An Abstraction-Refinement Theory for the Analysis and Design of Real-Time Systems (Extended Version)

This paper is an extended version of the journal paper [11]

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.

Compared to the journal version of this paper, we further present several additions in this extended version, such as an inclusion abstraction-refinement theory for the same component model, the definition of the expression of several timed dataflow models in our component model, as well as various formalizations of previously informal definitions and proofs.

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

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 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 [12, 16]. 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 [3] 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 [8], 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 [10], as well as both temporal and functional bounding. But the key difference compared to [8, 10] 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 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,

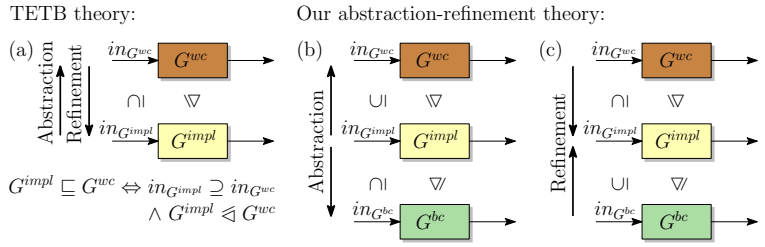

Fig. 1. Input acceptance preservation and bounding.

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 [20, 21], 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 [10], 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 highlights the differences between the journal version [11] and this extended version of the paper. Section 3 gives an informal description of our timed component model and abstraction-refinement theory and Section 4 presents related work. Section 5 formalizes the denotational timed component model and Section 6 the corresponding bounding abstraction-refinement theory. Section 7 describes an analogous inclusion abstraction-refinement theory for the same component model and Section 8 discusses input acceptance preservation and component replacement. Section 9 defines expression of timed dataflow models in our component model and Section 10 discusses the extended applicability of our theory compared to the TETB refinement theory in a case study. Section 11 finally draws the conclusions.

2 DIFFERENCES WITH JOURNAL PAPER

This paper is an extension of the journal paper [11]. Besides the original content, the following extensions are presented in the following:

- On top of the aforementioned bounding abstraction-refinement theory, also a corresponding inclusion abstraction-refinement theory is introduced for the same component model.
- The informally described concepts of input acceptance preservation and replaceability are formalized.
- The expression of several timed dataflow models in the component model is defined.
- Additional case studies are presented for a better accessibility.
- Several other informal definitions and proofs are formalized.

3 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.

3.1 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**.

3.2 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.

3.3 Inclusion Abstraction & Refinement

As discussed in the introduction, the main advantage of bounding abstraction and refinement over inclusion is that bounding can be used to remove behaviors, such that e.g. deterministic models can be created for non-deterministic implementations. The single downside of bounding is, however, that if an implementation component is not monotone with respect to the bounding relation, any valid component bounding comes with a loss of accuracy. The benefits of deterministic models for non-deterministic implementations usually outweigh a minor loss in accuracy. Nevertheless, for

cases in which accuracy is the main concern we additionally present an inclusion abstraction and refinement theory for the same component model.

Analogous to bounding abstraction and refinement, our inclusion abstraction-refinement theory is based on the concepts of **inclusion** and **input acceptance preservation**. We begin with trace equality and based on that develop inclusion of components, as well as inclusion of component graphs. Lastly, we discuss the relation of component inclusion to inclusion abstraction and refinement.

We define that a trace equals another trace if both are defined on the same interfaces and if it holds for all ports of these interfaces that the respective streams are equal.

Based on trace equality we define component inclusion as follows. Let it hold for a trace that it is a valid input trace of two different components. Furthermore, let it hold that for this input trace the first component can match all output traces of the second, i.e. the first component can produce at least all output traces that the second component can produce. If this holds for any input traces of the two components that are equal we say that the first component **includes** the second (and the second is included by the first).

We prove that inclusion is preserved on component composition, without any further requirements. This allows us to conclude that two graphs include each other if all their components include each other, i.e. inclusion is automatically lifted from component to graph level.

Lastly we define that a component graph is an **inclusion abstraction** of another graph if the first accepts at least all inputs of the second and if for the same inputs the first can match all outputs of the second. Likewise, a graph is an **inclusion refinement** of another graph if the first accepts at least all inputs of the second and if for the same inputs the second can match all outputs of the first. We discuss that inclusion abstraction and refinement can be concluded from component level inclusion and graph level input acceptance preservation. Finally, we prove that inclusion abstraction and refinement are transitive.

3.4 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

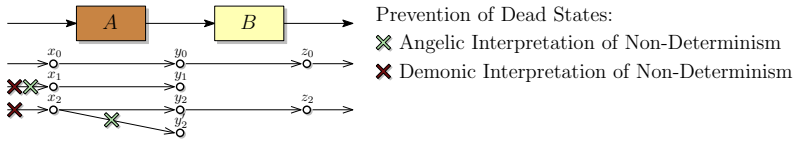

Fig. 2. Dead states and non-determinism.

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.

3.5 Exclusion of Dead States

Consider the two serially connected components depicted in Figure 2. Let us further assume that component A is a very simple component, such that it only accepts three input streams, x_0, x_1 and x_2, and produces for these input streams the following output streams: y_0 for x_0, y_1 for x_1 and y_2 or y_2' for x_2. Note that A is non-deterministic as it can produce both y_2 and y_2' for x_2. Further let B be of similar simplicity, such that it only accepts the streams y_0 and y_2 as inputs and produces for these inputs the output streams z_0 and z_2.

If the two components are serially connected, as indicated in Figure 2, **dead states** can occur, that is, component A can produce outputs such as y_1 that B does not accept. To prevent such dead states, different measures can be taken.

The most conservative approach would be to disallow the composition of such components altogether. But this would severely restrict expressiveness of our component model and prevent application for many relevant use-cases. Consequently, it appears more reasonable to allow the composition of the two components and to prevent the occurrence of dead states by different means.

For instance, we can allow the composition of any two components in principle, but explicitly define that on composition inputs that can result in dead states are disallowed. For our example, this would mean that if components A and B are serially composed, the serial composition does accept x_0 as an input (as the corresponding output y_0 is accepted by B), but not x_1 (as y_1 is not accepted by B). With respect to the non-deterministic case that occurs on an input x_2 it can be further differed between two interpretations.

According to the so-called **angelic interpretation of non-determinism** it is assumed that components always behave well with respect to their surroundings. For our example, this would mean that component A always produces y_2 on an input of x_2, as only y_2 can be accepted by B. This approach would result in maximal compatibility between components. However, a major disadvantage would be that components must internally adapt to the acceptance of other components. For instance, A would have to be aware of its surroundings, i.e. the input acceptance of B, and internally restrict its outputs to satisfy the requirements of B. While such an approach may appear suitable if considering the component model in isolation, we should keep in mind that the purpose of the component model is to represent actual, i.e. real, components, for which an internal adjustment of behaviors is not straightforward or sometimes even infeasible.

This is the reason why we adopt a **demonic interpretation of non-determinism** in our model, which assumes that components can behave, or even more drastic, always behave bad with respect to their surroundings. Applied on our example, this means that if A receives an input of x_2 it always produces an output y_2' that is not accepted by B. To prevent the occurrence of dead states we

consequently have to disallow the input of x_2 to A altogether. While this approach comes at the cost that compatibility is reduced compared to the angelic interpretation, it also allows to compose any components with each other without the need to modify component behavior: The input of x_2 to A is simply prevented, without adapting the internal behavior of A.

In consequence, we define serial and feedback compositions such that an external input is only accepted by the respective compositions if not only one, but all outputs for that input are also accepted by the connected components that take these outputs as inputs.

One important implication of such definitions of compositions is obviously that input acceptance can be reduced on compositions. This is the reason why it is generally not sufficient to evaluate input acceptance of single components in order to determine input acceptance of an entire graph of components. As comparison of graph level input acceptance is a prerequisite for both abstraction and refinement, we dedicate the entire Section 8 to this discussion.

Finally, note that even inclusion is not safe from a reduction of input acceptance: Consider that component A includes a component A', i.e. A has more behaviors than A', such that A' can only produce y_2 for an input of x_2 whereas A can produce both y_2 and y_2'. If A' is composed with B then the composition accepts both x_0 and x_2 as inputs because all respective outputs are accepted by B. But as discussed above, for the case that A is composed with B it holds that only x_0 is accepted. This demonstrates that replacing a component by another that has more behaviors than the original one can result in overall less behaviors of the composition. Nevertheless, this is not problematic in our theory as also for inclusion abstraction we clearly distinguish between inclusion, which only needs to hold for accepted inputs, and input acceptance preservation, i.e. an input acceptance reduction is tolerable with respect to inclusion.

4 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** [13] 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** [13], that is also called trace containment [15], and the stronger **simulation refinement** [13]. 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.

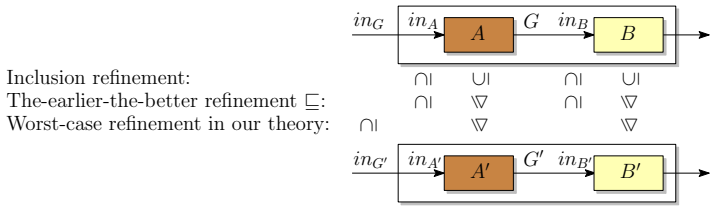

Fig. 3. A graph G and its refinement graph G'.

In [6] 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 [18] 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.

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 [7] for timed automata [1], allowing an explicit specification of progress of time. However, [1] lacks a definition of refinement, which is added in [4]. 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 3 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 [16], 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).

According to our classification, the abstract interpretation theory [2] is a functional bounding refinement theory. The key differences between abstract interpretation and our theory are as follows: First, we reason in relations between values in the same value domain, whereas abstract interpretation reasons in mappings between different domains. However, if one considers that multiple domains can be united in one value domain and relations can be established as mappings within such domains, it becomes apparent that expressibility is not reduced by our approach. Second, our theory supports a component-based reasoning (e.g. by an automatic lifting of value bounding from component to graph level), which abstract interpretation does not. Consequently, abstract interpretation also does not support a one-by-one replacement of components that is especially useful for refinement-based design processes. And third, our approach allows to combine

temporal and functional (value) bounding, whereas abstract interpretation is limited to functional bounding.

A bounding refinement relation is introduced in [17] for modular real-time systems. The theory is asynchronous and temporal as it makes use of arrival and service curves to characterize traffic, as well as provided and remaining service. This allows to express both data and resource dependencies in a single model. However, the events that form the curves are specified in the time interval instead of the time domain, which prevents a correlation of events from different curves. In consequence, this disallows cycles, i.e. the application of the theory for components with feedback. Moreover, the service curves only denote time, but no values, which renders an application for use-cases in which reordering can occur impossible. In contrast, both feedback composition and reordering can be expressed and handled with our theory.

In [19] 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 [8], 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 [10], 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 3, 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 6.5). 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.

5 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. For such components, we introduce appropriate definitions of monotonicity and continuity. Finally we define parallel, serial and feedback compositions of components, which enable the construction of component graphs.

5.1 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 5.1 (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 $x : \mathbb{N} \to \mathcal{T} \times \mathcal{O}$, with $\mathcal{T}$ a continuous time domain and $\mathcal{O}$ 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 $\mathcal{O}$ is a lattice with respect to an ordering relation $\models$ and that $\mathcal{O}$ has an infimum $\vartheta^0 \in \mathcal{O}$, as well as a supremum $\vartheta^\infty \in \mathcal{O}$, such that $\forall_{\vartheta \in \mathcal{O}} \colon\ \vartheta^0 \models \vartheta \models \vartheta^\infty$. We use $\tau_x : \mathbb{N} \to \mathcal{T}$ and $\vartheta_x : \mathbb{N} \to \mathcal{O}$ 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 5.2 (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 5.3 (Port). A port p is characterized by a tuple $(x, \mathcal{O}_p, \models_p)$ 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 $\mathcal{O}_p$, such that $St(p) = \{x \mid x : \mathbb{N} \to \mathcal{T} \times \mathcal{O}_p\}$. The port-specific value domain $\mathcal{O}_p$ 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 ϑ_p^0 the infimum and ϑ_p^∞ the supremum. Based on the ordering relations $\leq$ of $\mathcal{T}$ and $\models_p$ of $\mathcal{O}_p$ 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 5.4 (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. $q \to p$, 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 5.5 (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 [8], which implies that we can reuse the results from [8] that are based on these properties.

In the following we need to compute limits of sequences of streams. For that matter we require a distance function for streams that converges towards 0 if the distances between all timestamps and values approach 0.

First, let us define a suitable distance functions for timestamps. The most straightforward distance function between two timestamps τ and τ' would be to simply compute the absolute value of the difference, i.e. $d_{\mathcal{T}}(\tau, \tau') = |\tau - \tau'|$. However, this function has the disadvantage that if one of the timestamps is infinity (which is allowed as explicitly $\infty \in \mathcal{T}$) and the other timestamp approaches ∞, the distance will never converge to 0, as long as the second timestamp only approaches, but does not become infinity. For that reason we make use of the following distance function that does not have this disadvantage:

Definition 5.6 (Timestamp Distance). The distance between two timestamps $\tau, \tau' \in \mathcal{T}$ is a function $d_{\mathcal{T}} : \mathcal{T} \times \mathcal{T} \to \mathbb{R}_0^+$ with:

$$d_{\mathcal{T}}(\tau, \tau') = \left| 2^{-\tau} - 2^{-\tau'} \right|$$

As timestamps can take values between 0 and ∞ this function gives distances between 0 and 1, which is maximal if one of the timestamps is infinity and the other 0.

For values we cannot define a distance function directly, as the value domain is port-specific and can be defined as needed. Nevertheless, we can define the requirements that a value distance function must adhere to as follows:

Definition 5.7 (Value Distance). Let O be a lattice with respect to an ordering relation $\models$, such that O has an infimum $\vartheta^0 \in O$, as well as a supremum $\vartheta^\infty \in O$ with $\forall_{\vartheta \in O} : \ \vartheta^0 \models \vartheta \models \vartheta^\infty$. Furthermore let $n : O \to \mathbb{R}_0^+$ be a partial mapping such that $\forall_{\vartheta, \vartheta' \in O} : \ \vartheta \models \vartheta' \Rightarrow n(\vartheta) \leq n(\vartheta') \wedge \vartheta = \vartheta' \Rightarrow n(\vartheta) = n(\vartheta')$, $n(\vartheta^0) = 0$ and $n(\vartheta^\infty) = \infty$. Then the distance between two values ϑ, ϑ' can be defined as a function $d_O : O \times O \to \mathbb{R}_0^+$ with:

$$d_O(\vartheta, \vartheta') = \left| 2^{-n(\vartheta)} - 2^{-n(\vartheta')} \right|$$

A distance function for values as specified above has the same properties as the distance function for timestamps. This allows us to combine the timestamp and value distances to a distance between indices of streams as follows:

Definition 5.8 (Index Distance). Let $x, x' \in St(p)$ be two streams on a port p. The distance between two indices of such streams is a function $d_{St(p)} : St(p) \times St(p) \times \mathbb{N} \to \mathbb{R}_0^+$ with:

$$d_{St(p)}(x, x', i) = \max(d_{\mathcal{T}}(\tau_x(i), \tau_{x'}(i)), d_O(\vartheta_x(i), \vartheta_{x'}(i)))$$

With the index distance we can define a suitable distance function $D_{St(p)}$ for streams on a port p adhering to the property that if two streams converge to each other, such that all timestamps and values converge to each other, the distance function approaches 0. Formally, this means that for two streams $x, x' \in St(P)$ on a port p the following property must hold:

$$\forall_i : \ d_{St(p)}(x, x', i) \to 0 \ \Rightarrow \ D_{St(p)}(x, x') \to 0$$

The Cantor metric [14] does not adhere to this property as it requires equality of timestamps and values to converge. However, it can be seen that the following distance function satisfies above property:

Definition 5.9 (Stream Distance). The distance between two streams $x, x' \in St(p)$ on a port p is a function $D_{St(p)} : St(p) \times St(p) \to \mathbb{R}_0^+$ with:

$$D_{St(p)}(x, x') = \max_{i \in \mathbb{N}} \left(\frac{1}{i+1} \cdot (1 - 2^{-d_{St(p)}(x, x', i)})\right)$$

The first factor $(1 + i)^{-1}$ thereby ensures that $d_{St(p)}(x, x')$ converges to 0 if a stream of a finite length converges to a stream of an infinite length, whereas the second factor $1 - 2^{-d_{St(p)}(x,x',i)}$ ensures convergence per timestamp and value. The maximum finallyensures that the distance only approaches 0 if convergence of timestamps and values is given for all indices.

Using this stream distance function we can finally define the limit of stream sequences as follows:

Definition 5.10 (Stream Sequence Limit). Let $x_k \in St(p)$ be a sequence of streams and $x \in St(p)$ a stream on a port p. The sequence x_k converges to x for $k \to \infty$, i.e. $\lim_{k\to\infty} x_k = x$, iff it holds that for all $\varepsilon > 0$ there exists a $K \in \mathbb{N}$ such that for all $k \geq K$ it holds that $D_{St(p)}(x_k, x) < \varepsilon$.

5.2 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 5.11 (Interface). An interface P is a set of $|P|$ different ports.

Definition 5.12 (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 5.13 (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 $\Theta : Q \to P$ 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 Θ, i.e. $Q \to_\Theta P$, iff it holds that all mapped ports are connectible, i.e. $\forall_{q \in Q} \colon q \to \Theta(q)$.

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

Definition 5.14 (Interface Connection). Let Q and P be two disjoint interfaces of the same numbers of ports. Furthermore let $\Theta : Q \to P$ be a bijective mapping. If P is connectible to Q given mapping Θ, i.e. $Q \to_\Theta P$, then the interfaces Q and P can be connected by an interface connection C^Θ.

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

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

Definition 5.15 (Trace Order). Let $X, X' \in Tr(P)$ be two traces on an interface P. The prefix ordering relation $\preceq$ and earlier-than ordering relation $\leq$ for traces are defined as:

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

Lastly, we can also lift the limit of stream sequences to a limit of trace sequences as follows:

Definition 5.16 (Trace Sequence Limit). Let $X_k \in Tr(P)$ be a sequence of traces and $X \in Tr(P)$ a trace on an interface P. The sequence X_k converges to X for $k \to \infty$, i.e. $\lim_{k \to \infty} X_k = X$, iff all streams of the sequence X_k converge to the respective streams of X for $k \to \infty$, i.e. iff it holds that $\forall_{p \in P} : \; \lim_{k \to \infty} X_k[p] = X[p]$

5.3 Components

Instead of the term actor that is used in [8] 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 5.17 (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^∞, which implies that initially the respective output interface Q_A contains the empty output trace Y^∞.

5.4 Component Monotonicity & Continuity

We define component monotonicity with respect to any trace ordering relation as follows:

Definition 5.18 (Monotonicity). Let A be a component and $\propto$ a relation on traces. Component A is $\propto$-monotone iff it holds $\forall_{X, X' \in in_A}$:

$$(X \propto X' \;\Rightarrow\; \forall_{XAY, X'AY'} : \; Y \propto Y')$$

Moreover we call component A best-case $\propto$-monotone iff it holds $\forall_{X, X' \in in_A}$:

$$(X \propto X' \;\Rightarrow\; \forall_{X'AY'} \exists_{XAY} : \; Y \propto Y')$$

Analogously we call component A worst-case $\propto$-monotone iff it holds $\forall_{X, X' \in in_A}$:

$$(X \propto X' \;\Rightarrow\; \forall_{X'AY'} \exists_{XAY} : \; Y \propto Y')$$

Note that a $\propto$-monotone component is both best-case and worst-case $\propto$-monotone.

Furthermore, we define continuity of components as follows:

Definition 5.19 (Continuity). Let A be a component with input interface P and output interface Q and $\propto$ a relation on traces. Moreover let $\sup_\propto$ be the supremum and $\inf_\propto$ be the infimum of a

set of traces with respect to $\propto$ and let for any sub-relation $R'_A \subseteq R_A$ the respective sub-input and sub-output sets:

$$in'_A = \{X \in Tr(P) \mid \exists_{Y \in Tr(Q)} \colon (X, Y) \in R'_A\}$$
$$out'_A = \{Y \in Tr(Q) \mid \exists_{X \in Tr(P)} \colon (X, Y) \in R'_A\}$$

Then component A is $\propto$-continuous iff it holds that:

$$\forall_{R'_A \subseteq R_A} \colon (\sup_{\propto}(in'_A), \sup_{\propto}(out'_A)) \in R_A$$

Monotonicity and continuity are closely related, as can be deduced from the following corollary:

Corollary 5.20 (Monotonicity vs. Continuity). *If a component A is $\propto$-continuous then component A is also $\propto$-monotone.*

5.5 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 5.21 (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 5.22 (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 $\Theta : Q^*_A \to P^*_B$ be a bijective mapping. Then it holds that component B is connectible to component A given mapping Θ, i.e. $A \to_\Theta B$, iff it holds that P^*_B is connectible to Q^*_A given mapping Θ, i.e. $Q^*_A \to_\Theta P^*_B$.

This allows us to define serial composition as follows:

Definition 5.23 (Serial Composition). Let A and B be two components with disjoint input interfaces P_A and $P_B = P^\diamond_B \cup P^*_B$ and disjoint output interfaces $Q_A = Q^*_A \cup Q^\diamond_A$ and Q_B, respectively, with $P^\diamond_B \cap P^*_B = Q^*_A \cap Q^\diamond_A = \emptyset$. 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 Θ, i.e. $A \to_\Theta B$, then the serial composition of A and B given mapping Θ is obtained by connecting interface P^*_B to interface Q^*_A via an interface connection C^Θ. This results in a component $A\Theta B$ with input interface $P_{A\Theta B} = P_A \cup P^\diamond_B$, output interface $Q_{A\Theta B} = Q^\diamond_A \cup Q_B$ and the relation between input and output interfaces as follows:

$$R_{A\Theta B} = \{(X_A \cup X^\diamond_B, Y^\diamond_A \cup Y_B) \in Tr(P_{A\Theta B}) \times Tr(Q_{A\Theta B}) \mid \exists_{X_A A (Y^*_A \cup Y^\diamond_A)} \colon \exists_{(X^\diamond_B \cup C^\Theta(Y^*_A)) B Y_B} \wedge \forall_{X_A A (Y^{*\bullet}_A \cup Y^{\diamond\bullet}_A)} \colon \exists_{(X^\diamond_B \cup C^\Theta(Y^{*\bullet}_A)) B Y^\bullet_B}\}$$

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 5.24 (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 Θ, i.e. $A \to_\Theta A$, then the feedback composition of A given mapping Θ is obtained by connecting interface P_A^* to interface Q_A^* via an interface connection C^Θ. 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^* and the trace limit $\lim_{k\to\infty} Y_k$ according to Definition 5.16):

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

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

6 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 $\trianglelefteq$ and $\trianglelefteqslant$ 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.

6.1 Stream & Trace Bounding

The bounding of streams is defined as follows:

Definition 6.1 (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 [8], 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 6.2 (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.

6.2 Component Bounding

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

Definition 6.3 (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. $A \trianglelefteq A'$, 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. $A \trianglerighteq A'$, 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 $X'A'Y'$.

6.3 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 6.4 (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 6.5 (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 6.5 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 6.6 (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 $X'_{k+1} = (X^{\diamond\prime} \cup C^\Theta(Y^{*\prime}_k)) \in in_{A'}$. 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 $(X^\diamond \cup C^\Theta(Y^*_k))A(Y^*_{k+1} \cup Y^\diamond_{k+1})$ 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 $(X^\diamond \cup C^\Theta(Y^*_\infty))A(Y^*_\infty \cup Y^\diamond_\infty)$. Analogously $X^{\diamond\prime} \in in_{A'\Theta A'}$ implies that for any sequence $(X^{\diamond\prime} \cup C^\Theta(Y^{*\prime}_k))A'(Y^{*\prime}_{k+1} \cup Y^{\diamond\prime}_{k+1})$ with $Y^{*\prime}_0 = Y^{*,\infty}$ there exists a $Y'_\infty = \lim_{k\to\infty} Y'_k$ that defines a fixed point of the sequence, i.e. it holds that $(X^{\diamond\prime} \cup C^\Theta(Y^{*\prime}_\infty))A'(Y^{*\prime}_\infty \cup Y^{\diamond\prime}_\infty)$.

Now consider a sequence $(X^\diamond \cup C^\Theta(Y^*_k))A(Y^*_{k+1} \cup Y^\diamond_{k+1})$ that for any sequence $(X^{\diamond\prime} \cup C^\Theta(Y^{*\prime}_k))A'$ $(Y^{*\prime}_{k+1} \cup Y^{\diamond\prime}_{k+1})$ 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_∞ and Y'_∞ be the respective fixed points of such sequences. Then it holds that $Y_\infty \Delta Y'_\infty$ and thus also $Y^\diamond_\infty \Delta Y^{\diamond\prime}_\infty$. With $Y^\diamond = Y^\diamond_\infty$ and $Y^{\diamond\prime} = Y^{\diamond\prime}_\infty$ 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 6.4 to 6.6 finally prove the automatic lifting of bounding from component to graph level:

Theorem 6.7 (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 6.4 to 6.6 and the fact that graphs consist of parallel, serial and feedback compositions. □

6.4 Bounding Transitivity & Reflexivity

The presented definition of component bounding, Definition 6.3 generalizes the TETB component refinement, as it does not imply input acceptance preservation. While this generalization relaxes the requirement of input acceptance preservation on component level, it also comes at the cost that general transitivity does not hold for component bounding, i.e. with $\nabla{=}\trianglelefteq$ ($\nabla{=}\trianglerighteq$) it does not

hold in general that $A \nabla A' \nabla A'' \Rightarrow A \nabla A''$. Nevertheless, we can define and prove a restricted form of transitivity that is sufficient for our needs as follows:

Theorem 6.8 (Restricted Bounding Transitivity). *With $\nabla{=}\trianglelefteq$ ($\nabla{=}\trianglerighteq$) let $A \nabla A' \nabla A''$. Then it holds with $\Delta{=}\triangleleft$ ($\Delta{=}\triangleright$):*

$$\forall_{X \in in_A, X' \in in_{A'}, X'' \in in_{A''}} : \; X \,\Delta\, X' \,\Delta\, X'' \Rightarrow \forall_{X'' A'' Y''} \exists_{XAY} : \; Y \,\Delta\, Y''$$

This means that transitivity of component bounding holds given that the input traces of the respective components bound each other as well.

Proof. From $A \nabla A'$ and $X \,\Delta\, X'$ it follows with the definition of component bounding that $\forall_{X' A' Y'} \exists_{XAY} : \; Y \,\Delta\, Y'$. Likewise it follows from $A' \nabla A''$ and $X' \,\Delta\, X''$ that $\forall_{X'' A'' Y''} \exists_{X' A' Y'} : \; Y' \,\Delta\, Y''$. Combining the two statements and using the transitivity of trace bounding, i.e. $Y \,\Delta\, Y' \,\Delta\, Y'' \Rightarrow Y \,\Delta\, Y''$, results in $\forall_{X'' A'' Y''} \exists_{XAY} : \; Y \,\Delta\, Y''$, which concludes the proof. □

The restricted bounding transitivity allows us to create multiple layers of bounding components while still allowing to relate the temporal behavior of the highest and lowest levels to each other.

Next to transitivity also reflexivity of component bounding does not always hold, i.e. with $\nabla{=}\trianglelefteq$ ($\nabla{=}\trianglerighteq$) it does not hold in general that $A \nabla A$. Instead, reflexivity of bounding is tightly coupled to monotonicity, as shown by the following theorem:

Theorem 6.9 (Bounding Reflexivity vs. Monotonicity). *With $\nabla{=}\trianglelefteq$ ($\nabla{=}\trianglerighteq$) it holds that $A \nabla A$ is equivalent to A being best-case (worst-case) $\triangleleft$-monotone.*

Proof. Follows immediately from Definitions 6.3 and 5.18. □

The most important implication of the lack of general reflexivity is that Lemma 6.5 does neither imply $A' \Theta B \nabla A \Theta B$ nor $A \Theta B' \nabla A \Theta B$. Instead it must additionally hold that A is best-case (worst-case) $\triangleleft$-monotone if only B is replaced by B' or that B is best-case (worst-case) $\triangleleft$-monotone if only A is replaced by A'.

6.5 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 6.10 (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:

$$\forall_{X \in in_{G'}} : \; \forall_{X G' Y'} \exists_{XGY} : \; Y \,\Delta\, Y'$$

Definition 6.11 (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:

$$\forall_{X \in in_G} : \; \forall_{X G' Y'} \exists_{XGY} : \; Y \,\Delta\, Y'$$

Definition 6.10 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 6.11 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 [8]. 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 6.12 (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 6.3 and 6.10, as well as the fact that the $\lhd$ relation is reflexive, i.e. $X \lhd X$. □

Theorem 6.13 (Bounding Refinement). *A graph G' is a best-case (worst-case) refinement of a graph G if $G \trianglelefteq G'$ ($G \trianglerighteq G'$) and if $in_G \subseteq in_{G'}$.*

Proof. Follows immediately from Definitions 6.3 and 6.11, as well as the fact that the $\lhd$ relation is reflexive, i.e. $X \lhd X$. □

Theorem 6.14 (Bounding Abstraction-Refinement Transitivity). *With $\Delta{=}\trianglelefteq$ ($\Delta{=}\trianglerighteq$) 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. Follows immediately from Definitions 6.10 and 6.11, Theorems 6.8, 6.12 and 6.13, as well as the fact that the $\lhd$ relation is reflexive, i.e. $X \lhd X$. □

7 INCLUSION

This section discusses our inclusion abstraction and refinement theory, based on the component model introduced in Section 5. We begin with traces, then define when a component includes another and thereafter prove that inclusion is automatically lifted from component to graph level. Lastly, we combine component inclusion with input acceptance preservation and therewith define inclusion abstraction and refinement.

7.1 Trace Equality

Inclusion abstraction and refinement are about components that produce the same output traces for the same input traces. For that matter we first formally define when two traces are the same, i.e. equal:

Definition 7.1 (Trace Equality). Let $X, X' \in Tr(P)$ be two traces on an interface P. The equality relation = for traces is defined as:

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

7.2 Component Inclusion

Based on trace equality we define component inclusion, i.e. the condition under which one components includes another, such that it can produce at least the same output traces for the same input traces as follows:

Definition 7.2 (Component Inclusion). Let A, A' be two components with input interfaces $P_A = P_{A'}$ and output interfaces $Q_A = Q_{A'}$.

Component A includes A', i.e. $A \supseteq A'$, iff it holds $\forall_{X \in in_A, X' \in in_{A'}}$ that:

$$X = X' \Rightarrow \forall_{X'A'Y'} \exists_{XAY} \colon\ Y = Y'$$

In words this means that A includes A' iff for every input trace X of A that is equal to an input trace X' of A' it holds that for all output traces Y' with $X'A'Y'$ there exist equal output traces Y with XAY.

7.3 Inclusion Lifting

In this section we prove the automatic lifting of inclusion from component to graph level, i.e. that two graphs include each other if all their respective components include each other. As a graph consists of parallel, serial and / or feedback compositions we first prove the preservation of inclusion on these compositions.

We begin with parallel inclusion:

Lemma 7.3 (Parallel Inclusion Preservation). *Let $A \supseteq A'$ and $B \supseteq B'$. From this follows that the respective parallel compositions also include each other, i.e. $A||B \supseteq A'||B'$.*

Proof. Trivial. □

For serial inclusion we obtain analogously:

Lemma 7.4 (Serial Inclusion Preservation). *Let $A \supseteq A'$ and $B \supseteq B'$. From this follows that the respective serial compositions also include each other, i.e. $A \Theta B \supseteq A' \Theta B'$.*

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

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 $(X_A \cup X_B^\diamond) = (X'_A \cup X_B^{\diamond\prime})$. Thus it holds that also $X_A = X'_A$ and from $A \supseteq A'$ it follows:

$$\forall_{X'_A A'(Y_A^{*\prime} \cup Y_A^{\diamond\prime})} \exists_{X_A A(Y_A^* \cup Y_A^\diamond)} \colon\ (Y_A^* \cup Y_A^\diamond) = (Y_A^{*\prime} \cup Y_A^{\diamond\prime}) \tag{3}$$

$(X_A \cup X_B^\diamond) \in in_{A \Theta B}$ 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 3 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 $(X_A \cup X_B^\diamond) = (X'_A \cup X_B^{\diamond\prime})$ that $(X_B^\diamond \cup C^\Theta(Y_A^*)) = (X_B^{\diamond\prime} \cup C^\Theta(Y_A^{*\prime}))$. With $B \nabla B'$ we obtain:

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

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

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

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

Lastly, we prove preservation of inclusion on feedback composition:

Lemma 7.5 (Feedback Inclusion Preservation). *Let $A \supseteq A'$. From this follows that the respective feedback compositions also include each other, i.e $A \Theta A \supseteq A' \Theta A'$.*

Proof Idea. Let there be an input that is accepted by $A \Theta A$ and that is equal to an input that is accepted by $A' \Theta A'$. The respective combinations of these traces with empty traces on the internal interfaces are consequently also equal. From $A \supseteq A'$ then follows that for any respective output of A' there exists an equal output 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 that for any output of A' there exists an equal output of A. As the third line ensures that for any such equal sequences fixed points are reached, it follows that also the respective fixed points equal each other. This lets us conclude for any input of $A \Theta A$ that equals an input of $A' \Theta A'$ that for any respective output of $A' \Theta' A'$ there exists an equal output of $A \Theta A$, i.e. $A \Theta A \supseteq A' \Theta A'$.

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 equal to the external input of A' we first show for each feedback iteration that for all output traces of A' there exist equal output traces of A, using mathematical induction. Based on this we prove that for each fixed point of A' there exists a fixed point of A, such that the fixed points of A are equal to 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 $X^\diamond = X^{\diamond\prime}$. 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 $X^\diamond = X^{\diamond\prime}$ it holds that $X_0 = X'_0$. From $A \supseteq A'$ it then follows that there exists an $X_0 A Y_0$ for all $X'_0 A' Y'_0$ such that $Y_0 = Y'_0$.

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

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 equality it follows that $X_{k+1} = X'_{k+1}$ and $A \supseteq A'$ 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 $Y_{k+1} = Y'_{k+1}$.

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

$$\forall_{k \in \mathbb{N}} \colon \forall_{X'_k A' Y'_k} \exists_{X_k A Y_k} \colon X_k = X'_k \wedge Y_k = Y'_k \tag{4}$$

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 $Y_0^{*\prime} = Y^{*\prime,\infty}$ 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 sequences $(X^\diamond \cup C^\Theta(Y_k^*)) A (Y_{k+1}^* \cup Y_{k+1}^\diamond)$ that for any sequence $(X^{\diamond\prime} \cup C^\Theta(Y_k^{*\prime})) A'$ $(Y_{k+1}^{*\prime} \cup Y_{k+1}^{\diamond\prime})$ satisfy $X_k = X'_k$ and $Y_k = Y'_k$ for all $k \in \mathbb{N}$ (existence of such sequences is guaranteed by Equation 4). Furthermore, let Y_∞ and Y'_∞ be the respective fixed points of such sequences. Then it holds that $Y_\infty = Y'_\infty$ and thus also $Y_\infty^\diamond = Y_\infty^{\diamond\prime}$. With $Y^\diamond = Y_\infty^\diamond$ and $Y^{\diamond\prime} = Y_\infty^{\diamond\prime}$ this lets us finally

conclude $\forall_{X^\diamond \in in_{A\Theta A}, X^{\diamond\prime} \in in_{A'\Theta A'}}$:

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

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

Based on the preservation of inclusion on parallel, serial and feedback composition we can finally prove the automatic lifting of inclusion from component to graph level as follows:

Theorem 7.6 (Component Graph Inclusion). *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 including all respective components of G', i.e. $A \supseteq A'$, then it follows that also the graph G includes graph G', i.e. $G \supseteq G'$.*

Proof. Follows immediately from Lemmas 7.3 to 7.5 and the fact that graphs consist of parallel, serial and feedback compositions. □

7.4 Inclusion Transitivity & Reflexivity

First we prove that component inclusion is transitive:

Theorem 7.7 (Inclusion Transitivity). *Let $A \supseteq A' \supseteq A''$. This implies that $A \supseteq A''$, i.e. component inclusion is transitive.*

Proof. Let $X = X'$. From $A \supseteq A'$ and the definition of component inclusion follows that $\forall_{X'A'Y'} \exists_{XAY} \colon\ Y = Y'$. Likewise it follows from $X' = X''$ and $A' \supseteq A''$ that $\forall_{X''A''Y''} \exists_{X'A'Y'} \colon\ Y' = Y''$. From combining the two statements and taking into account that trace equality is transitive it can be concluded $\forall_{X \in in_A, X'' \in in_{A''}}$ that:

$$X = X'' \Rightarrow \forall_{X''A''Y''} \exists_{XAY} \colon\ Y = Y''$$

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

Next to that, component inclusion is also reflexive, as stated by the following theorem:

Theorem 7.8 (Inclusion Reflexivity). *It holds for any component A that $A \subseteq A$.*

Proof. Trivial. □

This implies that it follows from $A \supseteq A'$ unconditionally that $A\Theta B \supseteq A'\Theta B$, as well as it follows from $B \supseteq B'$ unconditionally that $A\Theta B \supseteq A\Theta B'$.

7.5 Inclusion Abstraction & Refinement

After defining equality of traces, inclusion of components, as well as proving the automatic lifting of inclusion from component to graph level we can now also define inclusion abstraction and refinement. Thereafter we discuss that inclusion abstraction and refinement can be directly concluded from input acceptance preservation and inclusion.

Definition 7.9 (Inclusion Abstraction). A graph G is an inclusion 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 each output trace of G' there exists an equal output trace of G, i.e.:

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

Definition 7.10 (Inclusion Refinement). A graph G' is an inclusion 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 each output trace of G' there exists an equal output trace of G, i.e.:

$$\forall_{X \in in_G} \colon\ \forall_{XG'Y'} \exists_{XGY} \colon\ Y = Y'$$

Analogous to bounding abstraction and refinement, it holds that Definition 7.9 applies to the creation of models for the analysis of an existing implementation (for which the specification lies in the implementation), while Definition 7.10 applies to the design process from a model to an implementation (for which the specification is given by the models).

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

Theorem 7.11 (Inclusion Abstraction). *A graph G is an inclusion abstraction of a graph G' if it holds that $G \supseteq G'$ and if $in_G \supseteq in_{G'}$.*

Proof. Follows immediately from Definitions 7.2 and 7.9. □

Theorem 7.12 (Inclusion Refinement). *A graph G' is an inclusion refinement of a graph G if it holds that $G \supseteq G'$ and if $in_G \subseteq in_{G'}$.*

Proof. Follows immediately from Definitions 7.2 and 7.10. □

Theorem 7.13 (Inclusion Abstraction & Refinement Transitivity). *Let it hold for three graphs G, G' and G'' that $G \supseteq G' \supseteq G''$ and that $in_G \supseteq in_{G'} \supseteq in_{G''}$. Then it holds that G is an inclusion abstraction of G''. If it holds instead that $in_G \subseteq in_{G'} \subseteq in_{G''}$ then it follows that G'' is an inclusion refinement of G.*

Proof. Follows immediately from Definitions 7.9 and 7.10, as well as Theorems 7.7, 7.11 and 7.12. □

8 INPUT ACCEPTANCE & REPLACEABILITY

In the TETB refinement theory the definition of refinement implies that given component refinement the input acceptance of components is automatically lifted from component to graph level. Moreover, the refinement relation is defined in such way that it is always possible to replace components by their refinements without reducing the input acceptance of the entire graph. In our theory both these implications do not hold in general, which is due to the fact that on component level we only require bounding or inclusion instead of abstraction or refinement. In the following we consequently derive additional component properties for which both automatic lifting of input acceptance from component to graph level and replaceability can be shown. For that matter we introduce two important subclasses of components, input-complete and operational components, which are defined as follows:

Definition 8.1 (Input-Complete Component). A component A with input interface P is input-complete iff it holds that it accepts all input traces, i.e. $in_A = Tr(P)$.

Definition 8.2 (Operational Component). A component A is operational iff it is continuous with respect to the prefix relation, i.e. it is $\preceq$-continuous.

8.1 Input Acceptance Lifting

In this section we derive conditions under which an automatic lifting of input acceptance from component to graph level is given, i.e. under which input acceptance is preserved on parallel, serial and feedback composition. This is of particular importance for the verification of input acceptance preservation on abstraction or refinement, as an automatic lifting of input acceptance allows to deduce graph level input acceptance preservation from component level input acceptance preservation. Afterwards we show that on compositions of input-complete and operational components the resulting components are also input-complete. First we define some notation that is used throughout the following sections.

Definition 8.3 (Partial trace sets). Let $Tr \subseteq Tr(P)$ be a set of traces on an interface P. We denote the set of streams on a port $p \in P$ that are part of traces in Tr as:

$$Tr[p] = \{x \in St(p) \mid \exists_{X \in Tr} \colon\ x = X[p]\}$$

Likewise, we denote the set of traces on an interface $P' \subseteq P$ that are part of traces in Tr as:

$$Tr[P'] = \{X' \in Tr(P') \mid \exists_{X \in Tr} \colon\ \forall_{p \in P'} \colon\ X'[p] = X[p]\}$$

Definition 8.4 (Output Set Connection). Let A be a component whose output interface $Q_A^* \subseteq Q_A$ is connected to another component interface P_B^* via an interface connection C^{Θ}. We denote the set of valid output traces of A with respect to interface P_B^* as:

$$C^{\Theta}(out_A^*) = \{X \in Tr(P_B^*) \mid \exists_{Y \in out_A^*} \colon\ X = C^{\Theta}(Y)\}$$

Definition 8.5 (Compositional Trace Relation). A relation $\propto$ on traces is compositional iff it is also defined for streams, such that it holds on any interface P that $X \propto X' \Leftrightarrow \forall_{p \in P} \colon\ X[p] \propto X'[p]$.

Corollary 8.6 (Compositional Extrema). *Let $\propto$ be a compositional trace relation. Then it holds that extrema with respect to this relation are preserved on merging and splitting of interfaces, i.e. it holds for any set of traces Tr on an interface $P = P' \cup P''$ that:*

$$\sup{}_{\propto}(Tr) = \sup{}_{\propto}(Tr[P']) \cup \sup{}_{\propto}(Tr[P''])$$
$$\inf{}_{\propto}(Tr) = \inf{}_{\propto}(Tr[P']) \cup \inf{}_{\propto}(Tr[P''])$$

Besides these definitions we introduce the subclasses of input-independent and empty-continuous components and subsequently establish their relations to input-complete and operational components:

Definition 8.7 (Input-Independent Component). A component A with input interface P_A is input-independent iff it holds that input acceptance per port is independent, i.e. $in_A = \bigotimes_{p \in P_A} in_A[p]$.

Corollary 8.8 (Input-Complete vs. Input-Independent). *An input-complete component is input-independent.*

Definition 8.9 (Empty-Continuous Component). A component A with output interface Q_A is empty-continuous iff it holds that the component is continuous with respect to a reflexive, compositional trace relation $\propto$ for which the empty trace X^{∞} on any interface P is a global infimum according to $\propto$, i.e. $X^{\infty} = \inf{}_{\propto}(Tr(P))$.

Corollary 8.10 (Operational vs. Empty-Continuous). *An operational component is empty-continuous.*

Using these subclasses we define sufficient conditions for the automatic lifting of input acceptance. We first prove that under certain conditions input acceptance is preserved on parallel, feedback and serial composition and afterwards discuss input acceptance lifting from component to graph level.

Lemma 8.11 (Parallel Input Acceptance Preservation). *Let A and B be two components and $A||B$ a parallel composition according to Definition 5.21. Then the input acceptance of A and B is preserved on parallel composition $A||B$, i.e. it holds that $in_{A||B} = in_A \times in_B$. Moreover, the parallel composition $A||B$ is input-independent / empty-continuous with respect to a relation $\propto$ if both A and B are input-independent / empty-continuous with respect to the same relation.*

Proof. Trivial. □

Lemma 8.12 (Serial Input Acceptance Preservation). *Let A and B be two components and $A \Theta B$ a serial composition according to Definition 5.23. Then the input acceptance of A and B is preserved on serial composition $A \Theta B$, i.e. it holds that $in_{A\Theta B} = in_A \times in_B^\diamond$, if B accepts all outputs of A, i.e. $C^\Theta(out_A^*) \subseteq in_B^*$, and if B is input-independent. Moreover, the serial composition $A \Theta B$ is input-independent if A is also input-independent. Finally, $A \Theta B$ is empty-continuous with respect to a relation $\propto$ if both A and B are empty-continuous with respect to the same relation.*

Proof Idea. From the input-independence of B and $C^\Theta(out_A^*) \subseteq in_B^*$ it follows that B accepts all combinations of external inputs and outputs of A, i.e. input acceptance of A and B is preserved on serial composition $A \Theta B$. Showing that $A \Theta B$ is input-independent if A is additionally input-independent is trivial.

Lastly, to prove empty-continuity of $A \Theta B$ if both A and B are additionally empty-continuous, we have to show that for any sub-relation of the relation of $A \Theta B$ the supremum of the respective sub-input set results in the supremum of the respective sub-output set. For that matter we derive the sub-relations of A and B corresponding to the sub-relation of $A \Theta B$ and the respective sub-input and sub-output sets of A and B. Then we use the empty-continuity of A to show that the supremum of the sub-input set of A results in the supremum of the sub-output set of A. We further show that this supremum of the sub-output set of A results in the supremum of the sub-input set of B. With the empty-continuity of B we can then conclude that this supremum results in the respective supremum of the sub-output set of B. This proves that for any sub-relation of $A \Theta B$ the input suprema are accepted and result in the respective output suprema, i.e. $A \Theta B$ is empty-continuous.

Proof. As B is input-independent it holds that $in_B = in_B^\diamond \times in_B^*$. With $C^\Theta(out_A^*) \subseteq in_B^*$ this implies that $in_B^\diamond \times C^\Theta(out_A^*) \subseteq in_B$. From this follows that it holds $\forall_{X_A \in in_A, X_B^\diamond \in in_B^\diamond}$:

$$\forall_{X_A A (Y_A^* \cup Y_A^\diamond)} \exists_{(X_B^\diamond \cup C^\Theta(Y_A^*)) B Y_B}$$

This lets us conclude with the definition of serial composition that $R_{A\Theta B}$ is defined $\forall_{X_A \in in_A, X_B^\diamond \in in_B^\diamond}$, i.e. it holds that $in_{A\Theta B} = in_A \times in_B^\diamond$.

If also A is input-independent it holds additionally that $in_A = \bigotimes_{p \in P_A} in_A[p]$ and as B is input-independent that $in_B^\diamond = \bigotimes_{p \in P_B^\diamond} in_B[p]$, from which follows that $A \Theta B$ is input-independent.

Lastly, consider that A and B are in addition both empty-continuous with respect to a relation $\propto$. In the following we make use of the fact that by definition $\propto$ must be a compositional trace relation, such that Corollary 8.6 holds, i.e. suprema are preserved on merging and splitting of interfaces. For any sub-relation $R'_{A\Theta B} \subseteq R_{A\Theta B}$ let $R'_A \subseteq R_A$ and $R'_B \subseteq R_B$ be the respective sub-relations of A and B according to the definition of serial composition, such that:

$$(X_A \cup X_B^\diamond, Y_A^\diamond \cup Y_B) \in R'_{A\Theta B} \;\Leftrightarrow\; (X_A, Y_A^* \cup Y_A^\diamond) \in R'_A \wedge (X_B^\diamond \cup C^\Theta(Y_A^*), Y_B) \in R'_B \tag{5}$$

Furthermore let $in'_{A\Theta B}$, $out'_{A\Theta B}$, in'_A, out'_A, in'_B and out'_B according to Definition 5.19.

We need to prove that $(\sup_\propto(in'_{A\Theta B}), \sup_\propto(out'_{A\Theta B})) \in R_{A\Theta B}$. For that matter let us consider $\sup_\propto(in'_{A\Theta B})$ as input of $A \Theta B$. From Equation 5 it follows that $in'_A = in'_{A\Theta B}[P_A]$ and therefore also $\sup_\propto(in'_A) = \sup_\propto(in'_{A\Theta B})[P_A]$. With the empty-continuity of A we obtain that $\sup_\propto(in'_A)$ is accepted by A, as well as that $\sup_\propto(in'_A) A \sup_\propto(out'_A)$. From Equation 5 it can be further deduced that $in'_B[P_B^\diamond] = in'_{A\Theta B}[P_B^\diamond]$ and $in'_B[P_B^*] = C^\Theta(out'_A[Q_A^*])$. With Corollary 8.6 it then follows that $\sup_\propto(in'_B) = (\sup_\propto(in'_{A\Theta B})[P_B^\diamond] \cup C^\Theta(\sup_\propto(out'_A)[Q_A^*]))$.

With the empty-continuity of B we obtain that $\sup_\propto(in'_B)$ is accepted by B, as well as that $\sup_\propto(in'_B) B \sup_\propto(out'_B)$. From Equation 5 we get $out'_{A\Theta B}[Q_A^\diamond] = out'_A[Q_A^\diamond]$ and $out'_{A\Theta B}[Q_B] = out'_B$ and with Corollary 8.6 it follows that $\sup_\propto(out'_{A\Theta B}) = (\sup_\propto(out'_A)[Q_A^\diamond] \cup \sup_\propto(out'_B))$.

As the respective suprema are accepted by both A and B, as $\sup_{\propto}(in'_{A\Theta B})$ is accepted by $A\Theta B$ due to the previously proven input acceptance preservation and as for this input $A\Theta B$ has the output $\sup_{\propto}(out'_{A\Theta B})$, i.e. $(\sup_{\propto}(in'_{A\Theta B}), \sup_{\propto}(out'_{A\Theta B})) \in R_{A\Theta B}$, we can finally conclude that $A\Theta B$ is empty-continuous. □

Lemma 8.13 (Feedback Input Acceptance Preservation). *Let A be a component and $A\Theta A$ a feedback composition according to Definition 5.24. Then the input acceptance of A is preserved on feedback composition $A\Theta A$, i.e. it holds that $in_{A\Theta A} = in^{\diamond}_{A}$, if A accepts all outputs of A, i.e. $C^{\Theta}(out^{*}_{A}) \subseteq in^{*}_{A}$, and if A is both input-independent and empty-continuous with respect to a relation $\propto$. Moreover, the feedback composition $A\Theta A$ is also both input-independent and empty-continuous with respect to relation $\propto$ (note that proving input-independence and empty-continuity of $A\Theta A$ is not strictly needed for proving input acceptance lifting from component to graph level as feedback composition can always be done in a single last step after all parallel and serial compositions).*

Proof Idea. Let there be an external input that is, together with an internal input, accepted by A. Following the definition of feedback composition, we reason in sequences of inputs and outputs on the respective internal input and output interfaces. Thereby we make use of four properties to prove that the external input is also accepted by $A\Theta A$, i.e. that input acceptance is preserved. First, the starting point of all sequences, i.e. the combination of the external input with the empty trace is accepted by A, due to the empty trace being accepted by any component and A being input-independent. Second, any combination of the external input with an internal output of A is also accepted by A, due to A being input-independent and $C^{\Theta}(out^{*}_{A}) \subseteq in^{*}_{A}$. These first two properties ensure that all input sequences are accepted by A, i.e. no dead states can occur. Third, starting from the empty trace on the internal interface, all following inputs and outputs are $\propto$-larger than the last, as empty-continuity implies that the empty trace is the global infimum and that A is $\propto$-monotone. This ensures that both input and output sequences keep increasing, i.e. A cannot get into an oscillating state for any sequence. And fourth, convergence to fixed points of all input-output sequences is guaranteed, as the fixed point of any input-output sequence is just the supremum of all feedback iterations of that sequence and as the empty-continuity of A implies acceptance of suprema. This ensures that the respective limits exist for all sequences, which proves input acceptance preservation.

Proving input-independence is again trivial with the input acceptance preservation of A and A being input-independent.

To prove empty-continuity of $A\Theta A$ we have to show that for any sub-relation of the relation of $A\Theta A$ the supremum of the respective sub-input set results in the supremum of the respective sub-output set. According to the definition of feedback composition, any element of a sub-relation of $A\Theta A$ is a fixed point of a certain input-output sequence of A, starting with the empty trace on the internal interface. For any such sequence we can now construct sub-relations of A such that the k'th sub-relation contains all input-output pairs that represent the k'th feedback iteration of the sequences leading to aforementioned fixed points. For instance, the first sub-relation of A contains all pairs of the first inputs and outputs of the sequences, with the first inputs combining the empty trace with the respective external inputs in the sub-input set of $A\Theta A$. Then we consider the supremum of the sub-input set of $A\Theta A$ as input of A. By repetitively using the empty-continuity of A with respect to $\propto$ for all sub-relations of A we show that the supremum of the sub-input set of $A\Theta A$ also results in a fixed point of A, which is the supremum of the sub-output set of $A\Theta A$. This proves that $A\Theta A$ is empty-continuous with respect to relation $\propto$.

Proof. To prove input acceptance preservation we have to show, according to the definition of feedback composition, that $\forall_{X^{\diamond} \in in^{\diamond}_{A}}$ it holds that $Y^{*}_{\infty} \cup Y^{\diamond}_{\infty}$ exists for all sequences ($X^{\diamond} \cup$

$C^{\Theta}(Y^*_{k-1}))A(Y^*_k \cup Y^\diamond_k)$ with $Y^*_{-1} = Y^{*,\infty}$ such that $\lim_{k\to\infty} Y^*_k \cup Y^\diamond_k = Y^*_\infty \cup Y^\diamond_\infty$. For that matter we need to prove that each $X^\diamond \cup C^{\Theta}(Y^*_k)$ is accepted by A and that each sequence converges for $k \to \infty$, i.e. it does not get into an oscillating state (note that "oscillating" in this context does not mean that values of a trace may be oscillating indefinitely in time, but that entire traces are oscillating).

In the following we denote with $X_k = (X^\diamond \cup X^*_k)$ and $Y_k = (Y^*_k \cup Y^\diamond_k)$ the input and output traces of component A in each feedback iteration. Moreover, we denote with $\{X_k\} = \bigcup_{0\leq k'<k+1} X_{k'}$ and $\{Y_k\} = \bigcup_{0\leq k'<k+1} Y_{k'}$ sets containing all respective input and output traces up to and including k.

Assuming $X^\diamond \in in^\diamond_A$ we first prove using mathematical induction that for all k all $X_k A Y_k$ exist and that it holds for all k that $X_k \propto X_{k+1}$ and $Y_k \propto Y_{k+1}$, i.e. A cannot get into an oscillating state. The latter implies for all k that $\sup_\propto(\{X_k\}) = X_k$, as well as $\sup_\propto(\{Y_k\}) = Y_k$. Based on this observation we show that also $\sup_\propto(\{X_\infty\}) = X_\infty$ and $\sup_\propto(\{Y_\infty\}) = Y_\infty$ exist.

Induction base: Let $X^\diamond \in in^\diamond_A$. With A accepting the empty trace by definition and with A being input-independent, i.e. $in_A = in^\diamond_A \times in^*_A$, it follows that $X_0 = (X^\diamond \cup C^{\Theta}(Y^{*,\infty})) \in in_A$, i.e. $X_0 A Y_0$. With $Y_0 = (Y^*_0 \cup Y^\diamond_0)$ we have $X_1 = (X^\diamond \cup C^{\Theta}(Y^*_0))$. According to the definition of empty-continuity it holds that $Y^{*,\infty}$ is the global infimum with respect to $\propto$. From this follows that $Y^{*,\infty} \propto Y^*_0$ and with $\propto$ being both reflexive and compositional that also $X_0 \propto X_1$. From $C^{\Theta}(out^*_A) \subseteq in^*_A$ and the input-independence of A we further obtain that $X_1 \in in_A$, i.e. $X_1 A Y_1$. Finally we have that according to Corollary 5.20 the continuity of A with respect to $\propto$ implies monotonicity with respect to $\propto$, such that we can conclude from $X_0 \propto X_1$ that also $Y_0 \propto Y_1$.

Induction hypothesis: Let $X_{k-1} A Y_{k-1}$ and $X_k A Y_k$ and let $X_{k-1} \propto X_k$, as well as $Y_{k-1} \propto Y_k$.

Induction step: With $Y_{k-1} = (Y^*_{k-1} \cup Y^\diamond_{k-1})$, $X_k = (X^\diamond \cup C^{\Theta}(Y^*_{k-1}))$, $Y_k = (Y^*_k \cup Y^\diamond_k)$ and $X_{k+1} = (X^\diamond \cup C^{\Theta}(Y^*_k))$ we get from $Y^*_{k-1} \propto Y^*_k$ and $\propto$ being reflexive and compositional that $X_k \propto X_{k+1}$. From $C^{\Theta}(out^*_A) \subseteq in^*_A$ and the input-independence of A we further obtain that $X_{k+1} \in in_A$, i.e. $X_{k+1} A Y_{k+1}$. As the continuity of A with respect to $\propto$ implies monotonicity with respect to $\propto$ we finally get from $X_k \propto X_{k+1}$ that also $Y_k \propto Y_{k+1}$.

From the induction we can thus conclude that if $X^\diamond \in in^\diamond_A$ it holds for all k that all $X_k A Y_k$ exist and that $X_k \propto X_{k+1}$, as well as $Y_k \propto Y_{k+1}$. From the latter we further obtain for all k that $\sup_\propto(\{X_k\}) = X_k$ and that $\sup_\propto(\{Y_k\}) = Y_k$. Now let $R'_A = R_{A,k} \subseteq R_A$ such that $in'_A = \sup_\propto(\{X_k\})$ and $out'_A = \sup_\propto(\{Y_k\})$ according to Definition 5.19. The continuity of A with respect to $\propto$ implies that $\sup_\propto(in'_A) A \sup_\propto(out'_A)$ must exist for all $R'_A \subseteq R_A$, even if the number of elements in R'_A is infinite and if $\sup_\propto(in'_A) \notin in'_A$. This finally allows us to conclude that if $X^\diamond \in in^\diamond_A$ that also $\sup_\propto(\{X_\infty\}) A \sup_\propto(\{Y_\infty\})$, i.e. $X_\infty A Y_\infty$, exists for any sequence of X_k and Y_k, which proves input acceptance preservation.

Moreover, with $in_{A\Theta A} = in^\diamond_A$ and A being input-independent it immediately follows that $in_{A\Theta A} = \bigotimes_{p\in P^\diamond_A} in_A[p]$, i.e. $A\Theta A$ is also input-independent.

Lastly, we prove that $A\Theta A$ is empty-continuous with respect to $\propto$. In the following we make use of the fact that by definition $\propto$ must be a compositional trace relation, such that Corollary 8.6 holds, i.e. suprema are preserved on merging and splitting of interfaces. For any sub-relation $R'_{A\Theta A} \subseteq R_{A\Theta A}$ let $\forall_k\colon\ R'_{A,k} \subseteq R_A$ and $R'_{A,\infty} \subseteq R_A$ be the respective sub-relations of A according to the definition of feedback composition, such that with $Y^*_{-1} = Y^{*,\infty}$, as well as $\lim_{k\to\infty} Y^*_k \cup Y^\diamond_k = Y^*_\infty \cup Y^\diamond_\infty$:

$$(X^\diamond, Y^\diamond_\infty) \in R'_{A\Theta A} \;\Leftrightarrow\; \forall_k\colon\ (X^\diamond \cup C^{\Theta}(Y^*_{k-1}), Y^*_k \cup Y^\diamond_k) \in R'_{A,k} \wedge (X^\diamond \cup C^{\Theta}(Y^*_\infty), Y^*_\infty \cup Y^\diamond_\infty) \in R'_{A,\infty} \quad (6)$$

Furthermore let $in'_{A\Theta A}$, $out'_{A\Theta A}$, $in'_{A,k}$, $out'_{A,k}$, $in'_{A,\infty}$ and $out'_{A,\infty}$ for all k according to Definition 5.19. In the following we need to prove that $(\sup_\propto(in'_{A\Theta A}), \sup_\propto(out'_{A\Theta A})) \in R_{A\Theta A}$. For that matter we use the same notation as above and make use of an induction-based proof again.

Induction base: For an $R'_{A\Theta A} \subseteq R_{A\Theta A}$ let $\sup_{\propto}(in'_{A\Theta A})$ be an input of $A\Theta A$. From Equation 6 it follows that $in'_{A,0}[P_A^{\diamond}] = in'_{A\Theta A}$ and that $in'_{A,0}[P_A^{*}] = \{C^{\Theta}(Y^{*,\infty})\}$. Consequently it also holds that $\sup_{\propto}(in'_{A,0})[P_A^{\diamond}] = \sup_{\propto}(in'_{A\Theta A})$, as well as $\sup_{\propto}(in'_{A,0})[P_A^{*}] = C^{\Theta}(Y^{*,\infty})$. With the empty-continuity of A we get that $\sup_{\propto}(in'_{A,0})$ is accepted by A and that $\sup_{\propto}(in'_{A,0}) A \sup_{\propto}(out'_{A,0})$.

Induction hypothesis: Let $sup_{\propto}(in'_{A,k})$ be accepted by A and let $\sup_{\propto}(in'_{A,k}) A \sup_{\propto}(out'_{A,k})$.

Induction step: From Equation 6 it follows that $in'_{A,k+1}[P_A^{\diamond}] = in'_{A\Theta A}$ and that $in'_{A,k+1}[P_A^{*}] = C^{\Theta}(out'_{A,k}[Q_A^{*}])$. Thus it also holds that $\sup_{\propto}(in'_{A,k+1})[P_A^{\diamond}] = \sup_{\propto}(in'_{A\Theta A})$ and $\sup_{\propto}(in'_{A,k+1})[P_A^{*}] = C^{\Theta}(\sup_{\propto}(out'_{A,k}[Q_A^{*}]))$. With the empty-continuity of A we finally obtain that $\sup_{\propto}(in'_{A,k+1})$ is accepted by A, as well as that $\sup_{\propto}(in'_{A,k+1}) A \sup_{\propto}(out'_{A,k+1})$.

From the induction we can conclude that if $R'_{A\Theta A} \subseteq R_{A\Theta A}$ it holds for all k that $\sup_{\propto}(in'_{A,k})$ is accepted by A, as well as that $\sup_{\propto}(in'_{A,k}) A \sup_{\propto}(out'_{A,k})$, such that $\sup_{\propto}(in'_{A,k})[P_A^{\diamond}] = \sup_{\propto}(in'_{A\Theta A})$ and $\sup_{\propto}(in'_{A,k})[P_A^{*}] = C^{\Theta}(\sup_{\propto}(out'_{A,k-1})[Q_A^{*}])$.

Finally, it follows from Equation 6 that $in'_{A,\infty}[P_A^{\diamond}] = in'_{A\Theta A}$ and that $in'_{A,\infty}[P_A^{*}] = C^{\Theta}(out'_{A,\infty}[Q_A^{*}])$. Thus it also holds that $\sup_{\propto}(in'_{A,\infty})[P_A^{\diamond}] = \sup_{\propto}(in'_{A\Theta A})$, as well as that $\sup_{\propto}(in'_{A,\infty})[P_A^{*}] = C^{\Theta}(\sup_{\propto}(out'_{A,\infty}[Q_A^{*}]))$. With the empty-continuity of A we obtain that $\sup_{\propto}(in'_{A,\infty})$ is accepted by A and that it moreover holds that $\sup_{\propto}(in'_{A,\infty}) A \sup_{\propto}(out'_{A,\infty})$. According to Equation 6 it further follows that $out'_{A\Theta A} = out'_{A,\infty}[Q_A^{\diamond}]$ and therewith $\sup_{\propto}(out'_{A\Theta A}) = \sup_{\propto}(out'_{A,\infty})[Q_A^{\diamond}]$.

This lets us conclude that if $R'_{A\Theta A} \subseteq R_{A\Theta A}$ it holds for all k that $\sup_{\propto}(in'_{A,k})$ is accepted by A and that $\sup_{\propto}(in'_{A,k}) A \sup_{\propto}(out'_{A,k})$, as well as that it holds that $\sup_{\propto}(in'_{A,\infty})$ is accepted by A and that $\sup_{\propto}(in'_{A,\infty}) A \sup_{\propto}(out'_{A,\infty})$. From the definition of feedback composition it consequently follows that $\sup_{\propto}(in'_{A\Theta A})$ is accepted by $A\Theta A$ and that $(\sup_{\propto}(in'_{A\Theta A}), \sup_{\propto}(out'_{A\Theta A})) \in R_{A\Theta A}$, i.e. $A\Theta A$ is empty-continuous. □

These Lemmas allow us to establish the automatic lifting of input acceptance from component to graph level as follows:

Theorem 8.14 (Input Acceptance Lifting). *Let G be a component graph consisting of components A with input interface $P_G = \bigcup_A P_A^{\diamond}$. If the output sets of all components are subsets of the input sets of the respective connected components, if all components are input-independent and if all components are empty-continuous with respect to the same relation $\propto$, then it holds that the input acceptance of the individual components is lifted to the graph level, i.e. $in_G = \bigotimes_A in_A^{\diamond}$, as well as that G is input-independent and empty-continuous with respect to relation $\propto$.*

Proof. Follows immediately from Lemmas 8.11 to 8.13 and the fact that graphs consist of parallel, serial and feedback compositions. □

Theorem 8.15 (Input-Completeness Lifting). *Let G be a component graph consisting of input-complete and operational components. Then it holds that the graph G is also input-complete and operational.*

Proof. Follows immediately from Corollaries 8.8, 8.10 and Theorem 8.14. □

Finally we introduce the concept of strict input acceptance preservation that we use in the next section.

Definition 8.16 (Strictly Input Acceptance Preserving Graph). We call a graph G strictly input acceptance preserving iff all its component compositions adhere to the requirements of Lemmas 8.11 to 8.13.

Corollary 8.17 (Input-Completeness vs. Strict Input Acceptance Preservation). *A graph consisting of input-complete and operational components is strictly input acceptance preserving.*

Corollary 8.18 (Strictly Input Acceptance Preserving Transformation). *Let G be a graph consisting of input-independent and empty-continuous components. If we replace in G all components A, B in serial compositions $A\Theta B$ with components A', B' such that $R_{A'} \subseteq R_A$, $R_{B'} \subseteq R_B$, $C^{\Theta}(out^*_{A'}) \subseteq in^*_{B'}$ and $R_{A'\Theta B'} = R_{A\Theta B}$ and all components A in feedback compositions $A\Theta A$ with components A' such that $R_{A'} \subseteq R_A$, $C^{\Theta}(out^*_{A'}) \subseteq in^*_{A'}$ and $R_{A'\Theta A'} = R_{A\Theta A}$ then we obtain a strictly input acceptance preserving graph G' for which it holds that $R_{G'} = R_G$.*

8.2 Replaceability

In the TETB refinement theory, refinement is defined in such way that input acceptance of a graph is preserved on a one-by-one replacement of refining components. By only requiring bounding or inclusion instead of abstraction or refinement of individual components, we lose this property. For that reason we have to separately derive conditions under which a component of a graph can be replaced by another component such that the input acceptance of the entire graph is preserved. We define replaceability as follows:

Definition 8.19 (Replaceability). Let component A be part of a component graph G. We say that component A is replaceable by a component A' iff the input set of the resulting graph G' after the replacement is larger or equal to the original input set, i.e. $in_{G'} \supseteq in_G$.

In the following we derive conditions for component replaceability under parallel, serial and feedback composition. Based on these conditions we then discuss replaceability of components in graphs.

Lemma 8.20 (Replaceability on Parallel Composition). *Let A and B be two components and $A||B$ a parallel composition according to Definition 5.21. If component A is replaced by a component A' with the same input and output interfaces as A and with $in_{A'} \supseteq in_A$ then it holds that $in_{A'||B} \supseteq in_{A||B}$.*

Proof. Trivial. □

Lemma 8.21 (Replaceability on Serial Composition). *Let A and B be two components and $A\Theta B$ a serial composition according to Definition 5.23.*

If A is replaced by a component A' with the same input and output interfaces as A, if B is input-independent, if $in_{A'} \supseteq in_A$ and if $C^{\Theta}(out^*_{A'}) \subseteq in^*_B$ then it holds that $in_{A'\Theta B} \supseteq in_{A\Theta B}$.

Moreover, if B is replaced by an input-independent component B' with the same input and output interfaces as B, if $in^{\diamond}_{B'} \supseteq in^{\diamond}_B$ and if $in^*_{B'} \supseteq C^{\Theta}(out^*_A)$ then it holds that $in_{A\Theta B'} \supseteq in_{A\Theta B}$.

Proof. In the first case in which A is replaced by A' it holds that Lemma 8.12 applies after the replacement, such that $in_{A'\Theta B} = in_{A'} \times in^{\diamond}_B$. As it furthermore holds that $in_{A\Theta B} \subseteq in_A \times in^{\diamond}_B$ it follows with $in_{A'} \supseteq in_A$ that $in_{A'\Theta B} \supseteq in_{A\Theta B}$.

In the second case in which B is replaced by B' it also holds that Lemma 8.12 applies after the replacement, such that $in_{A\Theta B'} = in_A \times in^{\diamond}_{B'}$. As it furthermore holds that $in_{A\Theta B} \subseteq in_A \times in^{\diamond}_B$ it follows with $in^{\diamond}_{B'} \supseteq in^{\diamond}_B$ that $in_{A\Theta B'} \supseteq in_{A\Theta B}$. □

Lemma 8.22 (Replaceability on Feedback Composition). *Let A be a component and $A\Theta A$ a feedback composition according to Definition 5.24. If A is replaced by an input-independent and empty-continuous component A' with the same input and output interfaces as A, if $in^{\diamond}_{A'} \supseteq in^{\diamond}_A$ and if $C^{\Theta}(out^*_{A'}) \subseteq in^*_{A'}$ then it holds that $in_{A'\Theta A'} \supseteq in_{A\Theta A}$.*

Proof. It holds that Lemma 8.13 applies after the replacement, such that $in_{A'\Theta A'} = in^{\diamond}_{A'}$. As it furthermore holds that $in_{A\Theta A} \subseteq in^{\diamond}_A$ it follows with $in^{\diamond}_{A'} \supseteq in^{\diamond}_A$ that $in_{A'\Theta A'} \supseteq in_{A\Theta A}$. □

These lemmas allow us to establish the replaceability of components in entire graphs as follows:

Theorem 8.23 (Replaceability in Graphs). *Let component A be part of a component graph G that consists of input-independent and empty-continuous components. Then component A can be replaced by a component A' according to Definition 8.19 if A' adheres to all requirements imposed by Lemmas 8.20 to 8.22 with respect to itself and the components that A' is composed with.*

Proof. Follows immediately from Lemmas 8.20 to 8.22 and the fact that graphs consist of parallel, serial and feedback compositions. □

This theorem defines conditions under which component replacement is allowed if knowledge about the output sets of predecessors and the input sets of successors is given. However, this knowledge may not always be available. For that purpose we additionally introduce the concept of independent replaceability as follows:

Theorem 8.24 (Independent Replaceability). *Let component A be part of a strictly input acceptance preserving component graph G. Then A can be replaced by a component A' according to Definition 8.19 if A' is input-independent, empty-continuous, has the same input and output interfaces as A and if both $in_{A'} \supseteq in_A$ and $out_{A'} \subseteq out_A$.*

Proof. In this proof in^*_A, out^*_A, $in^*_{A'}$ and $out^*_{A'}$ always refer to the respective discussed compositions. As G is strictly input acceptance preserving it holds for all components B serially preceding A, i.e. $B \in pred(A)$, that $in^*_A \supseteq C^\Theta(out^*_B)$ and for all components D serially succeeding A, i.e. $D \in succ(A)$, that $C^\Theta(out^*_A) \subseteq in^*_D$. Moreover, if A is feedback-composed, i.e. $A\Theta A$, it further holds that $C^\Theta(out^*_A) \subseteq in^*_A$. With $in_{A'} \supseteq in_A$ and $out_{A'} \subseteq out_A$ it immediately follows that $\forall_{B \in pred(A)} : \ in^*_{A'} \supseteq C^\Theta(out^*_B)$, that $\forall_{D \in succ(A)} : \ C^\Theta(out^*_{A'}) \subseteq in^*_D$, as well as that for feedback $C^\Theta(out^*_{A'}) \subseteq in^*_{A'}$ and that $in^\diamond_{A'} \supseteq in^\diamond_A$. Because all components of the graph after replacement are additionally both input-independent and empty-continuous it follows that Theorem 8.23 applies. □

9 TIMED DATAFLOW MODELS

Timed dataflow models are useful as they allow to express both the temporal and functional behavior of streaming applications and as their simpler variants can be analyzed efficiently. In the following we first introduce different variants of timed dataflow models and subsequently discuss their expression in our timed component model.

Definition 9.1 (HSDF). A Homogeneous Synchronous Dataflow (HSDF) graph is a directed graph $G = (V, E, \delta, \rho)$ that consists of a set of actors V and a set of directed edges E connecting these actors. An actor $v_k \in V$ communicates with other actors by producing tokens on and consuming tokens from edges, which represent unbounded queues. An edge $e_{kl} = (v_k, v_l) \in E$ initially contains $\delta(e_{kl})$ tokens. An actor v_k is enabled to fire iff at least one token is available on each of its incoming edges. Furthermore, the firing duration ρ_k specifies the difference between the start and finish times of a firing of an actor v_k. An actor consumes one token from each of its incoming edges at the start of a firing and produces one token on each of its outgoing edges when a firing finishes.

Definition 9.2 (SDF). An SDF graph is a generalization of an HSDF graph such that rate conversions are supported, i.e. an actor v_k consumes γ_{jk} tokens per firing instead of one from its incoming edges e_{jk} and produces π_{kl} tokens per firing instead of one on its outgoing edges e_{kl}.

Definition 9.3 (CSDF). A Cyclo-Static Dataflow (CSDF) graph is a generalization of an SDF graph such that cyclo-static actor phases are supported, i.e. an actor v_k consists of Θ instead of one phases, with each phase θ with $0 \leq \theta < \Theta$ having phase-specific firing durations ρ^θ_k, phase-specific

consumption rates γ_{jk}^{θ} for the incoming edges e_{jk} and phase-specific production rates π_{kl}^{θ} for the outgoing edges e_{kl}.

Besides the standard dataflow models HSDF, SDF and CSDF in which token consumptions and production occur in timestamp-order we also consider dataflow models in which tokens are consumed and produced in index-order. Note that the consideration of index-order enables auto-concurrency of dataflow actors, as indices allow to maintain the correct order of values even if the values are not ordered in time. We define dataflow models with support for auto-concurrency as follows:

Definition 9.4 (Auto-concurrent Dataflow). Dataflow models with support for auto-concurrency (HSDF$^{\text{a}}$, SDF$^{\text{a}}$ and CSDF$^{\text{a}}$) are a generalization of standard dataflow models (HSDF, SDF and CSDF) such that tokens are associated with indices and firings conducted in index- instead of timestamp-order.

As we show in the following we can express the different types of dataflow models quite simply in our timed component model. By defining both actor and token components we can express arbitrary dataflow graphs. We begin with the token component that is the same for all kinds of dataflow graphs:

Corollary 9.5 (Token Component). *The temporal and functional behavior of δ initial tokens on a dataflow edge e can be captured by a component T with a single input port p, a single output port q and the relation R_T as follows:*

$$\begin{aligned} R_T = \{(x,y) \in St(p) \times St(q) \mid x = x^{\infty} \;\Rightarrow\; & y = y^{\infty} \wedge \\ x \neq x^{\infty} \;\Rightarrow\; & \forall_{i<\delta}\colon\ \tau_y(i) = 0, \vartheta_y(i) = \vartheta_i^{init} \wedge \\ & \forall_{i\geq\delta}\colon\ \tau_y(i) = \tau_x(i-\delta), \vartheta_y(i) = \vartheta_x(i-\delta)\} \end{aligned}$$

Note that the first line is needed because components are by definition required to produce empty output traces for empty input traces.

In the definition of actor components we use the index i to differ between different firings of actors. With $i^p(i)$ and $i^q(i)$ we indicate the corresponding indices in the respective input and output streams. Note that for SDF and CSDF $i^p(i)$ and $i^q(i)$ return multiple indices in the form of sets. We use the application of the indexed timestamp and value functions on such sets as a shorthand notation to indicate that the respective functions apply to all indices in the sets.

Corollary 9.6 (Standard Actor Component). *The temporal and functional behavior of a standard dataflow actor v can be captured by a component V with an input interface P corresponding to the incoming edges of the actor, an output interface Q corresponding to the outgoing edges of the actor and the relation R_V as follows:*

$$\begin{aligned} R_V = \{(X,Y) \in Tr(P) \times Tr(Q) \mid \forall_{p\in P}\colon\ & \forall_i\colon\ \tau_{X[p]}(i) \leq \tau_{X[p]}(i+1) \wedge \\ & \forall_{q\in Q}\colon\ \forall_i\colon\ \tau_{Y[q]}(i^q(i)) = \max_{p\in P}(\tau_{X[q]}(i^p(i))) + \rho \wedge \\ & \vartheta_{Y[q]}(i^q(i)) = f_{p\in P}(\vartheta_{X[p]}(i^p(i)))\} \end{aligned}$$

For an HSDF actor component each firing consumes exactly one token from all input edges and produces exactly one token on all output edges. The stream indices accessed by a firing with index i are thus equal to that firing, i.e.:

$$\forall_{i\geq 0, p\in P, q\in Q}\colon\ i^p(i) = i^q(i) = i$$

An SDF actor component accesses per firing γ_p stream indices on each input edge $p \in P$ and π_p stream indices on each output edge $q \in Q$. This results in:

$$\forall_{i \geq 0}\colon\ \forall_{p \in P}\colon\ i^p(i) = \{i \cdot \gamma_p + i' \mid 0 \leq i' < \gamma_p\} \wedge$$
$$\forall_{q \in Q}\colon\ i^q(i) = \{i \cdot \pi_q + i' \mid 0 \leq i' < \pi_q\}$$

For a CSDF actor component the stream indices accessed per firing vary according to the cyclo-static phases of the actor. Thus we obtain with the phase corresponding to a firing with index i being $\theta(i) = i\%\Theta$ (with % the modulus operator):

$$\forall_{i \geq 0}\colon\ \forall_{p \in P}\colon\ i^p(i) = \{\sum_{i^*=0}^{i-1} \gamma_p^{\theta(i^*)} + i' \mid 0 \leq i' < \gamma_p^{\theta(i)}\} \wedge$$
$$\forall_{q \in Q}\colon\ i^q(i) = \{\sum_{i^*=0}^{i-1} \pi_q^{\theta(i^*)} + i' \mid 0 \leq i' < \pi_q^{\theta(i)}\}$$

The first line in the definition of R_V ensures that the component only accepts and produces in-order traces, i.e. that timestamps of traces are monotonically increasing in their indices. This is needed as standard dataflow actors do not have a notion of indices. The second line models the temporal behavior of a dataflow actor. The firing duration ρ is thereby intentionally defined ambiguously. It can be a constant, a range, a function, relation or adhere to a probabilistic distribution or even a finite state machine. This allows to express a wide range of applications using dataflow models, as well as to apply efficient dataflow analysis techniques on the simpler variants with e.g. constant firing durations. Our abstraction-refinement theory just allows to seamlessly bridge this gap between expressibility and analyzability by using various abstraction layers. The third line finally models the functional behavior, assuming that f is a function or relation that determines the values for all output trace indices corresponding to a firing i dependent on the values of all input trace indices that correspond to the same firing i. Note that this effectively makes dataflow actors stateless (actors with state must be modeled explicitly by using self-edges with at least one token).

For a dataflow actor with support for auto-concurrency we obtain analogously:

Corollary 9.7 (Auto-concurrent Actor Component). *The temporal and functional behavior of a dataflow actor v with support for auto-concurrency can be captured by a component V with an input interface P corresponding to the incoming edges of the actor, an output interface Q corresponding to the outgoing edges of the actor, the indexing functions $i^p(i)$ and $i^q(i)$ being the same as for a standard actor component and the relation R_V as follows:*

$$R_V = \{(X, Y) \in Tr(P) \times Tr(Q) \mid \forall_{q \in Q}\colon\ \forall_i\colon\ \tau_{Y[q]}(i^q(i)) = \max_{p \in P}(\tau_{X[q]}(i^p(i))) + \rho \wedge$$
$$\vartheta_{Y[q]}(i^q(i)) = f_{p \in P}(\vartheta_{X[p]}(i^p(i)))\}$$

Finally note that the aforementioned dataflow models are just presented here as an example of applicability. In a similar fashion also more complex dataflow models like Variable-Rate Dataflow (VRDF), as well as other models of computation like Kahn Process Networks (KPNs) or Timed Petri Nets can be expressed in our timed component model, allowing to construct combinations, as well as abstractions and refinements between different types of models.

10 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 4,

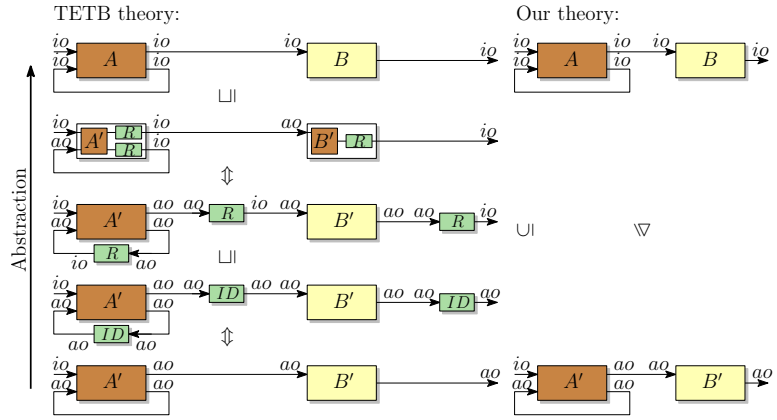

Fig. 4. In-order abstraction of an any-order implementation (*io*: in-order, *ao*: any-order).

our focus is in the following on other implications of separating bounding and input acceptance preservation.

10.1 Reordering

Consider the example depicted in Figure 4. 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 [8] 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 [10], which introduces indices to allow the expression of reordering and which does not require input-completeness for preservation of refinement on 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 4: 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.

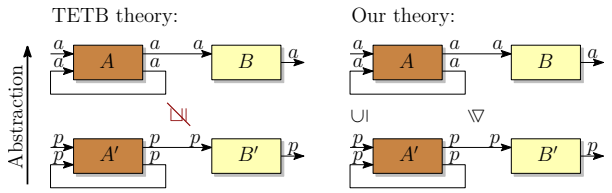

Fig. 5. Abstraction of periodic components A' and B' to input-complete components A and B (p: strictly periodic streams with period P, a: any streams).

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.

10.2 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 5. 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 [8, 10] 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. 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 5. 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.

10.3 Value Bounding

Another novelty in our abstraction-refinement theory is the introduction of the value ordering relation $\models_p$ on a per-port basis. This allows for the introduction of value abstractions, similar to the abstractions in the abstract interpretation theory [2], as well as value refinement.

Consider the left graph depicted in Figure 6. The two components A'' and B'' describe implementations of two parts of a program that both operate on streams of signed integer values. Our goal is

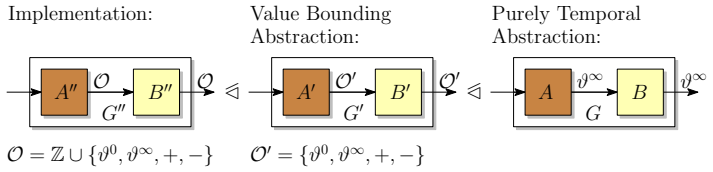

Fig. 6. Example for value bounding.

Input value	-6	-5	-4	-3	-2	-1	0	1	2	3	4	5	6
$L_{A''}$/µs	3.7	3.2	3.4	3.6	3.8	3.5	1.1	2.4	2.1	2.4	2.2	2.3	2.1
$L_{B''}$/µs	6.4	6.0	6.3	6.2	5.9	6.3	8.0	4.2	4.5	4.0	4.4	4.5	4.1

Table 1. Exemplary end-to-end latencies of components A'' and B'' in Figure 6 for different inputs.

to determine a conservative upper bound on the end-to-end latency of the graph G'' containing these components, i.e. to determine how long it maximally takes that an integer input value of component A'' results in an integer output value of component B''. For that matter, assume that each of the components has been analyzed in separation, such that, depending on the branches taken in the respective sub-programs, different component latencies have been determined for all integers. Some of these values are depicted in Table 1.

To determine the end-to-end latency of the whole graph, one could now test all combinations of input-output values of both A'' and B'', which would result in the highest accuracy. But already for this rather simple example of only two components, the number of combinations to test is pretty large. In a graph that would contain more components or even feedback compositions, the number of states that would have to be taken into account would inevitably explode.

Alternatively, one could simply take the maximum end-to-end latencies of both A'' and B'' to create purely temporal abstraction of the graph G'', using components A and B as depicted in the right graph in Figure 6. Suppose that it would have been determined that the maximum end-to-end latency of A'' for any integer input were 4µs and of B'' 8µs. Then the relations of A and B could be defined as:

$$R_A = \{(x, y) \in St(p_A) \times St(q_A) \mid \forall_i \colon \tau_y(i) = \tau_x(i) + 4.0\mu s \ \wedge \ \vartheta_y(i) = \vartheta^\infty\}$$
$$R_B = \{(x, y) \in St(p_B) \times St(q_B) \mid \forall_i \colon \tau_y(i) = \tau_x(i) + 8.0\mu s \ \wedge \ \vartheta_y(i) = \vartheta^\infty\}$$

It can be easily seen that A and B are valid worst-case bounds of A'' and B'', i.e. $A \trianglerighteq A''$ and $B \trianglerighteq B''$, and as A and B are input-complete also $in_G \supseteq in_{G''}$. This means that the graph G is a valid worst-case abstraction of G'' and the determined end-to-end latency of 12µs for this graph would be a valid upper bound on the end-to-end latency of G''. However, albeit easy to determine, this end-to-end latency appears rather conservative and one could ask whether there is not something in between maximum accuracy at highest complexity and minimum accuracy at lowest complexity.

With our theory, there is. Consider that it has been determined, based on integer-specific latencies such as the ones in Table 1, that component A'' does not take more than 4µs to produce output values if its input values are negative, no more than 1.1µs on an input of 0 and no more than 2.5µs on positive inputs. Likewise, consider that B'' does not take more than 6.5µs on negative inputs, no more than 8µs on an input of 0 and no more than 4.5µs on positive inputs. Lastly, assume that A'' always produces positive outputs on negative inputs, an output of 0 for an input of 0 and negative outputs on positive inputs.

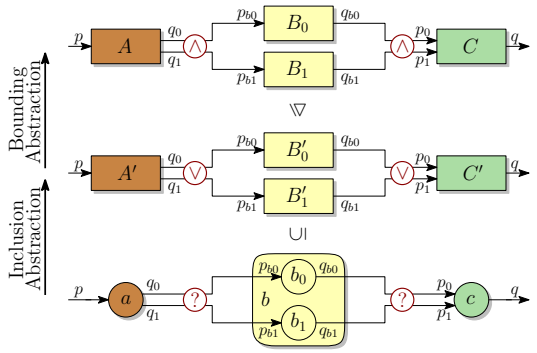

Fig. 7. Example for removal of uncertainty & non-determinism (?: uncertainty, ∨: non-deterministic "or", ∧: deterministic "and").

Given these observations, we can construct an abstraction of G'' that is both temporal and functional as follows. First, we define an extended value domain for all ports except the input port p_A, that does not only include all signed integer values, i.e. all values in $\mathbb{Z}$, and the infima and suprema ϑ^0 and ϑ^∞, but also the elements + and −, i.e. $\mathcal{O} = \mathbb{Z} \cup \{\vartheta^0, \vartheta^\infty, +, -\}$. For this value domain we can then define a value relation $\models$, such that $\forall_{\vartheta \in \mathcal{O}}\colon\ \vartheta \models \vartheta \wedge \vartheta^0 \models \vartheta \models \vartheta^\infty$, as well as $\forall_{\vartheta \in \mathbb{Z}^-}\colon\ \vartheta \models -$ and $\forall_{\vartheta \in \mathbb{Z}^+}\colon\ \vartheta \models +$. Based on this, we can define the components A' and B', that are depicted in the middle of Figure 6, via the following relations:

$$
\begin{aligned}
R_{A'} = \{(x,y) \in St(p_A) \times St(q_A) \mid\ & x = x^\infty \Rightarrow y = y^\infty \ \wedge\ x \neq x^\infty \Rightarrow \\
& \forall_i\colon\ \vartheta_x(i) < 0 \Rightarrow \tau_y(i) = \tau_x(i) + 4.0\mu s \ \wedge \vartheta_y(i) = + \ \wedge \\
& \qquad \vartheta_x(i) = 0 \Rightarrow \tau_y(i) = \tau_x(i) + 1.1\mu s \ \wedge \vartheta_y(i) = 0 \ \wedge \\
& \qquad \vartheta_x(i) > 0 \Rightarrow \tau_y(i) = \tau_x(i) + 2.5\mu s \ \wedge \vartheta_y(i) = -\} \\
R_{B'} = \{(x,y) \in St(p_A) \times St(q_A) \mid\ & x = x^\infty \Rightarrow y = y^\infty \ \wedge\ x \neq x^\infty \Rightarrow \\
& \forall_i\colon\ \vartheta_x(i) = - \Rightarrow \tau_y(i) = \tau_x(i) + 6.5\mu s \ \wedge \vartheta_y(i) = \vartheta^\infty \ \wedge \\
& \qquad \vartheta_x(i) = 0 \Rightarrow \tau_y(i) = \tau_x(i) + 8.0\mu s \ \wedge \vartheta_y(i) = \vartheta^\infty \ \wedge \\
& \qquad \vartheta_x(i) = + \Rightarrow \tau_y(i) = \tau_x(i) + 4.5\mu s \ \wedge \vartheta_y(i) = \vartheta^\infty\}
\end{aligned}
$$

Using our previously defined relation $\models$ it can be seen that A' bounds A'', but is not as conservative as A, i.e. $A'' \trianglelefteq A' \trianglelefteq A$, as well as analogously $B'' \trianglelefteq B' \trianglelefteq B$. Moreover, both G' and G'' accept any signed integer inputs, while G is input-complete, i.e. $in_G \supseteq in_{G'} \supseteq in_{G''}$. Consequently, G' is a valid worst-case abstraction of G'' and G is a valid worst-case abstraction of G'.

For the graph G' we can quite easily determine a maximum end-to-end latency by only considering three cases, namely that A' has a negative input (latency of $4 + 4.5 = 8.5\mu s$), that it has an input of 0 (latency of $1.1 + 8.0 = 10.1\mu s$) and that it has a positive input (latency of $2.5 + 6.5 = 9.0\mu s$). This gives us a maximum end-to-end latency of $10.1\mu s$ which is a valid upper bound on the end-to-end latency of G'', as G' is a valid worst-case abstraction of G'', but is less conservative than the end-to-end latency of $12\mu s$ determined with G.

10.4 Removing Uncertainty & Non-Determinism

We call a component non-deterministic if it has input traces for which it cannot produce only one, but multiple output traces. Non-determinism is usually introduced due to several uncertainties that are prevalent in reality. For example, it is practically impossible to determine the actual execution time of a task that is executed on a real processor. Several uncertainties like non-deterministic decisions, caching, memory port sharing, bus contention, processor sharing and even processor

clock jitter make it extremely challenging to find the actual, accurate execution time of a task. Another example is the uncertainty with respect to the access order of shared variables between different tasks on different processors. And finally, there can be even multiple instances of the same task running on multiple resources, for which the exact moments at which resources become available cannot be determined, resulting in a possible reordering of data.

All such uncertainties can be addressed by inclusion abstraction, as inclusion abstraction allows to replace any kind of uncertainty by non-determinism. For instance, if the exact execution time is not known, the respective inclusion-abstract component can produce non-deterministic outputs within a range between best-case and worst-case execution times. Likewise, an unknown order of memory accesses, as well as an unknown availability of resources can be abstracted away by considering all cases non-deterministically.

However, the big problem with non-deterministic models is that they usually cannot be analyzed efficiently. For that matter, it is regularly required to remove non-determinism from a model. In general, there are two ways to remove non-determinism: Composition and bounding abstraction.

In some particular cases, it is possible to remove non-determinism by only composing components. For example, consider two serially connected components of which the first produces non-deterministic outputs for deterministic inputs and the second produces deterministic outputs for non-deterministic inputs. Obviously, the serial composition of the two produces deterministic outputs for deterministic inputs, despite some internal non-determinism. This means that in this case non-determinism can be removed by composition alone. However, such use-cases are rare. In the following we consequently focus on removal of non-determinism by bounding abstraction, a concept that is applicable on any use-case involving non-determinism.

For instance, assume the following scenario depicted at the bottom of Figure 7: A task a is executed on a single processor and produces outputs for a task b. Task b is executed in two instances, on two different processors. Task a always sends the next data item to the instance of b that is first available. Task c then gets the outputs of both instances of task b as inputs and always consumes the input of the instance that is first finished.

The problem with this setup is that the availability and finish times of both instances of b are uncertain at design time. Consequently, it is uncertain to which instance of b the task a sends data, as well as from which instance of b the task c receives data. As discussed above, such uncertainties can be removed by inclusion abstraction. Let A' include task a, let B'_0 and B'_1 include the respective two instances of task b and let C' include task c. The uncertainty of a can be removed by introducing non-determinism in A', such that A' can at any moment either produce an output on port q_0 or produce an output on port q_1. Likewise, the uncertainty of c can be removed by introducing non-determinism in C', such that C' can at any moment either accept an input from port p_0 or accept an input from port p_1.

In the following, let us assume for simplicity that the differences between the arrival times on the inputs of both A' and C' are large enough compared to the execution times of the underlying tasks, such that iterations of these tasks cannot be delayed by preceding iterations. Moreover, let us only focus on the temporal aspects of the different relations, by ignoring any associated values. And finally, let us assume that if A produces an output for index i on p_0, it produces an output with timestamp 0 on p_1, and vice versa. Thereby we define an output with timestamp 0 as "no output", which is to ensure that corresponding indices can be maintained between abstraction layers. Given this, we can express the relation of A' as follows:

$$R_{A'} = \{(X,Y) \in Tr(P_A) \times Tr(Q_A) \mid \forall_i \colon\ (k = 0 \vee k = 1) \wedge \\ \tau_{Y[q_k]}(i) = \tau_{X[p]}(i) + \rho_a^i \ \wedge\ \tau_{Y[q_{1-k}]}(i) = 0\}$$

The parameter k is thereby used to express the non-determinism of A': On $k = 0$ for a certain index i the respective output is produced on port q_0 and no output (timestamp of 0) on port q_1, whereas on $k = 1$ the output is produced on port q_1 and no output on q_0. We further define the relations of B'_n (with $n \in \{0, 1\}$) as follows:

$$R_{B'_n} = \{(x, y) \in St(p_{bn}) \times St(q_{bn}) \mid \forall_i \colon \tau_y(i) = \max_{i' < i}(\tau_x(i), \tau_y(i')) + \rho_b^i\}$$

These components are deterministic as their outputs are unambiguously defined by their inputs. The formulation with max takes care that an iteration of a task execution cannot start before the previous ones are finished. Lastly, we define the relation of component C':

$$R_{C'} = \{(X, Y) \in Tr(P_C) \times Tr(Q_C) \mid \forall_i \colon (k = 0 \vee k = 1) \ \wedge \ \tau_{Y[q]}(i) = \tau_{X[p_k]}(i) + \rho_c^i\}$$

Here, the parameter k models the non-determinism of C': On $k = 0$ for a certain index i the input on port p_0 is consumed by C', which is reflected by the dependence of the output timestamp $\tau_{Y[q]}(i)$ on the input timestamp $\tau_{X[p_0]}(i)$, whereas on $k = 1$ the input on port p_1 is consumed.

The graph G' is apparently non-deterministic because both A' and C' are non-deterministic. We can now try to remove this non-determinism using component worst-case bounding. For component A' we can derive a deterministic worst-case bound A if we replace the choice on which port outputs are produced with a production on both ports, i.e.:

$$R_A = \{(X, Y) \in Tr(P_A) \times Tr(Q_A) \mid \forall_i \colon \tau_{Y[q_0]}(i) = \tau_{Y[q_1]}(i) = \tau_{X[p]}(i) + \rho_a^i\}$$

It is easy to see that A is deterministic and a valid worst-case bound on A', i.e. $A \trianglerighteq A'$. The components B'_0 and B'_1 are already deterministic and it holds that $B'_0 \trianglerighteq B'_0$, as well as $B'_1 \trianglerighteq B'_1$. Consequently, we can leave these components unchanged. Lastly, we can bound component C' with a deterministic component C that does not use the arrival times of data on either of the two inputs, but the maximum of both, i.e.:

$$R_C = \{(X, Y) \in Tr(P_C) \times Tr(Q_C) \mid \forall_i \colon \tau_{Y[q]}(i) = \max(\tau_{X[p_0]}(i), \tau_{X[p_1]}(i)) + \rho_c^i\}$$

Also this component is apparently deterministic and a valid worst-case bound on C', i.e. $C \trianglerighteq C'$. Because the input acceptance of the components is not changed between layers (the components are all input-complete), we can finally assess that graph G is a deterministic and valid worst-case bounding abstraction of G'.

10.5 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 [8, 10] 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 8. 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

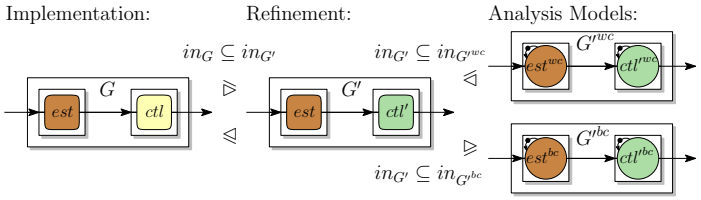

Fig. 8. Replacement of a slow input-complete controller implementation ctl by a faster controller ctl' with restricted input acceptance.

the tasks, their temporal behavior can be expressed using the following relations (with $\tau_y(-1) = 0$):

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

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:

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

We have to show that the graph G' in the middle of Figure 8 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 6.13 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 6.3, 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 HSDF actors, as depicted on the right side of Figure 8. Considering the semantics of HSDF actors [16], it follows with Corollaries 9.5 and 9.6 for the relation $R_{est^{wc}}$ (the self-edge token component is here already feedback-composed with the HSDF actor component):

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

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* [9, 12]. For any output stream y of *est* we can conclude from the best-case model that $\forall_{i<|y|}:\ i \cdot 10\mu s + c + 1\mu s \leq \tau_y(i)$ and from the worst-case model that $\forall_{i<|y|}:\ \tau_y(i) \leq i \cdot 10\mu s + c + 3\mu s$. From that we obtain:

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

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.

11 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.

On top of that, we extended the journal paper [11] in multiple ways. Besides several formalizations of formerly informal definitions and proofs, the most notable extension lies in the introduction of an inclusion abstraction-refinement theory for the same timed component model. Altogether, the combined theory enables analysis approaches based on both inclusion and bounding abstraction, as well as design processes based on both inclusion and bounding refinement. Moreover, we have formalized the expression of various timed dataflow models in our component model and included several additional case studies for an improved accessibility.

REFERENCES

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

[2] P. Cousot and R. Cousot. 1977. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. 238–252.

[3] 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.

[4] 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.

[6] 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.

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

[8] 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.

[9] 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.

[10] 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)*.

[11] P. Kurtin and M. Bekooij. 2017. An abstraction-refinement theory for the analysis and design of real-time systems. *ACM Transactions on Embedded Computing Systems (TECS)* 16, 5s (2017), 173:1–173:20.

[12] 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.

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

[14] X. Liu. 2005. *Semantic Foundation of the Tagged Signal Model.* Ph.D. Dissertation. University of California at Berkeley.

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

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

[17] L. Thiele, E. Wandeler, and N. Stoimenov. 2006. Real-time interfaces for composing real-time systems. In *ACM International Conference on Embedded Software (EMSOFT)*. 34–43.

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

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

[20] 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.

[21] 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.