that a deterministic SDF model can be created of a communication network in which packets are reordered as a result of non-deterministic packet selection and merge operations.

References

[1] F. Baccelli et al. *Synchronization and Linearity: an Algebra for Discrete Event Systems*. Wiley, 1992.

[2] V. Bebelis et al. BPDF: A Statically Analyzable Dataflow Model with Integer and Boolean Parameters. In *EMSOFT*, 2013.

[3] T. Bijlsma et al. Circular Buffers with Multiple Overlapping Windows for Cyclic Task Graphs. In *HiPEAC*, 2011.

[4] G. Bilsen et al. Cyclo-Static Dataflow. *IEEE Trans. on Signal Processing*, 44(2):397–408, 1996.

[5] A. Dasdan. Experimental Analysis of the Fastest Optimum Cycle Ratio and Mean Algorithms. *TODAES*, 9(4):385–418, 2004.

[6] L. de Alfaro et al. Interface Theories for Component-Based Design. In *EMSOFT*, pages 148–165. Springer, 2001.

[7] R. de Groote et al. Back to Basics: Homogeneous Representations of Multi-Rate Synchronous Dataflow Graphs. In *MEMOCODE*, pages 35–46, 2013.

[8] B. Dekens et al. Low-Cost Guaranteed-Throughput Communication Ring for Real-Time Streaming MPSoCs. In *DASIP*, pages 239–246, 2013.

[9] M. Geilen et al. Worst-Case Performance Analysis of Synchronous Dataflow Scenarios. In *CODES+ISSS*, pages 125–134, 2010.

[10] M. Geilen et al. The Earlier the Better: A Theory of Timed Actor Interfaces. In *HSCC*, April 2011.

[11] R. Graham. Bounds on the Performance of Scheduling Algorithms. *Computer and job scheduling theory*, pages 165–227, 1976.

[12] H. Groginsky et al. A Pipeline Fast Fourier Transform. *Trans. on Computers*, 100(11):1015–1019, 1970.

[13] A. Hansson et al. Enabling Application-Level Performance Guarantees in Network-Based Systems on Chip by Applying Dataflow Analysis. *Computers & Digital Techniques*, 3(5):398–412, 2009.

[14] J. Hausmans. *Abstractions for Aperiodic Multiprocessor Scheduling of Real-Time Stream Processing Applications.* PhD thesis, University of Twente, Enschede, The Netherlands, April 2015.

[15] J. Hausmans et al. Dataflow Analysis for Multiprocessor Systems with Non-Starvation-Free Schedulers. In *SCOPES*, pages 13–22, 2013.

[16] G. Kahn. The semantics of a Simple Language for Parallel Programming. In *Proc. of the IFIP Congress*, volume 74, pages 471–475, 1974.

[17] P. Kurtin et al. Combining Offsets with Precedence Constraints to Improve Temporal Analysis of Cyclic Real-Time Streaming Applications. In *RTAS*, 2016.

[18] E. Lee et al. Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing. *IEEE Trans. on computers*, 36(1):24–35, 1987.

[19] E. Lee et al. Dataflow Process Networks. *Proc. of the IEEE*, 83(5):773–801, 1995.

[20] E. Lee et al. A framework for comparing models of computation. *Trans. on Computer-Aided Des. of Int. Circuits and Syst.*, 17(12):1217–1229, 1998.

[21] A. Lele et al. A New Data Flow Analysis Model for TDM. In *EMSOFT*, pages 237–246, 2012.

[22] S. Sriram and S. Bhattacharyya. *Embedded Multiprocessors: Scheduling and Synchronization.* Marcel Dekker, Inc., 2000.

[23] L. Thiele et al. Real-Time Interfaces for Composing Real-Time Systems. In *EMSOFT*, pages 34–43, 2006.

[24] S. Tripakis et al. A Theory of Synchronous Relational Interfaces. *TOPLAS*, 33(4):14, 2011.

[25] M. Wiggers et al. Modelling Run-Time Arbitration by Latency-Rate Servers in Dataflow Graphs. In *SCOPES*, pages 11–22, 2007.

[26] M. Wiggers et al. Monotonicity and Run-Time Scheduling. In *EMSOFT*, pages 177–186, 2009.

[27] Y. Zhao et al. A programming model for time-synchronized distributed real-time systems. In *RTAS*, pages 259–268, 2007.