# An Abstraction-Refinement Theory for the Analysis and Design of Real-Time Systems

PHILIP S. KURTIN, University of Twente
MARCO J.G. BEKOOIJ, NXP Semiconductors and University of Twente

Component-based and model-based reasonings are key concepts to address the increasing complexity of real-time systems. Bounding abstraction theories allow to create efficiently analyzable models that can be used to give temporal or functional guarantees on non-deterministic and non-monotone implementations. Likewise, bounding refinement theories allow to create implementations that adhere to temporal or functional properties of specification models. For systems in which jitter plays a major role, both best-case and worst-case bounding models are needed.

In this paper we present a bounding abstraction-refinement theory for real-time systems. Compared to the state-of-the-art TETB refinement theory, our theory is less restrictive with respect to the automatic lifting of properties from component to graph level and does not only support temporal worst-case refinement, but evenhandedly temporal and functional, best-case and worst-case abstraction and refinement.

CCS Concepts: • **Theory of computation** → *Streaming models*; *Timed and hybrid models*; • **Computing methodologies** → *Model development and analysis*; • **Computer systems organization** → *Real-time systems*; *Real-time system specification*;

Additional Key Words and Phrases: Denotational & Asynchronous Component Model, Bounding Abstraction & Refinement, Worst-Case & Best-Case Modeling, Real-Time System Analysis & Design, Temporal & Functional Analysis, Discrete-Event Streams, The-Earlier-the-Better, Timed Dataflow

**ACM Reference format:**
Philip S. Kurtin and Marco J.G. Bekooij. 2017. An Abstraction-Refinement Theory for the Analysis and Design of Real-Time Systems. *ACM Trans. Embedd. Comput. Syst.* 16, 5s, Article 173 (September 2017), 20 pages.
DOI: 10.1145/3126507



## 1 INTRODUCTION

To cope with the ever-increasing complexity of computer systems in general and real-time systems in particular, two concepts gained major significance: A component-based reasoning allows to reduce complexity by breaking down complex systems into subproblems, which can be ideally treated in separation. And a model-based reasoning enables to give temporal or functional guarantees on implementations without being exposed to their entire complexity. The process of creating models from given implementations is called abstraction, whereas the process of creating implementations from given specification models is called refinement, with the former being mainly used for system analysis and the latter mainly for system design. Abstraction and refinement theories can be classified as follows: First, we differ between theories that make use of purely temporal, purely functional or both temporal and functional models. Second, we differ between theories whose

This article was presented in the International Conference on Embedded Software 2017 and appears as part of the ESWEEK-TECS special issue.
Author's addresses: P. S. Kurtin and M. J. G. Bekooij, Dept. of Computer Science, P.O. Box 217, 7500 AE Enschede, The Netherlands; emails: philip.kurtin@utwente.nl, marco.bekooij@nxp.com.

models either include all behaviors of an implementation (e.g. use intervals of task execution times instead of the actual execution times) or that bound all behaviors of an implementation (e.g. use worst-case task execution times instead of actual execution times). While both inclusion and bounding can reduce implementation complexity by e.g. creating monotone models for non-monotone implementations, only bounding supports the creation of deterministic models for non-deterministic implementations, which is a prerequisite for the application of many efficient analysis techniques. Lastly, we differ between best-case and worst-case bounding, with the former bounding implementation behaviors from below and the latter from above.

In this paper we present a denotational timed component model whose components relate discrete-event streams between input and output interfaces using mathematical relations. Requirements on components are thereby intentionally chosen as low as possible, in order to allow the expression of most discrete-event systems and models, as well as their combinations. For the timed component model we further provide an abstraction-refinement theory that evenhandedly allows temporal and functional, worst-case and best-case abstraction and refinement. The theory enables to abstract complex, non-deterministic implementations to efficiently analyzable, deterministic models, such that model analysis results are guaranteed to hold for the respective implementations. Likewise, it enables to refine models to implementations, such that implementations are guaranteed to adhere to model specifications. Lastly, we provide proofs that certain properties like temporal and functional bounding are preserved on parallel, serial and feedback compositions. This implies that these properties are automatically lifted from component to graph level, enabling a component-based reasoning without the need for holistic analysis.

To give an example of the application of our theory, consider a non-deterministic, non-monotone implementation consisting of software and hardware components with complex dependencies and scheduling anomalies. To determine the end-to-end latency of such an implementation, we can bound each of the components from above using deterministic timed dataflow components [11, 14]. The dependencies between components can be thereby ignored as bounding is automatically lifted from component to graph level. The resulting abstract timed dataflow model can then be analyzed efficiently [2] and the determined end-to-end latency does not only hold for the model, but is guaranteed to be an upper bound on the end-to-end latency of the implementation as well.

Our theory is mainly inspired by the The-Earlier-the-Better (TETB) theory [7], but generalizes it in multiple ways. While in TETB streams are only temporal, our notion of streams is based on indices, timestamps and values. This enables a consideration of components that produce values out of timestamp order like in [9], as well as both temporal and functional bounding. But the key difference compared to [7, 9] lies in the combination of bounding and input acceptance preservation. In TETB, bounding and input acceptance preservation are inseparably linked via the component refinement relation $\sqsubseteq$: The relation implies that a component $A^{impl}$ only refines a component $A^{wc}$ if it is worst-case temporally bounded by $A^{wc}$, i.e. if component $A^{impl}$ is never slower than the worst-case of $A^{wc}$, and if $A^{impl}$ preserves input acceptance of $A^{wc}$, i.e. $A^{impl}$ accepts at least all inputs that $A^{wc}$ also does. In contrast, our theory separates bounding and input acceptance preservation, which resolves several shortcomings:

First, in system design it is desirable that a refined implementation accepts at least all inputs of a design model, whereas in system analysis it is desirable that an analysis model accepts at least all inputs of an implementation. As depicted on the left side of Figure 1, abstraction and refinement are symmetric in TETB, which implies that an abstraction cannot accept more, but only less inputs than an implementation. This means that in TETB the only way to realize analysis models which hold for all cases of an implementation is to construct them such that they accept exactly the same inputs as the respective implementation. However, if one considers that many implementations

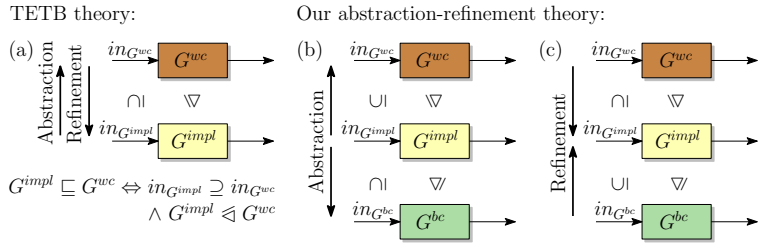


Fig. 1. Input acceptance preservation and bounding.

are input-restricted (such as components that are only laid out for inputs with a certain period and jitter), while many analysis models are by definition input-complete (i.e. accept any inputs, such as dataflow models), it appears that equal input acceptance is an unrealistic assumption in many cases. As depicted on the right side of Figure 1, in our theory we consider input acceptance preservation and bounding separately. This allows for a combination of the two as needed, i.e. refinement for system design with input acceptance preservation from model to implementation, as well as abstraction for system analysis with input acceptance preservation from implementation to model. Effectively, this means that TETB is mainly a refinement theory, while our theory is an abstraction-refinement theory.

Second, if for instance interference effects due to resource sharing or buffer allocation are in the scope of analysis, determining worst-case upper bounds on the temporal behavior of components is not sufficient. Instead, upper bounds on their jitters are needed [17, 18], which additionally requires to construct models that are best-case lower-bounding the temporal behavior of implementations, i.e. no behaviors of an implementation are allowed to be earlier than the earliest behavior of the model. As depicted in Figure 1, TETB with its refinement relation $\sqsubseteq$ only supports worst-case models. We introduce two bounding relations, $\trianglelefteq$ for worst-case bounding and $\vartriangleleft$ for best-case bounding, such that our theory supports both worst-case and best-case models.

And third, in TETB automatic lifting from component to graph level is only discussed with respect to refinement, which implies that input acceptance preservation must already hold on component level. To understand the impact of this requirement, consider two serially connected components, of which the latter is input-complete, but only receives strictly periodic inputs of the former. In TETB, it would be required that any refinement of the latter component were also input-complete, although it would actually suffice if the refinement of the latter accepted the strictly periodic inputs of the former. On top of that, in most cases it does not even suffice that input acceptance preservation holds on component level, but both abstract and refined components must be additionally input-complete. This requires complex workarounds for cases in which e.g. data streams do not arrive in order [9], while for other cases even no such workarounds exist (an example of this can be found in our case study). These implications show that input acceptance preservation on component level is a both severe and unnecessary restriction. Our theory circumvents this restriction by separately discussing lifting of bounding and input acceptance. Consequently, bounding is automatically lifted from component to graph level without any restrictions on component input acceptance.

The remainder of this paper is structured as follows. Section 2 gives an informal description of our timed component model and abstraction-refinement theory and Section 3 presents related work. Section 4 formalizes the denotational timed component model and Section 5 the corresponding abstraction-refinement theory. Section 6 discusses the extended applicability of our theory compared to the TETB refinement theory in a case study and Section 7 finally draws the conclusions.

## 2 INFORMAL DESCRIPTION

In this section we give an informal description of our theory, which serves the purpose of a basic idea and defines the terminology used in the remainder of this paper.

### 2.1 Timed Component Model

In our theory we reason in **streams** that map **indices** to tuples of **timestamps** and **values**. Technically, every stream has an infinite amount of events, but the smallest index from which onwards all timestamps and values are equal to infinity is used to mark the **length of a stream**. Streams are transfered over **ports**, with each port being characterized by a **value domain** that specifies which values the indices of a transfered stream can take, as well as an **ordering relation** that specifies an order on the value domain. Multiple ports form an **interface** and streams being transfered over the ports of an interface form a **trace**.

A **component** consists of an **input interface**, an **output interface** and a **relation** that translates traces on the input interface to traces on the output interface. As the relation of a component does not necessarily have to be a function, a component can also be **non-deterministic**, i.e. it can produce different output traces for the same input trace. We require that all components accept the **empty trace**, a trace consisting of zero length streams (all timestamps and values are infinite), and that the empty trace on the input interface always results in the empty trace on the output interface.

The **input set** of a component is defined by its relation and contains all traces that are accepted by the component. Likewise, the **output set** of a component is also defined by its relation and contains all traces that can be produced.

Components can be **composed** to form new components. Thereby we differ between **parallel compositions** in which the input and output interfaces of the original components are united, **serial compositions** in which (a part of) the output interface of one component is connected to (a part of) the input interface of another, and **feedback compositions** in which (a part of) the output interface is connected to (a part of) the input interface of the same component. Interface connections on serial and feedback compositions thereby adhere to one-to-one **port mappings**.

Composed components have input and output interfaces containing all ports not connected in the respective compositions. Thus we differ between **internal interfaces** whose ports are connected and **external interfaces** that remain unconnected. Consequently, a parallel composition does not have internal interfaces, whereas the external interfaces of serial and feedback compositions contain less ports than the individual components.

While the relations of parallelly composed components remain unchanged (as consequently also the input and output sets), the relations of serially composed components are reduced to the combinations of output traces that can be produced by the first and the input traces that can be accepted by the second component. This implies that any serial composition of two components is valid, however, both the input set and output set of the composition can contain less traces that can be accepted or produced, respectively. For feedback composition, any external input trace is valid if for this input a fixed-point is always reachable, starting from the empty trace on the internal interface. Finally, we call a component that consists of multiple composed components a **component graph**.

### 2.2 Bounding Abstraction & Refinement

Our abstraction-refinement theory is based on two key concepts: **Bounding** and **input acceptance preservation**, which we consider separately in our theory. In this section we focus on the bounding

part. We begin with the bounding of streams, then traces, components and lastly component graphs. Finally, we discuss the relation of bounding to abstraction and refinement.

We say that a stream upper-bounds another stream if for all indices the timestamps are equal or larger and if for all indices the values are equal or larger according to the port-specific ordering relation. Likewise, a trace upper-bounds another trace if all the streams of the former upper-bound all streams of the latter. A lower bound is the reverse of an upper bound.

Based on trace bounding we define two different variants of component bounding. Let it hold for two input traces of two components that the input trace of the first is an upper bound on the second. And let it further hold that there exists a corresponding output trace of the first that is an upper bound on all corresponding output traces of the second. If this holds for any input traces of the two components that are upper-bounding each other we say that the first component is a **worst-case upper bound** on the second (and the second a worst-case lower bound on the first).

Analogously, let it hold for an input trace of one component that is a lower bound on an input trace of another that for these input traces there exists an output trace of the first component that is a lower bound on all output traces of the second. If this holds for any input traces of the two components that are lower-bounding each other we say that the first component is a **best-case lower bound** on the second (and the second a best-case upper bound on the first).

We prove that both worst-case and best-case bounding are preserved on component composition, without any further requirements. This allows us to conclude that two graphs bound each other if all their components bound each other, i.e. bounding is automatically lifted from component to graph level.

Finally we define that a component graph is a **worst-case (best-case) abstraction** of another graph if the first accepts at least all inputs of the second and if for the same inputs the first has at least one output that upper-bounds (lower-bounds) all outputs of the second. Likewise, a graph is a **worst-case (best-case) refinement** of another graph if the first accepts at least all inputs of the second and if for the same inputs the second has at least one output that upper-bounds (lower-bounds) all outputs of the first. We show that bounding abstraction and refinement can be concluded from component level bounding and graph level input acceptance preservation. Lastly, we discuss that bounding abstraction and refinement are transitive.

### 2.3 Input Acceptance Lifting & Replaceability

Unlike the TETB refinement relation, our bounding relations do not imply input acceptance preservation. This generalization comes at the cost that graph level input acceptance preservation, which is needed for abstraction and refinement, cannot be concluded from component level bounding. To address this, we derive properties which imply automatic lifting of input acceptance from component to graph level. If such an automatic lifting is given, one can conclude input acceptance preservation between graphs from input acceptance preservation between individual components.

We say that a component is **input-independent** if it holds for the streams accepted according to the component relation that all combinations of streams are accepted, i.e. if two streams are accepted on one port and two other on another, then all four combinations must be accepted. Moreover, a component is **empty-continuous** if it holds that the relation between input and output traces is continuous with respect to a certain trace ordering relation for which the empty trace is the infimum of all traces.

It can be seen that a graph consisting of components that are both input-independent and empty-continuous is also input-independent and empty-continuous, if the input sets of all connected components are supersets of the respective connected output sets. Based on these properties it can

be further seen that input acceptance is also automatically lifted from component to graph level, i.e. that a graph accepts all inputs that are accepted by its respective components before composition.

Furthermore we discuss that all **input-complete components** (components that accept any inputs) are input-independent, as well as that all **operational components** (components that produce extended outputs for extended inputs) are empty-continuous. This lets us conclude that a graph of input-complete, operational components is also input-complete and operational, which is for instance the case for the important subclass of **timed dataflow models**.

Based on the same properties, conditions can be derived for which components in a graph can be replaced without reducing the input acceptance of the graph. Note that due to space limitations, both input acceptance and replaceability are only discussed informally in this paper. For a formal derivation please refer to [10].

## 3 RELATED WORK

In this section we give an overview of related work, the so-called interface refinement theories. The main scope of interface refinement is the replacement of components with refined components without violating certain graph level properties. In this context, component interfaces are abstractions of the components themselves, containing sufficient information to be representative for the underlying components, but not more information than needed to assert the adherence or violation of graph level properties. With respect to the graph level properties that are to be preserved we divide existing interface refinement theories into three classes.

With **type refinement** [12] it is merely ensured that refined components accept at least the same data types and produce no other data types than abstract components. **Inclusion refinement** subsumes all theories that involve inclusion of behaviors, meaning that abstract components can match at least all behaviors of refined components. Examples of such theories are **language refinement** [12], that is also called trace containment [13], and the stronger **simulation refinement** [12]. Lastly, **bounding refinement** differs from inclusion refinement in that it is not required for abstract components to match the exact behaviors of refined components, but it is sufficient that abstract components upper- or lower-bound behaviors of refinements. This is of special importance for analysis as, unlike inclusion refinement, bounding refinement allows to create deterministic and monotone abstractions of non-deterministic, non-monotone implementations, simplifying analysis drastically as only one behavior remains to be analyzed.

Besides this classification we further differ between theories considering the **temporal** and **functional** behavior of components and between theories for **synchronous** and **asynchronous** components. Synchronous components allow for an implicit notion of time by assuming a certain duration of synchronous rounds, while asynchronous components are more general, but require an explicit notion of time to consider temporal behavior.

In [4] a functional language refinement relation is introduced for synchronous components. Interfaces of components are defined in a relational manner using state machines, allowing to capture input-output dependencies of components. The theory defines refinement only for a subclass of relational interfaces. Moreover, the theory does not contain proofs for the preservation of refinement on composition. These shortcomings are amended in [15] that allows for any kind of relational interfaces (with the restriction that feedback composition is not allowed to be combinatorial) and that proves the automatic lifting of refinement from component to graph level. In contrast to these works, our theory assumes an asynchronous component model, resulting in a higher expressibility. Instead of the implicit notion of time that is inherent to synchronous theories we make use of an explicit notion in the form of timestamps.

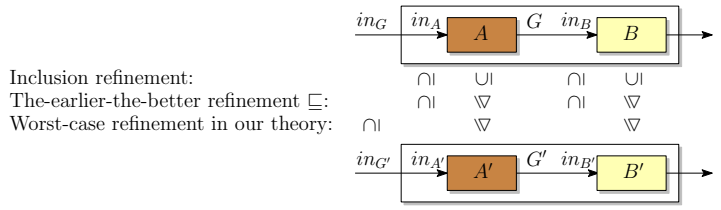


Fig. 2. A graph $G$ and its refinement graph $G'$.

A functional language refinement relation for concurrent asynchronous components is presented in [5]. The interfaces are described using automata in an operational manner. This theory of interface automata is extended in [6] for timed automata [1], allowing an explicit specification of progress of time. However, [1] lacks a definition of refinement, which is added in [3]. In comparison, our theory is asynchronous as well, but also denotational and it operates on entire streams instead of single values, which allows for a concise representation of complex interface relations.

Another fundamental difference of our theory compared to the aforementioned is that our refinement relation is bounding, i.e. refinement does not require trace containment, but merely bounding of streams. This is illustrated in Figure 2 in which a graph $G$ is refined to a graph $G'$. In inclusion refinement it is allowed that the refined components $A'$ and $B'$ accept more inputs than $A$ and $B$, but for the same inputs the components $A$ and $B$ must have at least the same behaviors as $A'$ and $B'$. This implies that if $G'$ is non-deterministic also $G$ must be non-deterministic. For TETB as well as our theory, the latter is not required, but it is only needed that the behaviors of $A$ and $B$ upper-bound the behaviors of $A'$ and $B'$. Consequently, $G$ is allowed to be both deterministic and monotone even if $G'$ is neither. This is of special importance for analysis purposes as deterministic and monotone analysis models enable the application of efficient analysis techniques [14], as well as the usage of algebraic techniques on closed form expressions, which allow for deep insights into the respective problems (e.g. enabling a quick identification of bottlenecks).

In [16] the concept of creating deterministic abstractions that upper-bound the temporal behavior of a non-deterministic implementation is introduced using deterministic timed dataflow models. But the work lacks the definition of a transitive refinement relation that can be used to create multiple refinement layers. This is amended with the temporal bounding refinement relation presented in [7], the aforementioned TETB refinement theory. However, in TETB streams consist only of timestamps, but do not have a notion of values and indices, which prevents an application for systems in which reordering takes place. Indexed streams are introduced in [9], enabling the refinement and abstraction of systems with reordering.

In all aforementioned theories, refinement and abstraction are **symmetric**, i.e. if a component refines another, the latter is an abstraction of the former. As in these theories a refinement must accept at least the inputs of an abstraction, the symmetry of abstraction and refinement implies that an analysis model may well be a valid abstraction of an implementation although it only accepts a small part of the implementation's inputs. But usually just the opposite is desirable, i.e. that an analysis model accepts at least all inputs of an implementation. To account for this requirement we define our notions of abstraction and refinement **asymmetric**, such that a refinement must accept at least the inputs of a model and an abstraction at least the inputs of an implementation. This makes our theory equally suitable for both system design and analysis.

Refinement in TETB further implies both worst-case lower-bounding and input acceptance preservation. This allows for the creation of non-deterministic refinements of deterministic worst-case models, which is fundamentally different to both language and simulation refinement and

similar to the worst-case refinement in our theory. As illustrated in Figure 2, input acceptance preservation does not only need to hold between two refining graphs as in our theory, but between all components of the graphs individually. A consequence of this limitation is that for preservation of refinement on serial composition the components must be in most cases input-complete and on feedback composition even always input-complete, which renders the whole notion of input-restricted components in many cases useless. We only require bounding on component level, which removes any requirement on input acceptance preservation. TETB further requires refinement-monotonicity of the individual components on feedback and serial compositions, as well as refinement-continuity on feedback composition, which we do not (a short discussion on these differences can be found after the respective proofs in Section 5.4). Lastly, we introduce the notion of value refinement, making our refinement theory both temporal and functional, and we define both worst-case and a best-case bounding relations, enabling the creation of both worst-case and best-case models.

## 4 TIMED COMPONENT MODEL

In this section we first give a formal definition of streams on ports, which is subsequently generalized to traces on interfaces consisting of multiple ports. Using such interfaces we define components relating traces on input interfaces to traces on output interfaces. Finally we define parallel, serial and feedback compositions of components, which enable the construction of component graphs.

### 4.1 Ports & Streams

We define streams as infinite sequences of events, with each event mapping an index to a timestamp and a value. Subsequently we define the length of streams, as well as so-called ports that are used to transfer streams from a specific value domain. Finally we specify the connectibility of ports and define the prefix and earlier-than relations for streams on the same ports.

*Definition 4.1 (Stream).* A stream $x$ on a port $p$ is an infinite sequence of indexed events, with each event consisting of the production time of the event in the form of a timestamp and a value from the value domain of the port. Formally $x$ can be described as a total mapping $x : \mathbb{N} \to \mathcal{T} \times O$, with $\mathcal{T}$ a continuous time domain and $O$ a value domain. We require that the time domain $\mathcal{T}$ is a lattice with respect to an ordering relation $\leq$ and that $\mathcal{T}$ has an infimum $0 \in \mathcal{T}$, as well as a supremum $\infty \in \mathcal{T}$, such that $\forall_{\tau \in \mathcal{T}} : 0 \leq \tau \leq \infty$. Analogously we require that the value domain $O$ is a lattice with respect to an ordering relation $\models$ and that $O$ has an infimum $\vartheta^0 \in O$, as well as a supremum $\vartheta^\infty \in O$, such that $\forall_{\vartheta \in O} \colon\ \vartheta^0 \models \vartheta \models \vartheta^\infty$. We use $\tau_x : \mathbb{N} \to \mathcal{T}$ and $\vartheta_x : \mathbb{N} \to O$ to retrieve timestamps and values of events by their indices, respectively.

Although streams are formally defined as infinite sequences of events, streams can also be seen as finite. We define the length of streams as follows:

*Definition 4.2 (Stream Length).* We define an event of a stream $x$ at index $i$ to be absent iff $\tau_x(i) = \infty$ and $\vartheta_x(i) = \vartheta^\infty$. The length of a stream can then be defined as the smallest index from which onwards all events are absent, i.e. (with $\min \emptyset \equiv \infty$):

$$|x| = \min\{i \in \mathbb{N} \mid \forall_{i' \geq i} \colon\ \tau_x(i') = \infty \wedge \vartheta_x(i') = \vartheta^\infty\}$$

In our timed component model we transfer streams over so-called ports, which are specified as follows:

*Definition 4.3 (Port).* A port $p$ is characterized by a tuple $(x, O_p, \models_p)$ and contains a stream $x \in St(p)$, with $St(p)$ the set of all valid streams on port $p$. The set $St(p)$ is constructed based on a port-specific value domain $O_p$, such that $St(p) = \{x \mid x : \mathbb{N} \to \mathcal{T} \times O_p\}$. The port-specific value

domain $O_p$ must adhere to the requirements on the value domains of streams, i.e. it must be a lattice with respect to an ordering relation $\models_p$, with $\vartheta_p^0$ the infimum and $\vartheta_p^\infty$ the supremum. Based on the ordering relations $\leq$ of $\mathcal{T}$ and $\models_p$ of $O_p$ we further define the null stream $x_p^0 \in St(p)$, such that $\forall_{i\in\mathbb{N}}\colon\ \tau_{x_p^0}(i) = 0 \wedge \vartheta_{x_p^0}(i) = \vartheta_p^0$, and the empty stream $x_p^\infty \in St(p)$, such that $\forall_{i\in\mathbb{N}}\colon\ \tau_{x_p^\infty}(i) = \infty \wedge \vartheta_{x_p^\infty}(i) = \vartheta_p^\infty$, respectively.

In the following we construct interfaces from multiple ports and connect such interfaces to other interfaces by connecting the underlying ports. However, not all ports can be connected as they may have different value domains. Consequently we define a sufficient requirement on the connectibility of ports as follows:

*Definition 4.4 (Port Connectibility).* Let $q$ and $p$ be two different ports, i.e. $q \neq p$. Then port $p$ is connectible to a port $q$, i.e. $q \rightarrow p$, iff it holds that all valid streams on port $q$ are also valid streams on port $p$, i.e. $St(q) \subseteq St(p)$.

We define prefix and earlier-than ordering relations for streams on the same port as follows:

*Definition 4.5 (Stream Order).* Let $x, x' \in St(p)$ be two streams on a port $p$. The prefix ordering relation $\preceq$ and the smaller-than ordering relation $\leq$ for streams are defined as:

$$x \preceq x' \equiv\ |x| \leq |x'| \wedge \forall_{i<|x|} : \tau_x(i) = \tau_{x'}(i) \wedge \vartheta_x(i) = \vartheta_{x'}(i)$$
$$x \leq x' \equiv\ |x| = |x'| \wedge \forall_{i<|x|} : \tau_x(i) \leq \tau_{x'}(i) \wedge \vartheta_x(i) \models_p \vartheta_{x'}(i)$$

It can be seen that both the prefix and the smaller-than ordering relations of streams have the same properties as in [7], which implies that we can reuse the results from [7] that are based on these properties.

### 4.2 Traces & Interfaces

We generalize the concept of streams on ports to traces on interfaces, with interfaces being sets of ports and traces being sets of streams on the ports of interfaces. Subsequently we define connectibility and connection of interfaces and lift the prefix and earlier-than ordering relations of streams to traces.

*Definition 4.6 (Interface).* An interface $P$ is a set of $|P|$ different ports.

*Definition 4.7 (Trace).* A trace $X$ is a set of $|X| = |P|$ streams on an interface $P$, such that each stream $x \in X$ is on a different port $p \in P$. In the following we use the shorthand notation $X[p]$ to retrieve the stream on port $p \in P$. The set of all valid traces on an interface $P$ is then defined as $Tr(P) = \{X \mid |X| = |P| \wedge \forall_{p\in P}\colon\ X[p] \in St(p)\}$, with $X_P^0 \in Tr(P)$ the null trace and $X_P^\infty \in Tr(P)$ the empty trace, such that $\forall_{p\in P}\colon\ X_P^0[p] = x_p^0 \wedge X_P^\infty[p] = x_p^\infty$.

These definitions allow us to lift connectibility of ports to connectibility of interfaces:

*Definition 4.8 (Interface Connectibility).* Let $Q$ and $P$ be two disjoint interfaces of the same numbers of ports, i.e. $Q \cap P = \emptyset$ and $|Q| = |P|$. Furthermore let $\Theta : Q \rightarrow P$ be a bijective mapping, i.e. $\forall_{q\in Q}\exists_{p\in P}\colon\ p = \Theta(q)$ and $\forall_{q,q'\in Q}\colon\ q \neq q' \Rightarrow \Theta(q) \neq \Theta(q')$. Then interface $P$ is connectible to interface $Q$ given mapping $\Theta$, i.e. $Q \rightarrow_\Theta P$, iff it holds that all mapped ports are connectible, i.e. $\forall_{q\in Q}\colon\ q \rightarrow \Theta(q)$.

Given connectibility of interfaces we can now also define the connection of interfaces as follows:

*Definition 4.9 (Interface Connection).* Let $Q$ and $P$ be two disjoint interfaces of the same numbers of ports. Furthermore let $\Theta : Q \rightarrow P$ be a bijective mapping. If $P$ is connectible to $Q$ given mapping $\Theta$, i.e. $Q \rightarrow_\Theta P$, then the interfaces $Q$ and $P$ can be connected by an interface connection $C^\Theta$.

An interface connection $C^{\Theta}$ is characterized by a tuple $(Q, P, \Theta)$ and assigns the streams on $P$ to streams on $Q$, according to mapping $\Theta$. Formally $C^{\Theta}$ can be described as a function $C^{\Theta} : Tr(Q) \rightarrow Tr(P)$ with $X = C^{\Theta}(Y) \equiv \forall_{q \in Q} :\ X[\Theta(q)] = Y[q]$.

Finally we can lift the prefix and earlier-than ordering relations of streams to prefix and earlier-than ordering relations of traces as follows:

*Definition 4.10 (Trace Order).* Let $X, X' \in Tr(P)$ be two traces on an interface $P$. The prefix ordering relation $\leq$ and earlier-than ordering relation $\leq$ for traces are defined as:

$$X \leq X' \equiv \forall_{p \in P} :\ X[p] \leq X'[p]$$
$$X \leq X' \equiv \forall_{p \in P} :\ X[p] \leq X'[p]$$

### 4.3 Components

Instead of the term actor that is used in [7] we use the term component to prevent confusion with actors from the timed dataflow theory, of which Synchronous Dataflow (SDF) actors are an example. A component is defined as follows:

*Definition 4.11 (Component).* A component $A$ is characterized by a tuple $(P_A, Q_A, R_A)$ and assigns the traces on output interface $Q_A$ to traces $Y$ that are derived according to relation $R_A \subseteq Tr(P_A) \times Tr(Q_A)$ from traces $X$ on input interface $P_A$. We use $XAY$ to denote $(X, Y) \in R_A$, with $X \in in_A$ and $Y \in out_A$. The input and output sets of $A$ are defined as:

$$in_A = \{X \in Tr(P_A) \mid \exists_{Y \in Tr(Q_A)} :\ XAY\}$$
$$out_A = \{Y \in Tr(Q_A) \mid \exists_{X \in Tr(P_A)} :\ XAY\}$$

We require that the empty input trace $X^{\infty} \in Tr(P_A)$ is a valid trace with respect to $R_A$, i.e. $X^{\infty} \in in_A$, and that for the empty input trace the relation $R_A$ always results in the empty output trace $Y^{\infty} \in Tr(Q_A)$, i.e. $X^{\infty}AY \Rightarrow Y = Y^{\infty}$. Initially (before a trace $X$ with $|X| > 0$ is assigned to $P_A$) the input interface $P_A$ contains the empty input trace $X^{\infty}$, which implies that initially the respective output interface $Q_A$ contains the empty output trace $Y^{\infty}$.

### 4.4 Component Graphs

Composing components by connecting interfaces yields new components. In the following we define the components resulting from parallel, serial and feedback compositions, as well as component graphs as components composed of other components.

*Definition 4.12 (Parallel Composition).* Let $A$ and $B$ be two components with disjoint input interfaces $P_A$ and $P_B$ and disjoint output interfaces $Q_A$ and $Q_B$. Then the parallel composition of $A$ and $B$ is a component $A||B$ with input interface $P_{A||B} = P_A \cup P_B$, output interface $Q_{A||B} = Q_A \cup Q_B$ and the relation between input and output interfaces as follows:

$$R_{A||B} = \{(X_A \cup X_B, Y_A \cup Y_B) \in Tr(P_{A||B}) \times Tr(Q_{A||B}) \mid X_A A Y_A \wedge X_B B Y_B\}$$

For serial and feedback compositions we need the following notion of component connectibility:

*Definition 4.13 (Component Connectibility).* Let $A$ and $B$ be two components with input interfaces $P_A$ and $P_B$ and output interfaces $Q_A$ and $Q_B$, respectively. Furthermore let $Q_A^* \subseteq Q_A$ and $P_B^* \subseteq P_B$ be two disjoint interfaces with the same numbers of ports and let $\Theta : Q_A^* \rightarrow P_B^*$ be a bijective mapping. Then it holds that component $B$ is connectible to component $A$ given mapping $\Theta$, i.e. $A \rightarrow_{\Theta} B$, iff it holds that $P_B^*$ is connectible to $Q_A^*$ given mapping $\Theta$, i.e. $Q_A^* \rightarrow_{\Theta} P_B^*$.

This allows us to define serial composition as follows:

*Definition 4.14 (Serial Composition).* Let $A$ and $B$ be two components with disjoint input interfaces $P_A$ and $P_B = P_B^\diamond \cup P_B^*$ and disjoint output interfaces $Q_A = Q_A^* \cup Q_A^\diamond$ and $Q_B$, respectively, with $P_B^\diamond \cap P_B^* = Q_A^* \cap Q_A^\diamond = \emptyset$. Furthermore let $Q_A^*$ and $P_B^*$ be two disjoint interfaces with the same numbers of ports and let $\Theta : Q_A^* \to P_B^*$ be a bijective mapping.

If $B$ is connectible to $A$ given mapping $\Theta$, i.e. $A \to_\Theta B$, then the serial composition of $A$ and $B$ given mapping $\Theta$ is obtained by connecting interface $P_B^*$ to interface $Q_A^*$ via an interface connection $C^\Theta$. This results in a component $A\Theta B$ with input interface $P_{A\Theta B} = P_A \cup P_B^\diamond$, output interface $Q_{A\Theta B} = Q_A^\diamond \cup Q_B$ and the relation between input and output interfaces as follows:

$$R_{A\Theta B} = \{(X_A \cup X_B^\diamond, Y_A^\diamond \cup Y_B) \in Tr(P_{A\Theta B}) \times Tr(Q_{A\Theta B}) \mid \exists_{X_A A(Y_A^* \cup Y_A^\diamond)}: \quad \exists_{(X_B^\diamond \cup C^\Theta(Y_A^*))BY_B} \wedge \\ \forall_{X_A A(Y_A^{*\bullet} \cup Y_A^{\diamond\bullet})}: \ \exists_{(X_B^\diamond \cup C^\Theta(Y_A^{*\bullet}))BY_B^\bullet}\}$$

The first line thereby ensures that an input is only accepted by the composition $A\Theta B$ if there exists a corresponding output of $A$ to that input which is also accepted by $B$. And the second line addresses potential non-determinism of $A$, such that an input is only accepted by $A\Theta B$ if all possible outputs of $A$ for that input are also accepted by $B$. This prevents the occurrence of dead states.

Lastly, we define feedback composition as follows:

*Definition 4.15 (Feedback Composition).* Let $A$ be a component with input interface $P_A = P_A^\diamond \cup P_A^*$ and output interface $Q_A = Q_A^* \cup Q_A^\diamond$, with $P_A^\diamond \cap P_A^* = Q_A^* \cap Q_A^\diamond = \emptyset$. Furthermore let $Q_A^*$ and $P_A^*$ be two disjoint interfaces with the same numbers of ports and let $\Theta : Q_A^* \to P_A^*$ be a bijective mapping.

If $A$ is connectible to $A$ given mapping $\Theta$, i.e. $A \to_\Theta A$, then the feedback composition of $A$ given mapping $\Theta$ is obtained by connecting interface $P_A^*$ to interface $Q_A^*$ via an interface connection $C^\Theta$. This results in a component $A\Theta A$ with input interface $P_{A\Theta A} = P_A^\diamond$, output interface $Q_{A\Theta A} = Q_A^\diamond$ and the relation between input and output interfaces as follows (with $Y^{*,\infty}$ the empty trace on interface $Q_A^*$, for a formal definition of the trace limit $\lim_{k\to\infty} Y_k$ please refer to [10]):

$$R_{A\Theta A} = \{(X^\diamond, Y^\diamond) \in Tr(P_{A\Theta A}) \times Tr(Q_{A\Theta A}) \mid Y_{-1}^* = Y_{-1}^{*\bullet} = Y^{*,\infty} \wedge \\ \exists_{(X^\diamond \cup C^\Theta(Y_{k-1}^*))A(Y_k^* \cup Y_k^\diamond)}: \quad \exists_{Y^* \cup Y^\diamond = \lim_{k\to\infty} Y_k^* \cup Y_k^\diamond} \wedge \\ \forall_{(X^\diamond \cup C^\Theta(Y_{k-1}^{*\bullet}))A(Y_k^{*\bullet} \cup Y_k^{\diamond\bullet})}: \ \exists_{Y^{*\bullet} \cup Y^{\diamond\bullet} = \lim_{k\to\infty} Y_k^{*\bullet} \cup Y_k^{\diamond\bullet}}\}$$

For a component $A$ without feedback the relation of the component is only applied once for each input trace $X$, resulting in one output trace $Y$. For a component $A\Theta A$ with feedback, however, the relation of the component is applied multiple times for each external input trace $X^\diamond$ until a fixed point is reached, as the internal input trace on interface $P_A^*$ of the underlying component $A$ depends on the internal output trace on interface $Q_A^*$ of the same component. According to the definition of components, such a sequence of multiple applications always begins with the empty trace on the internal input interface. The second line captures this by ensuring that an input is only accepted by $A\Theta A$ if $A$ has a fixed point for this input that is reachable, starting from the empty trace on the internal interface. Note that this makes our feedback composition fundamentally different to the one in TETB, which only requires existence of fixed points. Compared to TETB, the reformulation facilitates an expression of operational components in our denotational timed component model and relaxes the conditions under which automatic lifting of bounding and input acceptance is given. The third line again addresses potential non-determinism of $A$, preventing dead states by ensuring that an input is only accepted by $A\Theta A$ if not only one, but any sequence of internal traces, starting from the empty trace, converges to a fixed point.

Based on these compositions we can finally define graphs of components as follows:

*Definition 4.16 (Component Graph).* A component graph $G$ is itself a component composed of other components via parallel, serial and / or feedback composition.

## 5 BOUNDING

In this section we introduce the notions of best-case and worst-case abstraction and refinement, which are used to create best-case and worst-case models of implementations, as well as implementations from best-case and worst-case models. For that purpose we define the relation $\lhd$ to express lower-bounding of streams and traces. Given trace lower-bounding, we define the relations $\lessdot$ and $\lessdot\!\!\!\cdot$ to express best-case and worst-case lower-bounding of components. We show that composition of components preserves bounding, which, together with input acceptance preservation, enables abstraction and refinement of component graphs.

### 5.1 Stream & Trace Bounding

The bounding of streams is defined as follows:

*Definition 5.1 (Stream Bounding).* Let $x, x' \in St(p)$ be two streams on a port $p$. The bounding relation $\lhd$ for streams is defined as:

$$x \lhd x' \equiv \ \forall_i : \ \tau_x(i) \leq \tau_{x'}(i) \ \wedge \ \vartheta_x(i) \models_p \vartheta_{x'}(i)$$

Just like the prefix and earlier-than relations, the stream bounding relation has the same properties as the stream refinement relation $\sqsubseteq$ defined in [7], which implies reusability of results. It can be seen that the set $St(p)$ forms a lattice with respect to the bounding relation, with the null stream $x_p^0 \in St(p)$ the infimum and the empty stream $x_p^\infty \in St(p)$ the supremum.

The bounding relation for streams is lifted to a bounding relation for traces as follows:

*Definition 5.2 (Trace Bounding).* Let $X, X' \in Tr(P)$ be two traces on an interface $P$. The bounding relation $\lhd$ for traces is defined as:

$$X \lhd X' \equiv \forall_{p \in P} : \ X[p] \lhd X'[p]$$

The set of traces $Tr(P)$ consequently also forms a lattice with respect to the bounding relation, with the null trace $X_P^0 \in Tr(P)$ the infimum and the empty trace $X_P^\infty \in Tr(P)$ the supremum.

### 5.2 Component Bounding

Given trace bounding we define best-case bounding and worst-case bounding of components as follows:

*Definition 5.3 (Component Bounding).* Let $A, A'$ be two components with input interfaces $P_A = P_{A'}$ and output interfaces $Q_A = Q_{A'}$.

Component $A$ is a best-case lower bound on $A'$, i.e. $A \lessdot A'$, iff it holds $\forall_{X \in in_A, X' \in in_{A'}}$ that:

$$X \lhd X' \Rightarrow \forall_{X'A'Y'} \exists_{XAY} : \ Y \lhd Y'$$

Analogously, component $A$ is a worst-case upper bound on $A'$, i.e. $A \gtrdot A'$, iff it holds $\forall_{X \in in_A, X' \in in_{A'}}$ that:

$$X \rhd X' \Rightarrow \forall_{X'A'Y'} \exists_{XAY} : \ Y \rhd Y'$$

In words this means that $A$ is a best-case lower bound (worst-case upper bound) on $A'$ iff for every input trace $X$ of $A$ that is a lower bound (upper bound) on an input trace $X'$ of $A'$ there exists an output trace $Y$ with $XAY$ that is a lower bound (upper bound) on every output trace $Y'$ with $X'A'Y'$.

### 5.3 Bounding Lifting

In this section we discuss the automatic lifting of bounding from component to graph level, i.e. that two graphs bound each other if all their respective components bound each other. For that

purpose we prove that bounding between individual components is preserved on parallel, serial and feedback composition.

For parallel composition it holds:

Lemma 5.4 (Parallel Bounding Preservation). *With $\nabla{=}\trianglelefteq$ ($\nabla{=}\trianglerighteq$) let $A \nabla A'$ and $B \nabla B'$. From this follows that the respective parallel compositions also bound each other, i.e. $A||B \nabla A'||B'$.*

Proof. Trivial. □

For serial composition we obtain analogously:

Lemma 5.5 (Serial Bounding Preservation). *With $\nabla{=}\trianglelefteq$ ($\nabla{=}\trianglerighteq$) let $A \nabla A'$ and $B \nabla B'$. From this follows that the respective serial compositions also bound each other, i.e. $A\Theta B \nabla A'\Theta B'$.*

Proof Idea. From $A \nabla A'$ follows that for any input that is accepted by $A\Theta B$ and that bounds an input of $A'\Theta B'$, there must be an output of $A$ that bounds all respective outputs of $A'$. As these outputs must be accepted by $B$, according to the second line in the definition of serial composition, it follows with $B \nabla B'$ that there must also be an output of $B$ that bounds all respective outputs of $B'$. This lets us conclude that for any input of $A\Theta B$ that bounds an input of $A'\Theta B'$ there must be an output of $A\Theta B$ that bounds all respective outputs of $A'\Theta B'$, i.e. $A\Theta B \nabla A'\Theta B'$.

Proof. Let $(X_A \cup X_B^\diamond) \in in_{A\Theta B}$, $(X'_A \cup X_B^{\diamond\prime}) \in in_{A'\Theta B'}$ and with $\Delta{=}\triangleleft$ ($\Delta{=}\triangleright$) that $(X_A \cup X_B^\diamond)\ \Delta\ (X'_A \cup X_B^{\diamond\prime})$. Thus it holds that also $X_A\ \Delta\ X'_A$ and from $A \nabla A'$ it follows:

$$\forall_{X'_A A'(Y_A^{*\prime} \cup Y_A^{\diamond\prime})} \exists_{X_A A(Y_A^* \cup Y_A^\diamond)} \colon\ (Y_A^* \cup Y_A^\diamond)\ \Delta\ (Y_A^{*\prime} \cup Y_A^{\diamond\prime}) \tag{1}$$

$(X_A \cup X_B^\diamond) \in in_{A\Theta B}$ implies that it holds for all $Y_A^*$ with $X_A A(Y_A^* \cup Y_A^\diamond)$ that $(X_B^\diamond \cup C^\Theta(Y_A^*)) \in in_B$. Analogously $(X'_A \cup X_B^{\diamond\prime}) \in in_{A'\Theta B'}$ implies that it holds for all $Y_A^{*\prime}$ with $X'_A A'(Y_A^{*\prime} \cup Y_A^{\diamond\prime})$ that $(X_B^{\diamond\prime} \cup C^\Theta(Y_A^{*\prime})) \in in_{B'}$.

For $(Y_A^* \cup Y_A^\diamond)$ and $(Y_A^{*\prime} \cup Y_A^{\diamond\prime})$ according to Equation 1 it thus holds that $(X_B^\diamond \cup C^\Theta(Y_A^*)) \in in_B$, $(X_B^{\diamond\prime} \cup C^\Theta(Y_A^{*\prime})) \in in_{B'}$ and with $(X_A \cup X_B^\diamond)\ \Delta\ (X'_A \cup X_B^{\diamond\prime})$ that $(X_B^\diamond \cup C^\Theta(Y_A^*))\ \Delta\ (X_B^{\diamond\prime} \cup C^\Theta(Y_A^{*\prime}))$. With $B \nabla B'$ it follows:

$$\forall_{(X_B^{\diamond\prime} \cup C^\Theta(Y_A^{*\prime}))B'Y'_B} \exists_{(X_B^\diamond \cup C^\Theta(Y_A^*))BY_B} \colon\ Y_B\ \Delta\ Y'_B$$

From this it can be finally concluded that it holds $\forall_{(X_A \cup X_B^\diamond) \in in_{A\Theta B}}$ and $\forall_{(X'_A \cup X_B^{\diamond\prime}) \in in_{A'\Theta B'}}$:

$$(X_A \cup X_B^\diamond)\ \Delta\ (X'_A \cup X_B^{\diamond\prime})\ \Rightarrow\ \forall_{(X'_A \cup X_B^{\diamond\prime})A'\Theta B'(Y_A^{\diamond\prime} \cup Y'_B)} \exists_{(X_A \cup X_B^\diamond)A\Theta B(Y_A^\diamond \cup Y_B)} \colon\ (Y_A^\diamond \cup Y_B)\ \Delta\ (Y_A^{\diamond\prime} \cup Y'_B)$$

This is just the definition of $A\Theta B \nabla A'\Theta B'$, q.e.d. □

Note that Lemma 5.5 does neither imply $A\Theta B \nabla A\Theta B'$ nor $A\Theta B \nabla A'\Theta B$ in general.

For feedback composition it holds:

Lemma 5.6 (Feedback Bounding Preservation). *With $\nabla{=}\trianglelefteq$ ($\nabla{=}\trianglerighteq$) let $A \nabla A'$. From this follows that the respective feedback compositions also bound each other, i.e $A\Theta A \nabla A'\Theta A'$.*

Proof Idea. Let there be an input that is accepted by $A\Theta A$ and that bounds an input that is accepted by $A'\Theta A'$. Consequently, the respective combinations of these traces with empty traces on the internal interfaces also bound each other. From $A \nabla A'$ then follows that for these inputs there exists an output of $A$ that bounds all respective outputs of $A'$. With the third line of the definition of feedback composition it follows that these outputs must be accepted by $A$ and $A'$ as inputs, respectively, again resulting in an output of $A$ that bounds all outputs of $A'$. As the third line ensures that for any such bounding sequences fixed points are reached, it follows that also the respective fixed points bound each other. This lets us conclude that for any input of $A\Theta A$ that

bounds an input of $A'\Theta A'$ there exists an output of $A\Theta A$ that bounds all respective outputs of $A'\Theta A'$, i.e. $A\Theta A \nabla A'\Theta A'$.

Proof. We denote with $X_k = (X^\diamond \cup X_k^*)$ and $Y_k = (Y_k^* \cup Y_k^\diamond)$ the input and output traces of component $A$ in each feedback iteration $k$ after assignment of an external input trace $X^\diamond$ and with $X'_k = (X^{\diamond\prime} \cup X_k^{*\prime})$ and $Y'_k = (Y_k^{*\prime} \cup Y_k^{\diamond\prime})$ the input and output traces of $A'$ after assignment of $X^{\diamond\prime}$ analogously.

Given that the external input of $A$ is a lower bound (upper bound) on the external input of $A'$ we first show with $\Delta{=}\triangleleft$ ($\Delta{=}\triangleright$) that for each feedback iteration there exist output traces of $A$ that are lower bounds (upper bounds) on all output traces of $A'$, using mathematical induction. Based on this we prove that there exists a fixed point of $A$ for each fixed point of $A'$, such that the fixed point of $A$ is a lower bound (upper bound) on the fixed points of $A'$. We begin with the mathematical induction:

*Induction base:* Let $X^\diamond \in in_{A\Theta A}$, $X^{\diamond\prime} \in in_{A'\Theta A'}$ and $X^\diamond \;\Delta\; X^{\diamond\prime}$. It follows from the definition of components that initially (before $X^\diamond$ and $X^{\diamond\prime}$ are assigned) the traces on all interfaces are empty traces, such that the input traces of $A$ on assignment of $X^\diamond$ and $A'$ on assignment of $X^{\diamond\prime}$ are $X_0 = (X^\diamond \cup C^\Theta(Y^{*,\infty}))$ and $X'_0 = (X^{\diamond\prime} \cup C^\Theta(Y^{*\prime,\infty}))$, respectively. With the definition of feedback composition it follows from $X^\diamond \in in_{A\Theta A}$ that $X_0 \in in_A$ and from $X^{\diamond\prime} \in in_{A'\Theta A'}$ that $X'_0 \in in_{A'}$. Due to $X^\diamond \;\Delta\; X^{\diamond\prime}$ it holds that $X_0 \;\Delta\; X'_0$. From $A \nabla A'$ it then follows that there exists an $X_0 A Y_0$ for all $X'_0 A' Y'_0$ such that $Y_0 \;\Delta\; Y'_0$.

*Induction hypothesis:* Let $X_k \in in_A$, $X'_k \in in_{A'}$, let $X_k \;\Delta\; X'_k$ and let an $X_k A Y_k$ exist for all $X'_k A' Y'_k$ such that $Y_k \;\Delta\; Y'_k$.

*Induction step:* With the definition of feedback composition it follows from $X^\diamond \in in_{A\Theta A}$ that $X_{k+1} = (X^\diamond \cup C^\Theta(Y_k^*)) \in in_A$ and from $X^{\diamond\prime} \in in_{A'\Theta A'}$ that $X'_{k+1} = (X^{\diamond\prime} \cup C^\Theta(Y_k^{*\prime})) \in in_{A'}$. From the definition of trace bounding it follows that $X_{k+1} \;\Delta\; X'_{k+1}$ and $A \nabla A'$ lets us conclude that there exists an $X_{k+1} A Y_{k+1}$ for all $X'_{k+1} A' Y'_{k+1}$ such that $Y_{k+1} \;\Delta\; Y'_{k+1}$.

Thus it holds for $X^\diamond \in in_{A\Theta A}$, $X^{\diamond\prime} \in in_{A'\Theta A'}$ and $X^\diamond \;\Delta\; X^{\diamond\prime}$ that:

$$\forall_{k\in\mathbb{N}}\colon\ \forall_{X'_k A' Y'_k} \exists_{X_k A Y_k}\colon\ X_k \;\Delta\; X'_k \wedge Y_k \;\Delta\; Y'_k \tag{2}$$

According to the definition of feedback composition $X^\diamond \in in_{A\Theta A}$ implies that for any sequence $(X^\diamond \cup C^\Theta(Y_k^*))A(Y_{k+1}^* \cup Y_{k+1}^\diamond)$ with $Y_0^* = Y^{*,\infty}$ there exists a $Y_\infty = \lim_{k\to\infty} Y_k$ that defines a fixed point of the sequence, i.e. it holds that $(X^\diamond \cup C^\Theta(Y_\infty^*))A(Y_\infty^* \cup Y_\infty^\diamond)$. Analogously $X^{\diamond\prime} \in in_{A'\Theta A'}$ implies that for any sequence $(X^{\diamond\prime} \cup C^\Theta(Y_k^{*\prime}))A'(Y_{k+1}^{*\prime} \cup Y_{k+1}^{\diamond\prime})$ with $Y_0^{*\prime} = Y^{*,\infty}$ there exists a $Y'_\infty = \lim_{k\to\infty} Y'_k$ that defines a fixed point of the sequence, i.e. it holds that $(X^{\diamond\prime} \cup C^\Theta(Y_\infty^{*\prime}))A'(Y_\infty^{*\prime} \cup Y_\infty^{\diamond\prime})$.

Now consider a sequence $(X^\diamond \cup C^\Theta(Y_k^*))A(Y_{k+1}^* \cup Y_{k+1}^\diamond)$ that for any sequence $(X^{\diamond\prime} \cup C^\Theta(Y_k^{*\prime}))A'$ $(Y_{k+1}^{*\prime} \cup Y_{k+1}^{\diamond\prime})$ satisfies $X_k \;\Delta\; X'_k$ and $Y_k \;\Delta\; Y'_k$ for all $k \in \mathbb{N}$ (existence of such a sequence is guaranteed by Equation 2). Furthermore, let $Y_\infty$ and $Y'_\infty$ be the respective fixed points of such sequences. Then it holds that $Y_\infty \;\Delta\; Y'_\infty$ and thus also $Y_\infty^\diamond \;\Delta\; Y_\infty^{\diamond\prime}$. With $Y^\diamond = Y_\infty^\diamond$ and $Y^{\diamond\prime} = Y_\infty^{\diamond\prime}$ this lets us finally conclude $\forall_{X^\diamond \in in_{A\Theta A}, X^{\diamond\prime} \in in_{A'\Theta A'}}$:

$$X^\diamond \;\Delta\; X^{\diamond\prime} \Rightarrow \forall_{(X^{\diamond\prime})A'\Theta A'(Y^{\diamond\prime})} \exists_{(X^\diamond)A\Theta A(Y^\diamond)}\colon\ Y^\diamond \;\Delta\; Y^{\diamond\prime}$$

This is just the definition of $A\Theta A \nabla A'\Theta A'$, q.e.d. □

Note that in contrast to TETB refinement we do not need any further requirements on the individual components to prove serial and feedback bounding. First, input-completeness of individual components is not needed as, unlike refinement in TETB, bounding does not imply input acceptance preservation. Second, monotonicity is not needed as we make use of a different definition of component bounding, which does not only ensure output bounding for the same inputs like in

TETB, but also bounding outputs for different, but bounding inputs. And third, continuity is also not needed for feedback bounding as we make use of a different definition of fixed points than TETB, such that not only existence, but also reachability of fixed points is ensured.

Lemmas 5.4 to 5.6 finally prove the automatic lifting of bounding from component to graph level:

Theorem 5.7 (Bounding Lifting). *With $\nabla{=}\trianglelefteq$ ($\nabla{=}\trianglerighteq$) let $G$ be a graph composed of components $A$ and let $G'$ be a graph composed of components $A'$. If it holds for all components of $G$ that they are best-case lower bounds (worst-case upper bounds) on the respective components of $G'$, i.e. $A \nabla A'$, then it follows that also the graph $G$ is a best-case lower bound (worst-case upper bound) on graph $G'$, i.e. $G \nabla G'$.*

Proof. Follows immediately from Lemmas 5.4 to 5.6 and the fact that graphs consist of parallel, serial and feedback compositions. □

### 5.4 Bounding Abstraction & Refinement

After defining bounding of streams, traces and components, as well as proving the automatic lifting of bounding from component to graph level we can now also define bounding abstraction and refinement. Subsequently we discuss that bounding abstraction and refinement can be directly concluded from input acceptance preservation and bounding, as already indicated in Figure 1.

*Definition 5.8 (Bounding Abstraction).* A graph $G$ is a best-case (worst-case) abstraction of a graph $G'$ iff $G$ accepts at least all input traces that $G'$ also accepts, i.e. $in_G \supseteq in_{G'}$, and iff it holds for all accepted input traces of $G'$ that for these input traces there exists an output trace of $G$ that is a lower bound (upper bound) on all output traces of $G'$ for the same input traces, i.e. with $\Delta{=}\triangleleft$ ($\Delta{=}\triangleright$) it holds:

$$\forall_{X \in in_{G'}} \colon \forall_{XG'Y'} \exists_{XGY} \colon \; Y \; \Delta \; Y'$$

*Definition 5.9 (Bounding Refinement).* A graph $G'$ is a best-case (worst-case) refinement of a graph $G$ iff $G'$ accepts at least all input traces that $G$ also accepts, i.e. $in_G \subseteq in_{G'}$, and iff it holds for all accepted input traces of $G$ that for these input traces there exists an output trace of $G$ that is a lower bound (upper bound) on all output traces of $G'$ for the same input traces, i.e. with $\Delta{=}\triangleleft$ ($\Delta{=}\triangleright$) it holds:

$$\forall_{X \in in_{G}} \colon \forall_{XG'Y'} \exists_{XGY} \colon \; Y \; \Delta \; Y'$$

Definition 5.8 thereby applies to the creation of best-case and worst-case models for the analysis of an existing implementation (for which the specification lies in the implementation), whereas Definition 5.9 applies to the design of an implementation from a best-case and / or worst-case model (for which the specification lies in the model). Note that our definition of worst-case refinement is equal to the definition of TETB refinement, whereas our definition of best-case refinement corresponds to the The-Later-the-Better (TLTB) refinement mentioned in the future work section of [7]. As we additionally consider a different notion of bounding abstraction, it follows that our abstraction-refinement theory is a generalization of the TETB refinement theory.

With above definitions of bounding abstraction and refinement we can conclude the following three important theorems:

Theorem 5.10 (Bounding Abstraction). *A graph $G$ is a best-case (worst-case) abstraction of a graph $G'$ if $G \trianglelefteq G'$ ($G \trianglerighteq G'$) and if $in_G \supseteq in_{G'}$.*

Proof. Follows immediately from Definitions 5.3 and 5.8, as well as the fact that the $\triangleleft$ relation is reflexive, i.e. $X \triangleleft X$. □

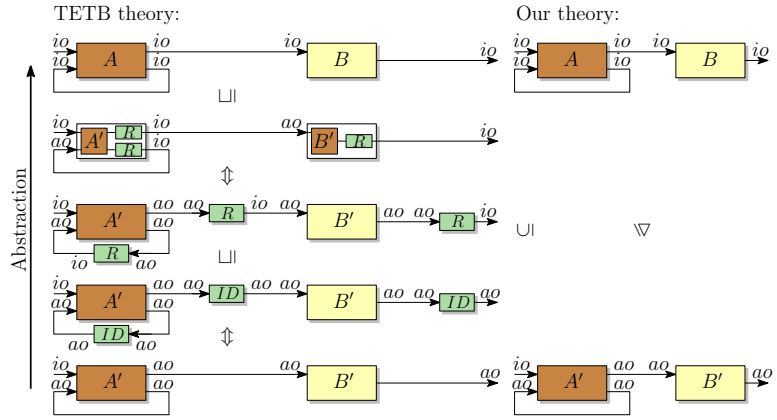


Fig. 3. In-order abstraction of an any-order implementation (*io*: in-order, *ao*: any-order).

Theorem 5.11 (Bounding Refinement). *A graph $G'$ is a best-case (worst-case) refinement of a graph $G$ if $G \lhd G'$ ($G \rhd G'$) and if $in_G \subseteq in_{G'}$.*

Proof. Follows immediately from Definitions 5.3 and 5.9, as well as the fact that the $\lhd$ relation is reflexive, i.e. $X \lhd X$. □

Theorem 5.12 (Bounding Abstraction-Refinement Transitivity). *With $\Delta{=}\lhd$ ($\Delta{=}\rhd$) let it hold for three graphs $G$, $G'$ and $G''$ that $G\ \Delta\ G'\ \Delta\ G''$ and that $in_G \supseteq in_{G'} \supseteq in_{G''}$. Then it follows that $G$ is a best-case (worst-case) bounding abstraction of $G''$. If it holds instead that $in_G \subseteq in_{G'} \subseteq in_{G''}$ then it follows that $G''$ is a best-case (worst-case) bounding refinement of $G$.*

Proof. For a formal proof refer to [10]. □

## 6 CASE STUDY

In this section we evaluate the applicability and utility of our timed component model and abstraction-refinement theory, with an emphasis on differences with the TETB refinement theory. As the obvious advantage of supporting best-case models is already discussed in Sections 1 and 3, our focus is in the following on other implications of separating bounding and input acceptance preservation.

### 6.1 Reordering

Consider the example depicted in Figure 3. It is often desirable to make an abstraction of a graph containing any-order components (i.e. components producing streams whose timestamp-order does not necessarily match index-order) like $A'$ and $B'$ to a graph consisting of in-order components (i.e. components producing streams whose timestamps are monotonically increasing in their indices) like $A$ and $B$ because in-order graphs can be analyzed more efficiently. However, with the original TETB theory [7] this is not possible, due to the following two shortcomings: First, the definition of streams in TETB does not support the expression of reordering. And second, the requirement of input-completeness for preservation of refinement on both serial and feedback compositions is too strict to allow abstractions of any-order components to in-order ones, even if reordering were expressible.

Both these shortcomings are amended in [9], which introduces indices to allow the expression of reordering and which does not require input-completeness for preservation of refinement on

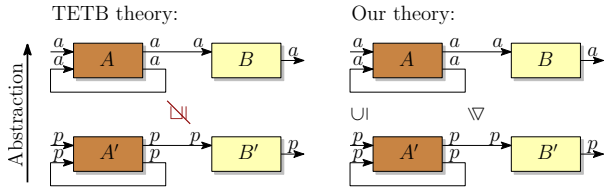


Fig. 4. Abstraction of periodic components $A'$ and $B'$ to input-complete components $A$ and $B$ ($p$: strictly periodic streams with period $P$, $a$: any streams).

serial and feedback compositions, but only that abstract components accept all outputs of the connected refined components. For our example this means that the abstract components $A$ and $B$ must accept the respective outputs of $A'$, i.e. formally for the serial composition $in_B \supseteq out^{\diamond}_{A'}$ and for the feedback composition $in^*_A \supseteq out^*_{A'}$. But as it holds that in-order is a subset of any-order, i.e. $io \subseteq ao$, these requirements are not satisfied.

Therefore one has to apply a trick to narrow the output sets of the components, like the one depicted on the left-hand side of Figure 3: At first, input-complete so-called identity components $ID$ are inserted on the outgoing edges, which simply forward input streams to output streams, resulting in a graph equivalent to the lowest level. Then these identity components are abstracted to $\sqsubseteq$-monotone, input-complete reorder components $R$ which accept any-order streams and delay them such that they become in-order. Thereafter, these reorder components are serially composed with the components $A'$ and $B'$, resulting in the graph on the second-to-highest level. This graph fulfills the requirements $in_B \supseteq out^{\diamond}_{A'}$ and $in^*_A \supseteq out^*_{A'}$, which enables the final abstraction to the graph with components $A$ and $B$.

In our theory none of this is needed. Given worst-case bounding of the individual components, i.e. $A' \trianglelefteq A$ and $B' \trianglelefteq B$, one can directly conclude that also the entire graphs bound each other, without any further requirements on the interfaces or monotonicity of components and without any additional tricks involving identity and reorder components. And together with the input acceptance preservation on graph level one can finally conclude abstraction.

### 6.2 Input Set Widening on Abstraction

While the in-order / any-order case from the previous section already illustrates the potential of our theory, one could still argue that the trick with the identity and reorder components is merely an inconvenience. As we illustrate with the following example, however, there are cases in which TETB simply fails, while our theory remains applicable.

Consider the components $A'$ and $B'$ at the bottom of Figure 4. The components only accept input streams that are strictly periodic with a period $P$. This is a realistic use-case for components representing tasks being executed on actual processors, for which the determined response times are only valid assuming a certain period. The goal is to abstract these component to dataflow components $A$ and $B$, which are by definition input-complete.

Both TETB approaches [7, 9] fail for this example as the components violate the fundamental requirement of input acceptance preservation on component level, i.e. it would have to hold that $in_{A'} \supseteq in_A$ and $in_{B'} \supseteq in_B$, but due to the fact that periodic streams with a period $P$ are a subset of any streams, i.e. $p \subseteq a$, this is clearly not the case. Now one could try to apply a similar trick as discussed in the previous section, using a "periodize" component instead of a reorder component which is input-complete and delays all streams to ones with a period of $P$. In this case, such components could be used not to narrow output sets, but to widen the respective input sets.

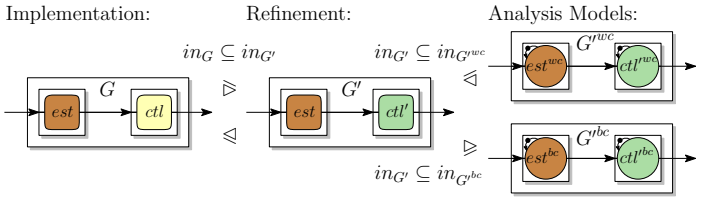


Fig. 5. Replacement of a slow input-complete controller implementation $ctl$ by a faster controller $ctl'$ with restricted input acceptance.

However, for input streams with a period larger than $P$, the only valid delayed stream would be the empty stream. Consequently, the usage of such components would prevent any useful analysis, making TETB effectively inapplicable.

In our theory, in contrast, one only has to prove that $A$ and $B$ worst-case upper-bound $A'$ and $B'$ without considering input sets, as depicted on the right-hand side of Figure 4. To prove abstraction, one has to additionally show that the input sets of the respective graphs are widening towards abstraction, which is trivial here as both $A$ and $B$ are input-complete.

### 6.3 Practical Example

In the following we apply our theory on a more practical example. We consider a simple graph of two components of which one is to be replaced by a faster one. According to TETB [7, 9] this replacement would be invalid as the faster component has a more restricted input acceptance than the original one. According to our theory this replacement would be only valid if the graph after replacement were a worst-case refinement of the graph before. To prove refinement we have to show bounding between the components individually and input acceptance preservation between the whole graphs. Proving the latter is not trivial for this example. Nevertheless, we show that input acceptance preservation can be proven by means of both best-case and worst-case models.

Consider the graph $G$ depicted on the left side of Figure 5. This graph depicts a part of a control loop which consists of two components, an estimator task that is laid out to operate on strictly periodic in-order input streams with a frequency equal to 100kHz and a controller task that is laid out in such way that it is input-complete with respect to in-order streams. It is determined that the estimator task takes between 1µs and 3µs to execute, whereas the controller task executes between 2µs and 4µs. Both tasks are executed on separate processors. Ignoring the functional behavior of the tasks, their temporal behavior can be expressed using the following relations (with $\tau_y(-1) = 0$):

$$\begin{aligned} R_{est} =& \{(x,y) \in St(p_{est}) \times St(q_{est}) \mid c \geq 0 \ \wedge \ \forall_{i<|x|}\colon \ \tau_x(i) = i \cdot 10\mu s + c \ \wedge \\ & \qquad \forall_i\colon \ 1\mu s \leq \rho_i \leq 3\mu s \ \wedge \ \tau_y(i) = \max(\tau_x(i), \tau_y(i-1)) + \rho_i\} \\ R_{ctl} =& \{(x,y) \in St(p_{ctl}) \times St(q_{ctl}) \mid \forall_i\colon \ 2\mu s \leq \rho_i \leq 4\mu s \ \wedge \ \tau_y(i) = \max(\tau_x(i), \tau_y(i-1)) + \rho_i\} \end{aligned}$$

To reduce the end-to-end latency of the two tasks and thereby improve control behavior we attempt to replace the controller task component $ctl$ with a faster component $ctl'$ that executes only between 2µs and 3µs, but that requires its input stream to be periodic with a frequency of 100kHz and a maximum jitter of 3µs. The temporal behavior of $ctl'$ is captured by the following relation:

$$\begin{aligned} R_{ctl'} = \{(x,y) \in St(p_{ctl}) \times St(q_{ctl}) \mid c' \geq 0 \ \wedge \ \forall_{i<|x|}\colon \ i \cdot 10\mu s + c' \leq \tau_x(i) \leq i \cdot 10\mu s + c' + 3\mu s \ \wedge \\ \forall_i\colon \ 2\mu s \leq \rho_i \leq 3\mu s \ \wedge \ \tau_y(i) = \max(\tau_x(i), \tau_y(i-1)) + \rho_i\} \end{aligned}$$

We have to show that the graph $G'$ in the middle of Figure 5 is a valid worst-case refinement of $G$, as only then it is guaranteed that $G'$ accepts all input streams with a frequency of 100kHz and that

for these inputs $G'$ is overall faster than $G$. According to Theorem 5.11 we must thus show that $G' \trianglelefteq G$ and $in_{G'} \supseteq in_G$.

For $G' \trianglelefteq G$ it must hold that $est \trianglelefteq est$, which is trivial for this example, and that $ctl' \trianglelefteq ctl$, which holds due to the following reasoning: Consider two input streams $x' \in in_{ctl'}$ and $x \in in_{ctl}$ with $x' \lhd x$. Using the relations of the respective components it follows for any corresponding output streams $y'$ of $ctl'$ that $\forall_i\colon\ \tau_{y'}(i) \leq \max(\tau_{x'}(i), \tau_{y'}(i-1)) + 3\mu s$, whereas $ctl$ has an output stream $y$ with $\forall_i\colon\ \tau_y(i) = \max(\tau_x(i), \tau_y(i-1)) + 4\mu s$. From $x' \lhd x$ it follows that $\tau_{y'}(0) \leq \tau_y(0)$, from this that also $\tau_{y'}(1) \leq \tau_y(1)$, and so on. Thus we can conclude that $y' \lhd y$, which satisfies Definition 5.3, and it follows $ctl' \trianglelefteq ctl$ and thus $G' \trianglelefteq G$.

It remains to be proven that $in_{G'} \supseteq in_G$. If it held that $in_{ctl'} \supseteq in_{ctl}$ this proof would be trivial. But $ctl$ is input-complete with respect to in-order streams and $ctl'$ is not (which prevents a usage of TETB for this example). Consequently we need to determine both $in_G$ and $in_{G'}$. For graph $G$ it holds that $in_G = in_{est}$, as $ctl$ is input-complete for in-order streams and as no reordering takes place in $est$. For graph $G'$ it can be seen that also $in_{G'} = in_{est}$ if $out_{est} \subseteq in_{ctl'}$. From the relation $R_{ctl'}$ we determine:

$$in_{ctl'} = \{x \in St(p_{ctl}) \mid c' \geq 0\ \wedge\ \forall_{i<|x|}\colon\ i \cdot 10\mu s + c' \leq \tau_x(i) \leq i \cdot 10\mu s + c' + 3\mu s\}$$

Deducing $out_{est}$ is not straightforward. However, a superset of $out_{est}$ can be determined rather easily by constructing two analysis models that bound the temporal behavior of $G'$. For these models we make use of deterministic Homogeneous Synchronous Dataflow (HSDF) actors, as depicted on the right side of Figure 5. From the semantics of HSDF actors [10, 14] it follows for the relation $R_{est^{wc}}$:

$$R_{est^{wc}} = \{(x, y) \in St(p_{est}) \times St(q_{est}) \mid \forall_i\colon\ \tau_y(i) = \max(\tau_x(i), \tau_y(i-1)) + 3\mu s\}$$

The other relations $R_{est^{bc}}$, $R_{ctl'^{wc}}$ and $R_{ctl'^{bc}}$ are all of similar forms, only the so-called firing durations are not 3µs, but 1µs, 3µs and 2µs, respectively. By construction it holds that $est^{bc} \trianglelefteq est \trianglelefteq est^{wc}$, $ctl'^{bc} \trianglelefteq ctl' \trianglelefteq ctl'^{wc}$, and thus also $G'^{bc} \trianglelefteq G' \trianglelefteq G'^{wc}$. With the input-completeness of the analysis models it further follows that $G'^{bc}$ is a valid best-case and $G'^{wc}$ is a valid worst-case abstraction of $G$. These dataflow abstractions allow to efficiently compute schedules that bound the temporal behavior of $est$ [8, 11]. For any output stream $y$ of $est$ we can conclude from the best-case model that $\forall_{i<|y|}\colon\ i \cdot 10\mu s + c + 1\mu s \leq \tau_y(i)$ and from the worst-case model that $\forall_{i<|y|}\colon\ \tau_y(i) \leq i \cdot 10\mu s + c + 3\mu s$. From that we obtain:

$$out_{est} \subseteq \{y \in St(q_{est}) \mid c \geq 0\ \wedge\ \forall_{i<|y|}\colon\ i \cdot 10\mu s + c + 1\mu s \leq \tau_y(i) \leq i \cdot 10\mu s + c + 3\mu s\}$$

By setting $c'$ in $in_{ctl'}$ to $c + 1\mu s$ it can be seen that all streams in $out_{est}$ are also included in $in_{ctl'}$, i.e. $out_{est} \subseteq in_{ctl'}$, and thus also $in_{G'} = in_{est} \supseteq in_G$. This concludes the proof that $G'$ is a valid worst-case refinement of $G$.

## 7 CONCLUSION

In this paper we presented a generic timed component model and an abstraction-refinement theory for asynchronous discrete-event systems. The theory supports temporal and functional bounding abstraction and refinement, which enables the usage of efficiently analyzable models for the analysis and design of non-deterministic, non-monotone real-time systems. We discussed that properties like best-case and worst-case bounding are automatically lifted from component to graph level, enabling a component-based reasoning.

Unlike the TETB theory, our theory separates the preservation of component input acceptance from bounding. We discussed that this separation evenhandedly enables abstraction, which preserves input acceptance from implementations to models, and refinement, which preserves input

acceptance from models to implementations. Consequently, our theory is an abstraction-refinement theory that equally supports model-based design and model-based analysis, whereas related works are mainly refinement theories, with a limited applicability for analysis purposes. Furthermore, we proved that the separation of input acceptance preservation and bounding enables the automatic lifting of bounding from component to graph level without a restriction to input-complete components, resulting in a significantly larger applicability compared to TETB. Finally, we introduced both worst-case and best-case bounding relations, such that our theory does not only allow for the usage of worst-case models, but also of best-case models, which are both needed to analyze systems in which jitter plays a major role.

In a case study, we discussed the extended applicability of our theory with respect to TETB on several examples. These included a simplified handling of reordering, an abstraction of input-restricted implementations to input-complete analysis models, as well as a replacement of input-complete components by faster, but input-restricted components.

## REFERENCES


[1] R. Alur and D. Dill. 1994. A theory of timed automata. *Journal of Theoretical Computer Science* 126, 2 (1994), 183–235.

[2] A. Dasdan. 2004. Experimental analysis of the fastest optimum cycle ratio and mean algorithms. *ACM Transactions on Design Automation of Electronic Systems (TODAES)* 9, 4 (2004), 385–418.

[3] A. David and others. 2010. Timed I/O automata: A complete specification theory for real-time systems. In *ACM International Conference on Hybrid Systems: Computation and Control (HSCC)*. 91–100.

[4] L. de Alfaro and T. Henzinger. 2001. Interface automata. In *European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 109–120.

[5] L. de Alfaro and T. Henzinger. 2001. Interface theories for component-based design. In *ACM International Conference on Embedded Software (EMSOFT)*. 148–165.

[6] L. de Alfaro, T. Henzinger, and M. Stoelinga. 2002. Timed interfaces. In *ACM International Workshop on Embedded Software (EMSOFT)*. 108–122.

[7] M. Geilen, S. Tripakis, and M. Wiggers. 2011. The earlier the better: A theory of timed actor interfaces. In *ACM International Conference on Hybrid Systems: Computation and Control (HSCC)*. 23–32.

[8] J. Hausmans and others. 2013. Dataflow analysis for multiprocessor systems with non-starvation-free schedulers. In *International Workshop on Software and Compilers for Embedded Systems (SCOPES)*. 13–22.

[9] J. Hausmans and M. Bekooij. 2016. A refinement theory for timed dataflow analysis with support for reordering. In *ACM International Conference on Embedded Software (EMSOFT)*.

[10] P. Kurtin and M. Bekooij. 2017. *An abstraction-refinement theory for the analysis and design of real-time systems (Extended version).* Technical Report. Centre for Telematics and Information Technology (CTIT), University of Twente, Enschede, The Netherlands.

[11] P. Kurtin, J. Hausmans, and M. Bekooij. 2016. Combining offsets with precedence constraints to improve temporal analysis of cyclic real-time streaming applications. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 1–12.

[12] E. Lee and S. Seshia. 2015. *Introduction to embedded systems: A cyber-physical systems approach* (2nd ed.).

[13] R. Milner. 1971. An algebraic definition of simulation between programs. In *International Joint Conference on Artificial Intelligence (IJCAI)*. 481–489.

[14] S. Sriram and S. Bhattacharyya. 2009. *Embedded Multiprocessors: Scheduling and Synchronization* (2nd ed.).

[15] S. Tripakis and others. 2009. On relational interfaces. In *ACM International Conference on Embedded Software (EMSOFT)*. 67–76.

[16] M. Wiggers, M. Bekooij, and G. Smit. 2009. Monotonicity and run-time scheduling. In *ACM International Conference on Embedded Software (EMSOFT)*. 177–186.

[17] P. Wilmanns and others. 2014. Accuracy improvement of dataflow analysis for cyclic stream processing applications scheduled by static priority preemptive schedulers. In *Euromicro Conference on Digital System Design Architectures, Methods and Tools (DSD)*. 9–18.

[18] P. Wilmanns and others. 2015. Buffer sizing to reduce interference and increase throughput of real-time stream processing applications. In *IEEE International Symposium on Real-Time Computing (ISORC)*. 9–18.