The Journal of Systems and Software 210 (2024) 111940

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

Time, causality, and realizability: Engineering interactive, distributed software systems

Manfred Broy [#]

Institut für Informatik, Technische Universität München, D-80290 München Germany

ARTICLE INFO

Keywords:
Interaction
Distribution
Specification
Verification
Time
Interface

ABSTRACT

Today, software systems are distributed, concurrent, interactive, embedded, and often time-critical. They communicate and cooperate by data exchange. Introducing notions of time, causality, and realizability into interface-based data flow models results into a powerful model for engineering cyber-physical and, generally, distributed software systems. Architectures and their specification are designed by the concurrent composition of systems. This approach provides high expressive power supporting explicitly specification of interface behavior, encapsulation, abstraction, refinement, verification, concurrent composition, and modularity. The key idea is a formal, semantically coherent modelling, and specification framework that integrates the specification of time-critical and non-time-critical software embedded into cyber-physical systems. The approach supports the refinement of interface specifications of non-time-critical systems into interface specifications of time-critical systems and vice versa. This leads to an integrated specification and design approach for time-critical data flow architectures with soft, firm as well as hard time-critical requirements.

1. Introduction: Time, computation, and systems engineering

"Computation needs time" is the title of a paper (see (Lee, 2009)) published by Edward Lee more than 10 years ago. As he accurately points out, any form of computation needs time. Time is an inherent and sometimes critical aspect in computing and for computing systems. Historically, there was a kind of separation between, on one hand, scientific work on concurrent distributed interactive software systems, where most of the models were developed with the goal to avoid and abstract away explicit time and, on the other hand, practical work on embedded and communicating systems often dealing with time-criticality. Abstracting from time was initially motivated by the concept of portability with the idea that a piece of software should run on different hardware platforms while always showing the same behavior.

With the movement from host systems to client server hardware and further on to general distributed software on networks, with the paradigm of interaction, and finally the event of cyber-physical systems (see (Geisberger and Broy, 2012)) time requirements became more and more relevant. Nevertheless, in many formal methodological frameworks, quantitative time was not and is not explicitly included. Typically, time is handled technically at the level of operating systems or middleware. Sometimes, abstractions such as fairness are introduced that address specific aspects of the timing of systems in an abstract way without mentioning time explicitly. Even in temporal logics, where time is explicitly mentioned by the term "temporal", time is not explicitly considered except in specific versions of temporal logics.

Most of the work on time-critical software is more pragmatic and more oriented towards practical problems and technical applications, but less towards foundations. Exceptions are found, for instance, in the contributions of Tom Henzinger (see (Henzinger and Jagadeesan, 1997)) on timed automata or in his work on Giotto (see (Henzinger et al., 2003)) and the work of Gerard Berry (see (Berry, 2000)).

In the following, we suggest a novel view onto time in system computations with practical, methodological and foundational benefits. We introduce time into an interface centric system model as a general concept serving two purposes:

- The system model supports the specification of the behavior of time-critical systems.
- The system model is used as a construct to specify and reason formally about systems – in particular, also non-time-critical systems and their feedback loops in terms of causality and realizability.

E-mail address: broy@in.tum.de.
[#] http://www.broy.informatik.tu-muenchen.de

https://doi.org/10.1016/j.jss.2023.111940
Received 15 November 2021; Received in revised form 18 November 2023; Accepted 20 December 2023
Available online 27 December 2023

This way – from a theoretical perspective – a sound and relatively complete calculus for the specification, composition, refinement, and verification of both non-time-critical and time-critical systems is constructed (see (Broy, 2023)) and – from a practical perspective – a coherent modeling and development approach for the design of distributed interactive systems with time-critical behavior.

1.1. Time and data flow in system interfaces

In distributed interactive systems, messages and signals are exchanged. Each instance of a message or a signal exchange is called a *communication event*. Each communication event carries some message, some signal or data, and happens at a specific time. Strictly speaking, we may distinguish between the time a message is sent and the time it is received. The obvious fact that the answer to a message received as input is sent later than the time it was received, is a basic fact of physical reality captured by the principle of causality.

We include causality into the behavior of systems by the following principle. If a system receives a message at time t – and the time granularity is fine enough, then output caused or influenced by this input can be issued earliest at time $t + 1$. In the following, we associate only one time with a transmitted message, the time a message is sent. It is received at exactly that time since we neglect delay in transmission. However, if we want to model the time delay in transmission this can be done by special transmission components that receive a message at the time it was sent and forward it later with some delay. We may also model communication networks, the physical resources used for communications (e.g., circuits, memory buffers) which are almost invariably shared by multiple logical channels. This way, we specify, as in real-world systems, the timed data flow on individual channels.

In principle, with every communication event on a channel we associate two items, the data it carries and the time it is sent. We consider timed streams of such events. By such timed streams, both the stream of data, the data flow, and the time flow is represented. Systems receive messages via timed streams over their input channels, called their *input* streams, and send messages as elements of timed streams over their output channels, called their *output* streams. There is a causal relationship between input and output of systems both in terms of data and of time.

1.2. Modelling time

We work with a most basic simple model of time in the following where time is modelled by an infinite sequence of time intervals numbered by natural numbers. Such a notion of time is nevertheless sufficient for digital systems.

Clearly, we can distinguish between two types of systems:

- *time-critical* systems, where sharp time requirements are essentially part of their specifications and, moreover,
- *time-free* or *non-time-critical* systems, where the timing is not relevant in their application requirements nor for their data flow.

Nevertheless, also for non-time-critical systems we are interested in some kind of loose timing requirements, because every practical system is supposed to produce answers after a reasonable period of time, although it might not be necessary to specify all the details of their precise timing.

In the following, we suggest to use time for two purposes:

- For time-critical systems, we work with a system model with an explicit concept of time. We choose a very simple, straightforward discrete model of time, where we describe time by a sequence of time intervals of equal lengths and specify system behaviors relative to that timing. This is a global time synchronous model of computation, as introduced also by Berry (Berry, 2000). However, in order to deal with systems in which it is (practically) not possible to provide a global synchronization model, the "globally asynchronous, locally synchronous" paradigm can be introduced by specifying time shifts for modelling local time.
- A second way to make use of time is to introduce time in terms of the same system model but exploit time as an auxiliary construct to reason about systems in terms of causality – even about non-time-critical systems as shown in (Broy, 2023) and also demonstrated in the following.

The second way to make use of time can be explained and justified by the following observation. Operationally, every physical computation and reaction takes time. An explicitly given time frame allows us to derive additional properties about the behavior of computing systems – even in cases where time constraints are not included in the requirements specification of systems. Just assuming the time frame leads to useful rules and principles as expressed by causality and realizability.

Including time into the model of interactive distributed systems ends up with a framework of high expressive power for the specification and verification of systems. Moreover, using two semantically coherent models, side by side, a timed model and a non-timed one, we have the freedom to refer to time whenever useful and needed or methodological appropriate as well as – on the other hand – to go away and abstract from time, whenever time is not needed.

We show in the following how these complementary models of systems with time and without time form a very powerful basis for software and systems engineering, in particular, for specification and formal reasoning.

1.3. On the timing of systems

Actually, there are several aspects of the timing of systems. First of all, it is interesting how much time a computation requires. This is addressed in computational complexity theory (see (Leeuwen, 1990; Rich, 2008)) speaking about the orders of magnitude of computing steps a computation needs for solving a problem, in principle, depending of the magnitude of its parameters. Actually, these considerations are kept independent of the concrete physical execution time of machines. If machines get faster, just another factor is multiplied with the complexity measurement to estimate physical time.

When considering software systems that interact with the real world, we speak of cyber-physical systems. In this case, it is important to capture time-critical behavior. If systems control physical devices, then it is critical to know, what the timing of the controlled system is and how to react in time by the controller. This leads to the question whether the controller computes fast enough to be able to timely control the system behavior.

For time-critical applications, we find two different, relevant measures of time. First of all, there is the, the inherent time needed for the computation of a software system given a specific hardware. This inherent timing depends on the speed of the used hardware. A different notion is the *specified* time in terms of system requirements which describes the time in which the system is required to react. In fact, a system is able to react timely correctly in a time-critical context only, if the specified time requirement is not less than the execution time needed.

In timing requirements and also in time-critical behavior, we distinguish between two fundamental notions of timing system reactions.

- *Delay*: The time needed between a certain event happening and its observation by the system via some input, and the earliest time the system responds to the event. For some applications, a system has to respond with some specified delay. This leads to requirements about the earliest time a system must or can react.
- *Deadline*: A deadline describes the time span till when a system must respond to some input. Accordingly, deadlines are expressed

exclusively as relative to message reception. However, in many systems, deadlines are specified in an absolute manner (i.e. by reaction with respect to the time of day). In the latter case, we speak of *urgency*.

Timing concepts are crucial, when systems control physical devices. There not only the timing of the controller but also the timing of the system or process under control are relevant. Then the question is, what is the earliest and what is the latest time a process under control responds by a sensor signal to input received by some actuator. We may use the same system model and specification techniques for the controller and the system under control. Both show timed behavior. Take as an example the controller of an airbag. The need to activate the airbag is indicated by a signal of the crash sensor and then it should take about 200 msec until the airbag starts to inflate, but not much more or much less. Thus, there is a required delay as well as a required deadline for the controller.

1.4. The challenge

The challenge is to find a model and a theory for concurrent, interactive systems as a basis of a calculus for specifying and reasoning about abstract functional properties of embedded software systems and their architectures including data flow, time-critical properties (see (Broy, 2023, 2024) for an axiomatic description of the calculus) and, in a perspective, also probabilistic properties (see (Neubeck, 2012)). We concentrate in the following on logical interface properties including time-critical aspects. In particular, we show that – given basic interface properties by interface assertions – we derive from the specified properties additional properties using the concept of causality and realizability which address the timing in computations in terms of the causal dependencies between input and output. Whereas so far the concept of "causality" is probably clear, the meaning of "realizability" has not been explained at this point. Realizability basically expresses that a computation strategy does exist for computing correct results as specified. Assuming causality and realizability results in proof methods for systems and, in particular, for composed systems dealing with feedback and all kinds of concepts of interaction.

The basic idea is that by the calculus we derive interface properties, some of them by straightforward application of well-known rules of predicate logic, others by using the principles of strong causality or full realizability described by more specific rules. Therefore, we have the freedom, when composing systems, in the case of properties that can be derived without applying the axiom of strong causality and full realizability to apply straightforward reasoning. Properties which depend on causality considerations or even on realizability are proved by the principles of strong causality and full realizability. This way, they are used for the derivation of interface specifications of the composed systems from the enhanced interface specifications of their components. This way, architecture specification are calculated from the specifications of their subsystems.

1.5. Classes of time sensitive systems

When looking at time-critical systems, we distinguish three classes with respect to time sensitivity. In the simplest case, the data output, the data flow, of a system does not depend on the timing of the input, but only on the input data. Nevertheless, the timing of the output typically depends on the timing of the input.

Time has a more crucial role, if the requirements for the timing of the output are more specific. For instance, a system should not respond too early and guarantee some delays and it should not respond too late and therefore guarantee some deadlines. More specifically, this timing for the output may depend for certain applications on the specific input data.

For instance, if we consider the controller of a cruise control device, which is supposed to issue a signal at a certain point in time, then the delay and the deadline for the cruise control system may depend heavily on the speed of a vehicle which is given as input to the controller. This indicates, how sophisticated the role of timing can be with respect to the data and the corresponding physical events being processed.

In consequence, we distinguish between “hard”, “firm”, and “soft” time-criticality.

- A *hard* real-time system is a system with requirements about its deadlines or delays which are strict. If such a system does not reach its deadline its behavior is a failure and the system is called incorrect.
- For a *firm* real-time system, infrequent missing of deadlines is acceptable. To express “infrequent” formally, we may work with probabilities.
- A *soft* real-time system is a system where there are no strict delays nor deadlines but a notion of “quality of service”.

However, even for “soft” time-critical systems it is clear that, if a system shows a very long delay such that the response time may get unacceptable long, this may cause problems – even for applications which do not show any hard time-critical requirements. It is clear that, if we do a calculation for a large problem (say data analytics) it makes a lot of difference whether the computation takes a few microseconds, a few minutes, a few hours, a few days, or even months.

Timing properties are often globally called *non-functional*. This is an unfortunate and strictly speaking inadequate characterization, because the classification of timing properties as functional or as nonfunctional depends very much on the nature of the requirements, whether timing is function critical. We show in this paper specifications of time critical functions. There, we suggest to speak of timing properties being *functional* properties as pointed out in (Broy, 2015). This certainly applies to hard time-critical systems. As pointed out, for firm and soft time-critical systems, we may talk about time related to probabilities. In the following, we concentrate on time and nondeterminism but do not consider probabilities. Nevertheless, the introduced models including time can be generalized to probabilistic models with time (see (Neubeck, 2012)).

Next, we describe how we can work with an integrated model of data flow and time flow, and also demonstrate, how – by abstraction – we can get rid of time information just studying the data flow. We discuss the effects of abstracting from time for hard time-critical systems. The theory of such a model is explained in detail in (Broy, 2023). In the following, we rather demonstrate the engineering advantages of such an approach. In particular, we discuss timing properties in requirements and timing properties of implementations and show how to use and verify timing properties of implementations.

1.6. Engineering distributed interactive systems – related work

The research done over the last fifty years to find adequate formal models of concurrent distributed interactive systems (see, for instance (Milner, 1980; Hoare, 1985)) did not result so far in sufficiently tractable models for software system specification, design, and implementation. We introduce a model for interface behavior of concurrent distributed interactive systems based on time, causality, and realizability. We base this approach on a model that has been developed over the last 20 years (see Focus (Broy and Stølen, 2001), (Broy, 2010)). We define a logical calculus (see (Broy, 2024)) for reasoning about the interface behavior of system architectures in terms of their interface behaviors and therefore about the most important properties when dealing with such types of systems.

This is in contrast to many of the concurrent programming paradigms based on von Neumann architectures with shared memory where concurrency is inherently leading to nondeterminism due to the nondeterministic interleaving of concurrent access to shared resources such as shared memory. This leads to complex programming concepts,

with difficulties for specification and verification. Our approach is based on fully separated, encapsulated entities of systems, called subsystems, for explicit concurrent execution which interact by message passing.

Distributed interactive concurrent software systems and their formal modelling are an area of development and research in software and systems engineering since the 60ies. Beginning with the development of operating systems, concurrent processes and their synchronisation were studied. E.W. Dijkstra introduced the concept of semaphores (Dijkstra, 1963). Approaches like Dekker's algorithms (Dijkstra, 1968) provided solutions to the shared concurrent access to resources. Other approaches also oriented towards shared state are found in the work of Owicki and Gries (Owicki and Gries, 1976) on assertion logics for concurrent programs with shared variables. More abstract work was done by Chandy and Misra on Unity (Chandy and Misra, 1988), by Gurewich on abstract state machines (Gurevich, 2000), by de Alfaro and Henzinger on interface automata (de Alfaro and Th., 2001) (see also (Tripakis et al., 2011)), and by Lamport on TLA (Lamport, 1994) and by Abadi and Lamport on composing specifications (Abadi and Lamport, 1993).

In the 70ies proposals like CCS by Robin Milner (Milner, 1980) and CSP by Tony Hoare (Hoare, 1985) aimed at the construction of formal models for concurrent interacting systems with synchronous communication. However, the semantic treatment of these models proved to be quite sophisticated leading in the case of CSP to complex denotational models based on the idea of refusal and readiness sets and in the case of CCS to operational semantics and bisimulation equivalence, which do not provide an explicit basis for system specification. Further developments such as the chemical abstract machine by Berry and Boudól (Berry and Boudol, 1992), ambient calculus by Luca Cardelli (Cardelli and Gordon, 1998), and π-calculus by Robin Milner (Milner, 1999) aimed at specific aspects of concurrent systems.

Another line of work followed the idea of data flow. Early work of Jack Dennis (J., 1974) was influenced by the work of Carl Adam Petri on Petri nets (Petri, 1962), which introduced a first very basic model of concurrent computation. Gilles Kahn, published two fundamental papers, the second one together with David MacQueen, on deterministic data flow (Kahn, 1974; Kahn and MacQueen, 1977). David Park suggested concurrency through automata on infinite sequences (Park, 1981). Jaco de Bakker published work on metric spaces (de Bakker and van Breugel, 2000). Ideas of event and data flow are found in the work of Gerard Berry on Esterel (Berry, 2000) and in the work of Jay Misra on Orc (Misra, 2005).

We base our approach fundamentally on data flow concepts, but add a model of discrete time to it which is extending the model of David Park (Park, 1981). Furthermore, we assume strong causality which is related to the guardedness condition of Jaco deBakker and leads to contractive mappings as introduced by Banach (Banach, 1922).

1.7. Content of the paper

An approach to the specification, design, and refinement of cyber-physical systems is given with emphasis on interfaces and modularization. The approach is strongly based on the theory described in (Broy, 2023) where the fundamental models and their theory are explained in detail. There the basic concepts are introduced and justified from a theoretical perspective. This includes the concept of a model for the interface behavior of systems, interface assertions to specify interface behavior, refinement, and concurrent composition. It is shown that the logical calculus is sound and relatively complete with respect to an implementation concept based on generalized Moore machines – Moore machines which work on unbounded and infinite state spaces.

In the following, we demonstrate, how this approach can be applied for specifying systems with time-critical properties in a way closer to practical needs. First, in Section 2, we introduce the basic concepts such as timed streams, interface behavior of systems, and its specification by interface assertions. In Section 3, we introduce the notation for writing interface specifications by templates, define the key concepts for interface specifications such as refinement, strong causality and full realizability as well as concurrent composition with feedback. All concepts are explained and motivated by small examples and it is shown how properties are derived and proved for interface specifications. Also, the specification of system architectures is introduced and explained by the examples. It is shown, how timing properties are needed and used in the design of distributed systems. We do not give a general calculus and extensive instances for the verification of such systems, but demonstrate the approach by a few brief examples. In (Broy, 2024) an axiomatic proof calculus is introduced and a number of verification examples are given. Finally, in Section 4, the relationship of the approach to system engineering is discussed with emphasis on methodological essentials such as modularity, concurrent composition, system architecture and expressive power. Some concluding remarks are formulated at the end.

2. On systems, timed interfaces and their properties

Throughout this paper, we use the term "system" in a specific way. We address discrete systems, more precisely, models of discrete, concurrent, time-critical systems with input and output. For us, a system is an entity that shows some specific behavior by interacting over its interface with its *operational context*. A system has a *boundary*, which defines what is inside and what is outside the system. Inside the system there is an encapsulated internal structure, often consisting of states or state attributes and an architecture composed of subsystems. This structure is hidden following the principle of *information hiding*. The set of actions and events that may occur in the interaction of the system with its operational context determines the *syntactic ("static") interface* of the system. At its interface, a system shows some specific *interface behavior*. We specify interface behavior by *interface assertions*.

There are general logical properties that we assume for interface assertions. We require that system behaviors fulfill properties such as *strong causality* and *realizability* which hold for implementations. Causality requires that output is only generated after the corresponding input has been received. Realizability requires the existence of a strategy in the interaction between the system and its environment that leads for arbitrary input to results that are correct according to the specification (for details, see (Broy, 2015)). However, not all interface specifications guarantee these properties. Nevertheless, in such cases, causality as well as realizability can be added schematically as steps of refinements to system specifications resulting in additional properties.

Components of systems being distributed run in parallel and systems and, in general, operate in parallel, evolving in a common physical time frame. They communicate and interact in order to be active parts of distributed systems connected by communication networks. For modeling this, an interface concept has to be chosen appropriately. We aim at an interface concept to describe services in a property-oriented way, services that are interactive, time-critical, and run in parallel.

2.1. Streams, time, and behaviors

In this section, we introduce two models of concurrent, distributed interactive systems, one without time and one including time.

By M^* we denote the set of finite sequences over set M which formally can be represented by functions ($\mathbb{N}$ denotes the natural numbers including 0, $\mathbb{N}_+ = \mathbb{N}\backslash\{0\}$; $\mathbb{N}_+$ denotes the set of natural numbers without 0; $[1{:}n] \subseteq \mathbb{N}_+$denotes finite intervals of natural numbers for $n \in \mathbb{N}$)

$M^* = \cup_{n \in \mathbb{N}} ([1{:}n] \to M)$

By M^ω we denote the infinite sequences over M represented by functions:

$M^\omega = (\mathbb{N}_+ \to M)$

Based on these definitions we denote the set of *untimed streams* including finite ("partial") as well as infinite streams by

$M^{*|\omega} = (M^* \cup M^\omega)$

The length of a stream $x \in M^{*|\omega}$ is denoted by $|x|$ which is in $\mathbb{N}\cup\{\infty\}$.

The empty sequence as well as the empty stream is denoted by $\langle\rangle$. For $m \in M$, we denote the one element sequence and the one element time free stream by

$\langle m \rangle$

For $m_1, m_2, m_3, \ldots, m_n \in M$, we denote by

$\langle m_1\, m_2\, m_3 \ldots m_n \rangle$ the finite time free stream (and also the sequence) x of length n with $x(k) = m_k$ for all k with $1 \leq k \leq n$. For $m_1, m_2, m_3, \ldots \in M$, we denote by

$\langle m_1\, m_2\, m_3 \ldots \rangle$ the infinite time free stream x with $x(k) = m_k$ for all k with $k \in \mathbb{N}_+$. If $m_k = b$ for all $k \in \mathbb{N}_+$we write b^∞ for this infinite stream.

We denote the set by of infinite *timed streams* over a message set M by

$(M^*)^\omega = (\mathbb{N}_+ \to M^*)$

In both cases, we use functions on the set of natural numbers or intervals thereof to define timed streams as well as time free streams. A stream $x \in (M^*)^\omega$ is called *timed*, since we understand $\mathbb{N}_+$ to represent time in terms of an infinite sequence of time intervals. Each time interval is numbered by a number $t \in \mathbb{N}_+$. Then x(t) with $t \in \mathbb{N}_+$ denotes the sequence of messages communicated by the timed stream x in time interval t. We do not give explicit physical values for the length of these time intervals

Note that the role of natural numbers in $\mathbb{N}_+$ in the two expressions $(\mathbb{N}_+ \to M)$ and $(\mathbb{N}_+ \to M^*)$ is quite different. In case of timed streams x, the number $n \in \mathbb{N}_+$ denotes the sequence x(n) transmitted in the nth time interval, whereas in time free streams $z \in M^{*|\omega}$ the number $n \in \mathbb{N}_+$ simply denotes the position of a message z(n) in a stream and does not represent time.

The difference between $M^{*|\omega}$ and $(M^*)^\omega$ is seen by the different ways we denote the *empty* stream, the stream without messages. For $M^{*|\omega}$ the empty stream is denoted by $\langle\rangle$ – the empty sequence, for $(M^*)^\omega$ the empty stream, the stream without messages is denoted by $e = \langle \langle\rangle \langle\rangle \ldots \rangle$ where $e(t) = \langle\rangle$ for all time intervals $t \in \mathbb{N}_+$.

By $\#x \in \mathbb{N} \cup \{\infty\}$ we denote the number of elements from M occurring in a timed stream $x \in (M^*)^\omega$ or in a time free stream $x \in M^{*|\omega}$. For a timed stream $x \in (M^*)^\omega$ or a time free stream $x \in M^{*|\omega}$ over set M and a subset $D \subseteq M$ we denote by

D#x the number of copies of elements from D in x; we also write a#x for {a}#x and #x for M#x.

Timed streams can be used as representations for time free streams. Every timed stream $x \in (M^*)^\omega$ can be mapped onto a time free stream $\overline{x} \in M^\omega$ defined as follows: $\overline{x}$: {1, …, #x} → M if $\#x \in \mathbb{N}$ $\overline{x}$: $\mathbb{N}_+ \to M$ if $\#x = \infty$ where $\overline{x}$ denotes the time free stream that is the result of the concatenation of all the sequences $x(t),\ t \in \mathbb{N}\backslash\{0\}$. In the case of time free streams $z \in M^{*|\omega}$, we denote the n-th element in the streams by z(n). By x[k] we denote the k-th message in the timed stream $x \in (M^*)^\omega$ provided $0 < k \leq \#x$. We simply define the kth message x[k] in the timed stream x by $x[k] = \overline{x}(k)$ provided $\#x \geq k$

As well-known we may introduce a partial order $\sqsubseteq$ called *prefix order* on the set of time free streams $M^{*|\omega}$; for a, $b \in M^{*|\omega}$ we define the prefix order as follows $a \sqsubseteq b \Leftrightarrow |a| \leq |b| \wedge \forall k \in \mathbb{N}: 1 \leq k \leq |a| \Rightarrow a(k) = b(k)$

By this definition, $(M^{*|\omega}, \sqsubseteq)$ is a complete partial ordered set (cpo) with $\langle\rangle \in M^{*|\omega}$ where $\#\langle\rangle = 0$, as its least element. This applies as well to timed streams: $((M^*)^{*|\omega}, \sqsubseteq)$ is a cpo.

Given a sequence s of elements from M (or in the case of timed streams of elements from M*) and a stream x over M (finite or infinite, in the case of time streams of elements over M*) we denote by $s\hat{}x$ the concatenation of sequence s with stream x.

Given $x \in (M^*)^\omega$ and $t \in \mathbb{N}$ we denote a *time cut* of length n as follows $x{\downarrow}t \in ([1{:}n] \to M^*)$ where

$(x{\downarrow}t)(n) = x(n) \Leftarrow 1 \leq n \leq t$ x↓t denotes the first t sequences in the timed stream x, representing the messages transmitted in the first t time intervals. We define $x{\uparrow}t \in (M^*)^\omega$ where for $n \geq 1$:

$(x{\uparrow}t)(n) = x(n + t)$ x↑t denotes the timed stream as it appears after the first t sequences of stream x, representing the messages transmitted after the first t time intervals. We get for all $t \in \mathbb{N}$

$x = (x{\downarrow}t)(x{\uparrow}t)$

Given a set of typed channel names

$X = \{c_1$: Tstr $S_1, \ldots, c_m$: Tstr $S_m\}$ by $\overrightarrow{X}$ we denote channel histories given by families of timed streams, one timed stream for each of the channels:

$\overrightarrow{X} = (X \to (M^*)^\omega)$ where we assume that for $x \in \overrightarrow{X}$ the timed stream $x(c_k)$, $1 \leq k \leq m$, for the channel $c_k \in X$ carries messages of type S_k. All introduced notations, such as time abstraction $\overline{x}$, concatenation $s\hat{}x$, time cut x↓t, continuation x↑t, and prefix order carry over to channel histories $x \in \overrightarrow{X}$, as well.

2.2. Interface assertions and interface predicates

Let T_i and S_i be data types; by Tstr T_i and Tstr S_i we denote the data type of timed streams of elements of type T_i and S_i res.; given two sets of typed channel names

$X = \{x_1$: Tstr $T_1, \ldots, x_n$: Tstr $T_n\}$

$Y = \{y_1$: Tstr $S_1, \ldots, y_m$: Tstr $S_m\}$ we denote a syntactic interface of a system by (X▶Y) where X denotes the set of input channels and Y denotes the set of output channels of the syntactic interface of a system. This way, interfaces consist syntactically of channels where each channel is represented by an identifier with a specified data type indicating which types of data are communicated over the channel.

We specify the behavior of systems with this syntactic interface by logical formulas Q in higher order predicate logic with free variables from $\{x_1, \ldots, x_n\} \cup \{y_1, \ldots, y_m\}$ where each of these variables stands for a timed stream of the respective type. The logical formula Q is called *interface assertion* over the syntactic interface (X▶Y). We write

Q::(X▶Y) to express that Q is an interface assertion over the syntactic interface (X▶Y).

Given assertion Q, we write

(X▶Y): Q for the specification of a system with syntactic interface (X▶Y) that fulfills the interface assertion Q.

Given a syntactic interface (X▶Y) we have two semantically equivalent ways to formulate system specifications:

- By interface assertions Q::(X▶Y), being logical formulas which contain the channel identifiers in X∪Y as free logical variables for streams of the indicated type,
- By interface predicates P: $\overrightarrow{X} \times \overrightarrow{Y} \to \mathbb{B}$ over histories $x \in \overrightarrow{X}$ and $y \in \overrightarrow{Y}$.

Each interface assertion Q::(X▶Y) can be transformed into an interface predicate and vice versa. A difficulty arises, however, if the sets of channels X and Y are not disjoint. Then we write for every channel $c \in X \cap Y$ in an interface assertion c to denote the input channel $c \in X$ and c' to denote the output channel $c \in Y$.

This way, we write, for instance,

({a: Tstr M, c: Tstr M}▶{e: Tstr M, c: Tstr Nat}):

$e(1) = \langle\rangle \wedge e(1) = \langle 0 \rangle \wedge \forall t \in \mathbb{N}_+, m \in M$:

$m\#e(t + 1) = m\#a(t) + m\#c(t)) \wedge c'(n + 1) = \langle \#a + \#c \rangle$

for a system specification.

For interface assertions Q::(X▶Y) we use *substitutions*. We write Q[E/c] to replace the identifier c in the assertion Q by expression E. We use the same notation for interface predicates. For a specification $S = (X \blacktriangleright Y)$: Q with assertion Q and channel $c \in X$, $c \notin Y$ and identifier $b \notin X \cup Y$ we write S[b/c] for the specification $((X\backslash\{c\}) \cup \{b\} \blacktriangleright Y)$: Q[b/c]. This way we rename channel identifier c to b both in the syntactic interface (X▶Y) and the interface assertion Q. The same notation is used for $c \in Y$.

3. Interface specifications

In this section, we introduce interface specifications written in the form of templates. We introduce system properties including causality and realizability as well as composition and refinement and illustrate these concepts by examples.

3.1. Notations for system specifications

We do not formally introduce a syntax for writing system specifications but rather introduce a notation by templates for system interface specifications as illustrated by the following brief example:

MIX
in x, z: Tstr M
out y: Tstr M
∀ d ∈ M: d#x + d#z = d#y

The template gives a name to a system together with a specification (in our example MIX), the template describes its syntactic interface (in our example ({x, z: Tstr M}▶{y: Tstr M})) and an interface assertion (in our example ∀ d ∈ M: d#x + d#z = d#y).

In the interface assertion

∀ d ∈ M: d#x + d#z = d#y x, z, and y denote timed streams. MIX is the name of the system, x and z are input channels carrying messages of type M and y is an output channel of type M. Fig. 1 illustrates system MIX with its specifying assertion graphically by a data flow node.

System specifications of this form are called *untimed*, since according to the interface assertion the output data stream y does not depend on the timing of the input streams x and z. As a result, MIX is called *non-time-critical*.

3.2. Causality and refinement

For a system specification A, a system specification B is called *refinement* of A, if the syntactic interface of B includes the syntactic interface of A and if the interface assertion Q_B of B implies the interface assertion Q_A of A:

$Q_B \Rightarrow Q_A$

We give a first example of a refinement of MIX strengthening its specifying assertion:

MIXM
in x, z: Tstr M
out y: Tstr M
MERGE($\bar{x}$, $\bar{z}$, $\bar{y}$)

Here the predicate MERGE is the weakest predicate that fulfills the following specifying formulas: for streams x, y, z ∈ $M^{*|\omega}$

$z = \langle\rangle \Rightarrow$ MERGE(x, z, y) = ($y = x$)

$x = \langle\rangle \Rightarrow$ MERGE(x, z, y) = ($y = z$)

z ≠ ⟨⟩ ∧ x ≠ ⟨⟩ ⇒ MERGE(x, z, y) = ∃ a, b ∈ M*, x', z', y' ∈ $M^{*|\omega}$: $\#a > 0 \wedge \#b > 0 \wedge$

$x = a\hat{}x' \wedge z = b\hat{}z' \wedge (y = a\hat{}b\hat{}y' \vee y = b\hat{}a\hat{}y')$ ∧ MERGE(x', z', y')

The assertion MERGE($\bar{x}$, $\bar{z}$, $\bar{y}$) specifies a refinement of MIXM being a fair merge, the fairness of which guarantees the assertion ∀ d ∈ M: d#x + d#z = d#y. The verification condition for this refinement step therefore is given by the following assertion:

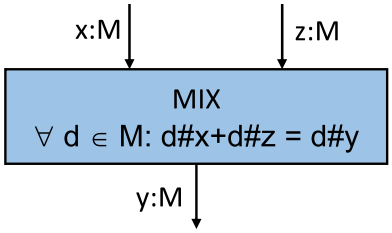

Fig. 1. System MIX as a data flow node.

MERGE($\bar{x}$, $\bar{z}$, $\bar{y}$) ⇒ ∀ d ∈ M: d#x + d#z = d#y

This assertion can be proved by induction . Note that the specification of a fair merge is not so trivial, which underlines the expressive power of the introduced approach.

An important aspect in structuring interfaces is the separation of the set of channels of the interface into input and output channels. This has semantic consequences. We require causality which replaces monotonicity as typically employed in domain theoretic approaches. Causality is addressing the fundamental asymmetry between input and output for an interface consisting of a set of input and output channels carrying timed streams. Causality basically indicates that the output produced till time t may only depend on input received before time t – or in simpler words that the input determines – "is causal for" – the output and not vice versa. Input generated at time t may be arbitrary and does not have to depend on the output produced till time t.

Adding strong causality is a specific form of refinement. The following specification MIXC is a refinement of MIX derived by imposing strong causality.

MIXC
in x, z: Tstr M
out y: Tstr M
∀ d ∈ M: d#x + d#z = d#y
∧ ∀ t ∈ ℕ: d#(y↓(t + 1)) ≤ d#(x↓t)+d#(z↓t)

This step of refinement adding strong causality is expressed by a timing property. MIXC is a refinement of MIX.

An interface predicate Q for the syntactic interface (X▶Y) is called *strongly causal* if for all x, x' ∈ $\overrightarrow{X}$, y ∈ $\overrightarrow{Y}$:

Q(x, y) ∧ x↓t = x'↓t ⇒ ∃ y' ∈ $\overrightarrow{Y}$: Q(x', y') ∧ y↓(t + 1) = y'↓(t + 1)

As shown in (Broy, 2023), for every interface predicate Q there exists a weakest predicate, denoted by Q©, in the set of predicates that are strongly causal and a refinement of Q. It is also shown how strong causality can be imposed onto a specification as a step of refinement. The timing property introduced this way basically expresses that output cannot be produced as an answer to input before that input has arrived and, moreover, for every further input well specified output is available. For a formal justification of causality in terms of implementations by Moore machines, see (Broy, 2023).

We may derive another refinement by introducing a further input channel w into MIX which allows indicating that a number of messages in x and z should be produced as output within the next time interval after they were received as input:

MIXF
in x, z: Tstr M, w: Tstr {◆}
out y: Tstr M
∀ d ∈ M: d#x + d#z = d#y
∧ ∀ t ∈ ℕ: #y↓t + 1 ≥ min((#x↓t)+(#z↓t), #w↓t)

Trivially, MIXF is a refinement of MIX since the interface assertion of MIXF implies the one of MIX while MIXF has more input channels than MIX. The time at which the signals ◆ on w arrive enforces how fast the output on channel y is produced.

We may add further more specific timing properties for MIX introducing, for instance, some deadline $e \geq 1$.

MIXCD
in x, z: Tstr M
out y: Tstr M
∀ d ∈ M: d#x + d#z = d#y
∧ ∀ t ∈ ℕ: d#(y↓(t + 1)) ≤ d#(x↓t)+d#(z↓t) ≤ d#(y↓(t + e))

MIXCD is a refinement of both MIX and MIXC. MIXCD specifies, in addition to strong causality, a deadline e that indicates that for each input after at most e time steps the corresponding output is produced.

We may also specify, in addition, a larger delay $c > 1$ than the one guaranteed by strong causality.

MIXCDD **in** x, z: Tstr M

out y: Tstr M
∀ d ∈ M: d#x + d#z = d#y
∧ ∀ t ∈ ℕ: d#(y↓(t + c)) ≤ d#(x↓t)+d#(z↓t) ≤ d#(y↓(t + e + c))

MIXCDD requires that input messages are forwarded not before c time steps and not later than $e + c$ time steps. MIXCDD is a refinement of MIXC.

As mentioned before, in many systems, deadlines are specified in an absolute manner (i.e., time of day). In the latter case, urgency and deadlines are different concepts, because urgency varies depending on the time when the message was received relative to the absolute deadline. Urgency can be modelled either by giving a concrete physical interpretation for the time intervals or, in addition, by a component sending a timed stream of messages representing absolute time points.

For instance, if we assume that time intervals have the duration of one millisecond, we describe a timer as follows:

Timer

in
out y: Tstr M
∀ k ∈ $\mathbb{N}_+$: y(k*1000) = ⟨time(k)⟩ ∧
∀ t ∈ $\mathbb{N}_+$: (t mod 1000 ≠ 0 ⇒ y(t) = ⟨⟩)

Here time(k) is supposed to return absolute time measured in k seconds starting from a fixed initial time. The output stream y can be used as a timer providing time in absolute terms for other specifications.

3.3. Time abstraction

A step which is generally inverse to refinement is *time abstraction*. We define a specification by the interface assertion Q

S

in x: Tstr D
out y: Tstr E
Q

Its time abstraction is defined as follows:

STA

in x: Tstr D
out y: Tstr E
∃ x', y': Q[x'/x, y'/y] ∧ $\bar{x}$ = $\bar{x}$' ∧ $\bar{y}$ = $\bar{y}$'

Obviously, S ⇒ STA. This shows, S is a refinement of STA. If S and STA are equivalent, then S is called *time-insensitive*.

The time abstraction of the specifications MIXC, MIXCD, and MIXCDD is MIX.

3.4. Realizability

A more sophisticated notion than causality is *realizability*. Realizability of a specification guarantees the existence a computational strategy. If a specification is not realizable it cannot be implemented (see (Broy, 2023)).

For an interface predicate Q::(X▶Y) a strongly causal function f: $\overrightarrow{X} \to \overrightarrow{Y}$ is called *realization*, if $y = f(x)$ implies Q, formally

∀ x ∈ $\overrightarrow{X}$: Q(x, f(x))

Formally, realizability of a specification requires that there exists a strongly causal function that fulfills the specification. If a realization exists for a spec Q. Q is called *realizable*. In (Broy, 2023), it is shown that there exists an implementation of a specification S by a Moore machine M iff S is realizable. A Moore machine M is a refinement of specification S, if the interface predicate defined by Moore machine M is a refinement of S.

A specification by interface predicate Q is *fully realizable* if Q is realizable and there exists a realization f for every pair (x, y) of input and output histories for which Q holds such that $y = f(x)$. The interface predicate defined by a Moore machine is always *fully realizable* (see (Broy, 2023)). For every specification that is realizable there exists a weakest refinement that is fully realizable. Since fully realizable system specifications describe the interface behavior of Moore machines they define a denotational semantics for Moore machines.

We give an example of a specification that is not fully realizable and show its fully realizable refinement. The example is somewhat artificial and only chosen to demonstrate the effects when requiring realizability. Let $e \geq 1$ hold.

MIXNR

in x, z: Tstr M
out y: Tstr M
∀ d ∈ M: d#x + d#z = d#y
∧ ∀ t ∈ ℕ: d#(y↓(t + 1)) ≤ d#(x↓t)+d#(z↓t)
∧ ((d#x + d#z) = ∞ ⇒ ∀ t ∈ ℕ: d#(x↓t)+d#(z↓t) ≤ d#(y↓(t + e)))

This specification MXNR requires that if message d ∈ M occurs infinitely often in the input history, then the output history returns all input messages within the deadline e. However, since the premise (d#x + d#z) = ∞ is a liveness property its validity or invalidity for input streams x and z cannot be observed in finite time, more precisely, by observing finite prefixes of x and y. Thus, in a computation the conclusion, which is a safety property, has to hold in any case. The explanation is that a safety property holds for a history, if it holds for all finite prefixes of the history. A liveness property holds for all finite histories. As explained in detail in (Broy, 2023), if, as in our case, there is a premise in a specification that is a nontrivial liveness property for the input and the conclusion is a safety property for the output, then the safety property has to hold in any case – independent of the fact whether in the end the liveness condition holds or does not.

As a result, we get a fully realizable refinement of MIXNR by MIXFR. Actually, MIXFR is the weakest refinement which is fully realizable. MIXFR results from the specification MIXNR by adding the property of strong causality and deleting the premise (d#x + d#z) = ∞.

MIXFR

in x, z: Tstr M
out y: Tstr M
∀ d ∈ M: d#x + d#z = d#y
∧ ∀ t ∈ ℕ: d#(y↓(t + 1)) ≤ d#(x↓t)+d#(z↓t) ≤ d#(y↓(t + e))

As for strong causality, for every realizable specification there is always a uniquely determined weakest refinement that is a fully realizable specification. In (Broy, 2023), a formula is given that characterizes this weakest fully realizable refinement. If a specification is not realizable a strategy to compute output as specified does not exist.

In fact, full realizability is a rather sophisticated rule of refinement. In practice, specifications may not be strongly causal. Refining a specification by strong causality is often a significant step of refinement of practical relevance, in particular, before composition of the specified systems (see below). In most cases, the resulting strongly causal specification is also fully realizable (as in our example refining MIX to MIXC). Only in special cases of sophisticated specifications, such as for the somewhat artificial example MIXFR, refinements by full realizability may lead to significant additional properties.

3.5. Concurrent composition with feedback

Cyber-physical and communication systems are typically concurrent and distributed. From a design point of view this distribution implies that the system is composed of (or decomposed into)a number of subsystems. Logically, it does not really matter, whether the subsystems are geographically far from each other or whether they are close even on a joint piece of hardware. If the distance may result in a time delay for the connecting channels this can be described by introducing transmission components.

The following approach to distributed systems is independent of hardware platforms and just talks about the logic of composition and interaction. According to the model, the interaction is described by connecting interfaces of two (or more) systems. Actually, this form of connection can also be used to describe the interaction of a system with its context and, in particular, to describe the interaction of a controller with its controlled cyber-physical system. A specific technique for dealing with logical relations between context, such as cyber-physical systems, and system is found in assumption/commitment approaches (see (Broy, 2018)).

Systems are syntactically composable if their sets of output channels are disjoint and for all channel names their types are consistent. We explain composition by considering two composable system specifications where we have for each system one input stream which is kept as input stream after the composition, one input stream that becomes a feedback channel, one output stream that is kept after the composition as output and one output stream which is used for feedback. The composition scheme works, of course, the same way, if we replace each of these channels by sets of channels.

The composition also works for more than two systems since composition is associative and commutative under the assumption that

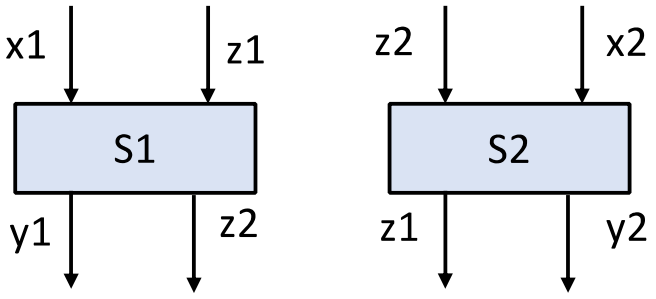

Fig. 2. Two composable systems.

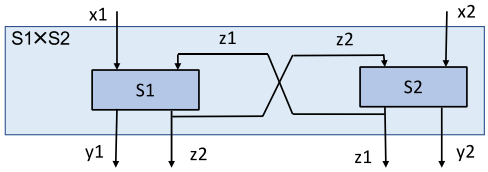

Fig. 3. Composition of systems S1 and S2.

for all composed systems their sets of input channels are pairwise disjoint and their sets of output channels are pairwise disjoint.

In any case, it is most remarkable that the composition of systems in terms of their interfaces is defined in terms of their interface assertions. This means that we compose systems S1 and S2 given by their interface specifications to a system interface specification of the composed system. Composition is carried out at a logical level by composing systems in terms of their specifications with a logical operation as simple as conjunction. This leads to a design and verification calculus where the syntactic interface and the interface behavior of composed systems is derived that way such that properties of composed systems can be captured and verified in terms of the specifications of their subsystems.

Consider the following two specifications that are composable (see Fig. 2).

S1

in x1, z1: Tstr Data
out y1, z2: Tstr Data
Q1

S2

in x2, z2: Tstr Data
out y2, z1: Tstr Data
Q2

Systems S1 and S2 are composed to a specification of the composite system S1 × S2 (see Fig. 3).

S1×S2

in x1, x2: Tstr Data
out y1, y2, z1, z2: Tstr Data
Q1 ∧ Q2

Hiding feedback channels leads to the system S1⊗S2 (see Fig. 4) where the feedback channels are hidden by encapsulation – logically expressed by existential quantification.

Note that S1×S2 is a refinement of S1⊗S2.

S1⊗S2

in x1, x2: Tstr Data
out y1, y2: Tstr Data
∃ z1, z2: Q1 ∧ Q2

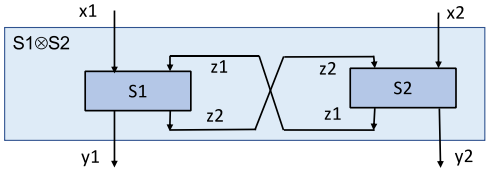

Fig. 4. Composition of systems S1 and S2, hiding feedback.

These rules of composition are sound (see (Broy, 2023)) in the following sense: all properties that are implied by the proposition Q1 ∧ Q2 actually hold for the composed system. If Q1 and Q2 specify the systems S1 and S2 in some incomplete way, then Q1 ∧ Q2 may specify the composed system is some incomplete way, too.

However, assuming that both the composed system S1×S2 (and as well for S1⊗S2) and the systems S1 and S2 are to be implemented by Moore machines, which guarantees that composition is well-defined, then interface assertion Q1 ∧ Q2 may be too weak to prove all of the properties that hold for the composed system under the assumption that the subcomponents are finally realized and implemented by Moore machines. This may be the case if they are not fully realizable. Nevertheless, if assertions Q1 and Q2 are strongly causal and fully realizable then this rule of composition is relatively complete in the following sense (see (Broy, 2023)). In this case, the assertion Q1 ∧ Q2 are strong enough to imply all properties that hold for S1×S2 (and as well as for S1⊗S2). Composition of fully realizable specifications results in fully realizable specifications. The same applies for strong causality. This is the basis of a sound and complete proof calculus (see (Broy, 2024)).

Accordingly, a straightforward approach for concurrent composition is as follows: first refine the specifications of the subsystems to compose to guarantee full realizability. The refined specifications exactly describe the behavior of the most general Moore machines which implement them. When the refined fully realizable specifications are composed the result is a specification of the composed system that is fully realizable, too (or a proof, see (Broy, 2023)). Note that the refined fully realizable specifications represent an extensional semantics for Moore machines. This shows that the specification framework includes as a border case an extensional denotational semantics for the interface behavior of Moore machines and their concurrent composition.

If systems are composed that are far from each other, this can be expressed by delays in the input streams of the systems that mimic that more time is needed until messages arrive. The following specification describes a delay by d time units, $d \in \mathbb{N}$, $d > 0$:

DELAY

in x: Tstr Data
out y: Tstr Data
$\forall\ t \in \mathbb{N}_+$: $y(t + d) = x(t) \wedge (t \leq d \Rightarrow y(t) = \langle\rangle)$

We may compose the delay with the system S1 and get
(S1[x/z2]⊗DELAY)[z2/y]
with the same syntactic interface as S1 and the following interface assertion

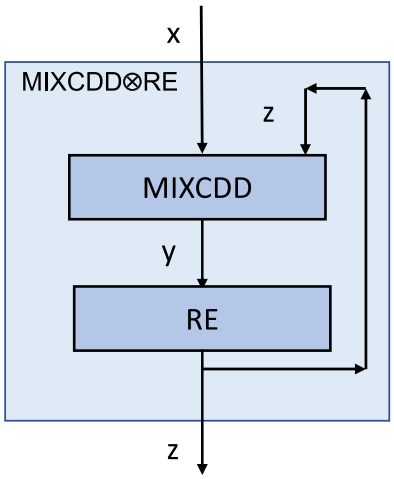

Fig. 5. Composition MIXCDD⊗RE as a data flow diagram (hiding channel y).

$\exists$ x: Q1[x/z2] $\wedge\ \forall\ t \in \mathbb{N}_+$: $z2(t + d) = x(t) \wedge (t \leq d \Rightarrow z2(t) = \langle\rangle)$
with delayed output on z2. Note the technique for renaming channel identifiers as applied here.

3.6. Illustrating examples

In the following, we give two fairly simple examples, where time is a relevant part in composition (for a proof calculus see (Broy, 2024)). In the first example, the timing is not uniquely specified, but time bounds are given and it is shown that composition results in a system with behavior within verifiable time bounds. In the second example, a precise timing is specified for its subsystems – one of which is unreliable with respect to message transmission but not with respect to timing – which is significant for verifying correct transmission in the end.

In the first example, we compose the system MIXCDD (see Section 3.2) with the system RE (see Fig. 5) specified as follows.

RE

in y: Tstr M
out z: Tstr M
$z(1) = \langle\rangle \wedge \forall\ t \in \mathbb{N}$: $1 \leq t \Rightarrow z(t + 1) = y(t)$

The system RE forwards its input on channel y as output on channel z with only one time delay. We conclude $\bar{z} = \bar{y}$. We compose the two systems MIXCDD and RE:

MIXCDD×RE

in x: Tstr M
out y, z: Tstr M
$d\#x + d\#z = d\#y \wedge \forall\ t \in \mathbb{N}$: $d\#(y{\downarrow}(t + c)) \leq d\#(x{\downarrow}t) + d\#(z{\downarrow}t) \leq d\#(y{\downarrow}(t + e + c))$
$\wedge\ z(1) = \langle\rangle \wedge \forall\ t \in \mathbb{N}$: $1 \leq t \Rightarrow z(t + 1) = y(t)$

The composed system specified by MIXCDD×RE forwards messages m received over channel x onto channel z. The message m is therefore also forwarded as input to system MIXCDD leading to repeated output of m on channel z. As a result, every message received on channel x is repeated infinitely often on output channel z. Both specifications MIXDD and RE are fully realizable. As shown in (Broy, 2023) then specification MIXCDD×RE is fully realizable.

We illustrate two methods how to prove properties about system MIXCDD×RE. First we argue without considering specific time. Then we argue more detailed about time.

Recall, we assume $c > 1$. From RE we derive $\bar{z} = \bar{y}$ and from we derive
$d\#\bar{y} = d\#\bar{x} + d\#\bar{z}$ and $d\#(y{\downarrow}(t + 1)) \leq d\#(x{\downarrow}t) + d\#(z{\downarrow}t)$
for all d. We combine this and get (by a simple proof by induction on $t \in \mathbb{N}$) for all $d \in M$:
$(d\#\bar{x} = 0 \Rightarrow d\#\bar{z} = 0) \wedge (d\#\bar{x} > 0 \Rightarrow d\#\bar{z} = \infty)$
Note that this assertion does not refer to time.

To demonstrate how to reason more detailed about time, we consider specific input for MIXCDD×RE with the following property inp for the input stream x with $b \in M$ $inp(x) = (x(1) = \langle b\rangle \wedge \#x = 1)$
This means that inp(x) asserts
$x = \langle\ \langle b\rangle\ \langle\rangle\ \langle\rangle\ \ldots\ \rangle$
By the result proved above we immediately conclude
$(d \neq b \Rightarrow d\#\bar{z} = 0) \wedge b\#\bar{z} = \infty$
This proves that then on channel z and as well on channel y the signal b is repeated over and over again while no other signals occur:
MIXCDD×RE $\wedge$ inp(x) $\Rightarrow \bar{z} = \langle$ b b b … $\rangle \wedge \bar{y} = \langle$ b b b … $\rangle$
But we can also prove properties about the timing of the messages in z for this case; we derive from MIXCDD×RE for all $d \in M$. $d\#x + d\#z = d\#y$
$\wedge\ x(1) = \langle b\rangle$
$\wedge\ \forall\ t \in \mathbb{N}$: $x(t + 2) = \langle\rangle$

$\wedge \forall t \in \mathbb{N}$: $d\#(y{\downarrow}(t+c)) \leq d\#(x{\downarrow}t)+d\#(z{\downarrow}t) \leq d\#(y{\downarrow}(t+e+c))$
$\wedge\ z(1) = \langle\rangle$
$\wedge \forall t \in \mathbb{N}_+$: $1 \leq t \Rightarrow z(t+1) = y(t)$

This gives for all $d \in M$: $d\#y = d\#z$ since for all $t \in \mathbb{N}_+$: $z(t+1) = y(t)$; we derive $d\#x + d\#z = d\#z \wedge d\#(z{\downarrow}(t+c+1)) \leq d\#(x{\downarrow}t)+d\#(z{\downarrow}t) \leq d\#(z{\downarrow}(t+e+c+1))$

Since $b\#x > 0$ we immediately conclude $b\#z = \infty$.

Moreover, for $d \neq b$ we have $d\#(z{\downarrow}(t+c+1)) \leq d\#(x{\downarrow}t)+d\#(z{\downarrow}t)$ and by simple induction on t we get for all t from $d\#x = 0$ and $d\#(z{\downarrow}(t+c+1)) \leq d\#(x{\downarrow}t)+d\#(z{\downarrow}t)$ implies $d\#(z{\downarrow}t) = 0$ and thus $d\#z = 0$. Hence, $\bar{z} = \langle\ b\ b\ \ldots\ \rangle$.

Next, we prove a timing property that indicates a guaranteed minimal and a guaranteed maximal time distance between the instances of message b as they occur infinitely often in the timed stream z. Since b is the only message that occurs in z, we get $b\#(z{\downarrow}n) = \#(z{\downarrow}n)$.

We get from the specification of the composed system for all $t \in \mathbb{N}$:
$x(1) = \langle b\rangle \wedge z(1) = \langle\rangle \wedge \#(z{\downarrow}(t+c+1)) \leq \#(x{\downarrow}t)+\#(z{\downarrow}t) \leq \#(z{\downarrow}(t+e+c+1))$

Define $t_i = \min\{t \in \mathbb{N}: i \leq \#(z{\downarrow}t)\}$; for $i > 0$ the number t_i denotes the time point at which the ith message occurs in z. Then we get $t_0 = 0$

Since $z{\downarrow}c = 0 < z{\downarrow}t_1$ we conclude
$c < t_1$
Moreover, we get since $\#(x{\downarrow}t) = 1$ for $t \geq 1$
$1 \leq \#(z{\downarrow}(1+e+c+1))$ which shows $t_1 \leq e+c+2$ and thus $t_1 < e+c+3$

Further, we get for all $i \in \mathbb{N}$
$\#(z{\downarrow}t_i) = i$
$\#(z{\downarrow}(t_{i+1}{-}1)) = i$
For all $i \in \mathbb{N}$ we get
$\#(z{\downarrow}(t_{i+1}{-}1)) < \#(z{\downarrow}t_{i+1})$
$\#(z{\downarrow}(t_i{+}c+1)) \leq 1{+}\#(z{\downarrow}t_i) \leq \#(z{\downarrow}(t_i{+}e+c+1))$
For $t = t_i$-1 where $0 < i$ we get
$\#(z{\downarrow}(t_i\text{-}1 + c + 1)) < 1{+}\#(z{\downarrow}(t_i\text{-}1))$ and thus (since $\#(z{\downarrow}(t_i\text{-}1)) < \#(z{\downarrow}t_i)$)
$\#(z{\downarrow}(t_i{+}c)) < 1{+}\#(z{\downarrow}t_i) = \#(z{\downarrow}(t_{i+1}))$
We get (since $\#z{\downarrow}n < \#z{\downarrow}m \Rightarrow n < m$) $t_i{+}c < t_{i+1}$
For $t = t_i$ we get by $1{+}\#(z{\downarrow}t_i) = \#(z{\downarrow}t_{i+1})$
$\#(z{\downarrow}(t_{i+1}{-}1)) < \#(z{\downarrow}t_{i+1}) = 1{+}\#(z{\downarrow}t_i) \leq \#(z{\downarrow}(t_i{+}e+c+1))$
and thus $t_{i+1}{-}1 < t_i{+}e+c+1$ and finally $t_{i+1} < t_i{+}e+c+2$ and thus for $i > 0$
$t_i{+}c+1 \leq t_{i+1} \leq t_i{+}e+c+1$
Hence, we get
$\forall i, t \in \mathbb{N}$: $(t_i < t < t_{i+1} \Rightarrow z(t) = \langle\rangle)$

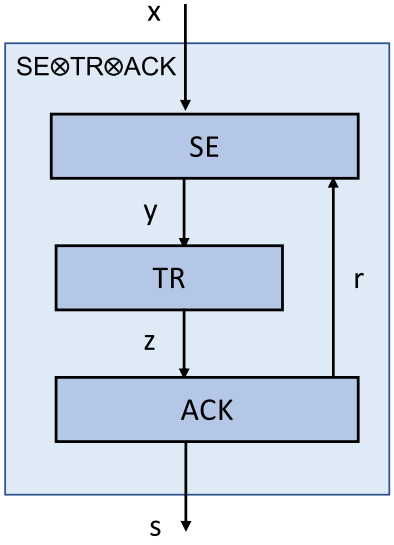

Fig. 6. Composition SE⊗TR⊗ACK.

$\wedge\ z(t_{i+1}) = \langle b\rangle \wedge c < t_1 < e+c+3$
$\wedge\ (i > 0 \Rightarrow t_i{+}c+1 \leq t_{i+1} \leq t_i{+}e+c+1)$

This completes the proof. For all $i \in \mathbb{N}$, $i > 0$, t_{i+1}, which is the time point at which the $(i+1)$th copy of b is sent via z, is in the interval $[t_i{+}c+1: t_i{+}c+e+1]$. The distance between two messages b in z is at least $c+1$ and at most $e+c+1$. Note how much more complicated the reasoning about the timing of the messages in z is.

The second example is similar to the alternating bit protocol (Rich, 2008), but in contrast, its timing is most relevant since time is used for the repeated sending of messages. We study a system as shown in Fig. 6. It specifies the behavior of the sender which receives messages to be sent via channel x, forwards them on channel y and repeats sending of messages until the sender finally gets an acknowledgement via channel r. If a message is not acknowledged after some time period it is sent once more. System TR represents the unreliable transmission that forwards messages but may lose them. However, if an infinite number of messages is sent over TR not all get lost. System ACK forwards the messages it receives via channel z on channel s and acknowledges them on channel r.

To begin with, we specify the transmission component TR:

TR

in y: Tstr Data
out z: Tstr Data
$z(1) = \langle\rangle$
$\wedge \forall t \in \mathbb{N}_+$: $(z(t+1) = y(t) \vee z(t+1) = \langle\rangle)$
$\wedge\ (\#y = \infty \Rightarrow \#z = \infty)$

System TR may forward each of its input message on channel y or TR may fail on forwarding some message, then an input message received on channel y may get lost. However, if the stream of messages on y is infinite, so is the stream on z. In other words, if an infinite number of messages is received on channel y, an infinite number of messages is transmitted on channel z. This is a liveness and also a kind of fairness condition. The specification of TR is strongly causal.

System ACK is quite simple. ACK receives messages via channel z and forwards them in the next time step on its channels s and r.

ACK

in z: Tstr Data
out s, r: Tstr Data
$s(1) = \langle\rangle \wedge r(1) = \langle\rangle$
$\wedge \forall t \in \mathbb{N}_+$: $s(t+1) = z(t) \wedge r(t+1) = z(t)$

Actually, if a message m is transmitted by system TR via channel y it takes two timesteps until success may be acknowledged to the sender SE. Therefore, the sender waits two timesteps after sending to find out whether it receives message m on channel r signaling success[1]; otherwise, receiving no message, it repeats the sending, as long as it does not receive an acknowledgement. The sender repeats sending a message until an acknowledgement is received within two timesteps after the sending of the message. The specification of ACK is again strongly causal.

In the following, we use for timed streams x the notation $x{\Downarrow}$ and $x{\Downarrow}t$ defined as follows. From a timed stream x we get the stream $x{\Downarrow}$ by deleting all entries in x with $x(t) = \langle\rangle$:
$\#x = 0 \Rightarrow x{\Downarrow} = \langle\rangle$
$x(1) = \langle\rangle \Rightarrow x{\Downarrow} = (x{\uparrow}1){\Downarrow}$
$x(1) \neq \langle\rangle \Rightarrow (x{\Downarrow})(1) = x(1) \wedge (x{\Downarrow}){\uparrow}1 = (x{\uparrow}1){\Downarrow}$

[1] This may require access to a local time source, which, in real systems, may not always be aligned with the synchronous global time reference.

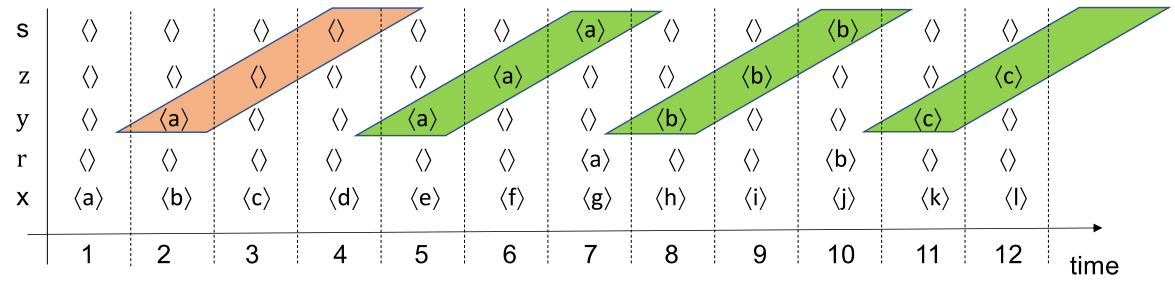

Fig. 7. Example of corresponding values for the streams x, r, y, z, s, communication with loss of a message in red, communications with successful transmission in green.[21]

$x\Downarrow t = (x\downarrow t)\Downarrow$

We define the predicate send(t, x, r, y) for $t \in \mathbb{N}_+$ which is used to specify the interface assertion for the sender SE. We define:

$Send(1, x, r, y) = (y(1) = \langle\rangle)$

$Send(2, x, r, y) = ((x(1) = \langle\rangle \Rightarrow y(2) = \langle\rangle) \wedge (x(1) \neq \langle\rangle \Rightarrow y(2) = x(1)))$

We define for $t \in \mathbb{N}$, $t \geq 2$:

$Send(t + 1, x, r, y)$

$= ((\#(x\downarrow t) \leq \#(r\downarrow t) \Rightarrow y(t + 1) = \langle\rangle)$

$\wedge\ ((\#(x\downarrow t) > \#(r\downarrow t) \wedge y(t) = \langle\rangle \wedge y(t\text{-}1) = \langle\rangle) \Rightarrow y(t + 1) = (x\Downarrow t)((\#r\Downarrow t)+1))$

$\wedge\ ((\#(x\downarrow t) > \#(r\downarrow t) \wedge (y(t) \neq \langle\rangle \vee y(t\text{-}1) \neq \langle\rangle)) \Rightarrow y(t + 1) = \langle\rangle))$

We specify the sender SE by the interface assertion

$\forall\ t \in \mathbb{N}_+$: Send(t, x, r, y)

The assertion sent(t, x, r, y) specifies the message y(t) depending on the streams $x\downarrow t\text{-}1$ and $r\downarrow t\text{-}1$. Based on this definition, we specify the interface behavior of the sender SE.

SE

in x: r: Tstr Data,
out y: Tstr Data
$\forall\ t \in \mathbb{N}_+$: Send(t, x, r, y)

We compose the three systems in terms of their specifications and get (see Fig. 6):

SE⊗TR⊗ACK

in x: Tstr Data
out s: Tstr Data
$\exists$ r, y, z: $\forall\ t \in \mathbb{N}_+$: Send(t, x, r, y)
$\wedge\ z(1) = \langle\rangle \wedge r(1) = \langle\rangle \wedge r = s$
$\wedge\ \forall\ t \in \mathbb{N}_+$: $((z(t + 1) = y(t) \vee z(t + 1) = \langle\rangle) \wedge r(t + 1) = z(t)) \wedge (\#y = \infty \Rightarrow \#z = \infty)$

The proof that we sketch in the following shows $SE\times TR\times ACK \Rightarrow \bar{z} = \bar{x} \wedge \bar{z} = \bar{s}$ and thus $SE\otimes TR\otimes ACK \Rightarrow \bar{s} = \bar{x}$.

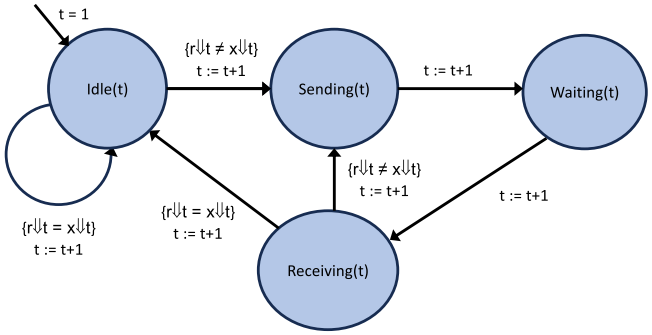

Fig. 8. State transitions for the invariant from t to $t + 1$.

Fig. 7 illustrates by an example the communication exchanged by the streams between the three systems involved. The message on channel r indicates successful transmission. It shows an interaction where the message a is sent, but not acknowledged; therefore, a is resent, in this case acknowledged and sending is resumed by sending b and then c.

The system SE×TR×ACK is supposed to imply correct transmission expressed by $x\Downarrow = s\Downarrow$. We give only a sketch of the proof of correctness of SE×TR×ACK. Let for streams x, r, y, z, s the assertion of the system SE×TR×ACK hold. We define the following assertions for $t \in \mathbb{N}_+$:

$Idle(t) = (r\Downarrow t\text{-}1 = x\Downarrow t\text{-}1 \wedge y(t) = \langle\rangle \wedge z(t) = \langle\rangle \wedge r(t) = \langle\rangle)$

$Sending\ (1) = \text{false}$

$Waiting\ (1) = Waiting(2) = \text{false}$

$Receiving\ (1) = Receiving(2) = Receiving(3) = \text{false}$

$Sending\ (t + 1) = (r\Downarrow t < x\Downarrow t \wedge y(t + 1) = (x\Downarrow t)((\#r\Downarrow t)+1) \wedge (Idle(t) \vee Receiving(t))$

$Waiting\ (t + 2) = (y(t + 2) = \langle\rangle \wedge Sending(t + 1))$

$Receiving\ (t + 3) = (y(t + 3) = \langle\rangle \wedge Waiting(t + 2))$

For Fig. 7 we get Idle(1), Sending(2), Waiting(3) and Receiving(4) and so on. Note that $r\Downarrow t + 1 = z\Downarrow t$.

We can prove by induction over t for $t \in \mathbb{N}_+$:

$Sending(t + 1) \Rightarrow (z(t + 1) = \langle\rangle \wedge r(t + 1) = \langle\rangle)$

$Waiting(t + 2) \Rightarrow (y(t + 2) = \langle\rangle \wedge r(t + 2) = \langle\rangle)$

$Receiving(t + 3) \Rightarrow (y(t + 3) = \langle\rangle \wedge z(t + 3) = \langle\rangle)$

We prove by induction over t for all $t \in \mathbb{N}_+$ the invariant GenInv(t) defined by:

$GenInv(t) = ((r\Downarrow t \sqsubseteq z\Downarrow t \sqsubseteq x\Downarrow t) \wedge (Idle(t) \vee Sending(t) \vee Waiting(t) \vee Receiving(t))$

We give a sketch of the proof by induction over $t \in \mathbb{N}_+$.

Fig. 8 shows the transitions for the assertions Idle, Sending, Waiting, and Receiving for $t \in \mathbb{N}_+$. Note that if assertion $Sending(t + 1)$ holds then $Waiting(t + 2)$ and $Receiving(t + 3)$ hold which implies

$(Sending(t + 4) \wedge r\Downarrow t + 3 < x\Downarrow t + 3) \vee (Idle(t + 4) \wedge r\Downarrow t + 3 = x\Downarrow t + 3)$

In the beginning we have Idle(1); then the step from t to $t + 1$ is proved by the following assertions:

$Idle(t) \wedge r\Downarrow t < x\Downarrow t \Rightarrow Sending(t + 1)$

$Idle(t) \wedge r\Downarrow t = x\Downarrow t \Rightarrow Idle(t + 1)$

$Sending(t) \Rightarrow Waiting(t + 1)$

$Waiting(t) \Rightarrow Receiving(t + 1)$

$Receiving(t) \wedge r\Downarrow t < x\Downarrow t \Rightarrow Sending(t + 1)$

$Receiving(t) \wedge r\Downarrow t = x\Downarrow t \Rightarrow Idle(t + 1)$

This proves that GenInv(t) is an invariant; it holds for all $t \in \mathbb{N}_+$. This shows the safety conditions for the system SE×TR×ACK.

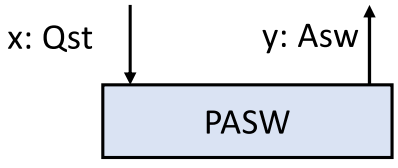

Fig. 9. Answering system.

Proving liveness is straightforward: assuming z⇓ ≠ x⇓ then there is an infinite number of t ∈ $\mathbb{N}_+$ with Sending(t); thus y⇓ is infinite and by the specification of TR the stream z⇓ is infinite. This leads to a contradiction. Thus z⇓ = x⇓ and since z⇓ = s⇓ we get s⇓ = x⇓ and thus correct transmission.

This is a fairly simple example, but it demonstrates how specification and reasoning by and about timing is carried out. More practical examples may be communication systems such as CAN busses in cars or time triggered protocols.

3.7. A more comprehensive example

Finally, we consider a short simple development history to demonstrate, how interface specifications, time, and causality work together. We consider a question answering system with the following data types

Qst→Data type of questions

Asw→Data type of answers with a predicate cqa: Qst × Asw → $\mathbb{B}$→Relationship between questions and correct answers

The proposition cqa(q, w) yields true iff w is a correct answer to question q; we assume that for every question there exists a correct answer.

We specify the system PASW that correctly answers all asked questions.

PASW

in x: Tstr Qst
out y: Tstr Asw
$\#x = \#y \wedge \forall\ k \in \mathbb{N}_+$: $k < (\#x)+1 \Rightarrow$ cqa(x[k], y[k])

PASW receives a stream of questions and responds by a stream of correct answers (see Fig. 9).

Of course, it is reasonable to assume that questions are never answered before they have been asked. This is expressed by causality. PASWC is the weakest refinement of PASWA that is strongly causal (and fully realizable, too).

PASWC

in x: Tstr Qst
out y: Tstr Asw
$\#x = \#y$
$\wedge\ \forall\ k \in \mathbb{N}_+$: $(k < (\#x)+1 \Rightarrow$ cqa(x[k], y[k]))
$\wedge\ \forall\ t \in \mathbb{N}$: $\#(y{\downarrow}(t+1)) \leq x{\downarrow}t$

To improve understanding and readability, we may introduce abbreviations for some of the interface assertions:

All_Questions_Answered = $(\#x = \#y)$

All_Answers_Correct = $\forall\ k \in \mathbb{N}_+$: $k < (\#y)+1 \Rightarrow$ cqa(x[k], y[k])

Causal_Behavior = $\forall\ t \in \mathbb{N}$: $\#(y{\downarrow}(t+1)) \leq x{\downarrow}t$

We get a perhaps more readable specification:

PASWC

in x: Tstr Qst
out y: Tstr Asw
All_Questions_Answered ∧ All_Answers_Correct ∧ Causal_Behavior

The next specification requires, in addition, that new questions must arrive only after all questions asked before have been answered – one question at a time, however, the specification is not including strong causality.

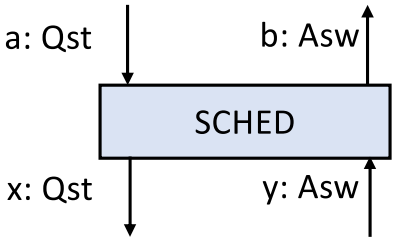

Fig. 10. Scheduler for one-at-a-time question answering system.

PASWE

in x: Tstr Qst
out y: Tstr Asw
$(\forall\ t \in \mathbb{N}$: $\#(x{\downarrow}(t+1)) \leq 1+\#(y{\downarrow}t)) \Rightarrow (\#x = \#y \wedge (\forall\ k \in \mathbb{N}_+$: $k < (\#x)+1 \Rightarrow$ cqa(x[k], y[k])))

We may introduce a new abbreviation

One_Question_at_a_Time = $(\forall\ t \in \mathbb{N}$: $\#(x{\downarrow}(t+1)) \leq 1+\#y{\downarrow}t)$ and get

PASWE

in x: Tstr Qst
out y: Tstr Asw
One_Question_at_a_Time ⇒ (All_Questions_Answered ∧ All_Answers_Correct)

The assertion One_Question_at_a_Time represents an assumption about the input in relation to the output received so far (see (Broy, 2018)). Note PASWE is not a refinement of PASWA, since the requirement of causality is left out.

For specification PASWE adding strong causality results in the following refined specification:

PASWA

in x: Tstr Qst
out y: Tstr Asw
$\forall\ t \in \mathbb{N}$: $(\forall\ t' \in \mathbb{N}$: $t' < t \Rightarrow \#(x{\downarrow}(t'+1)) \leq 1+\#(y{\downarrow}t')) \Rightarrow$
$(\forall\ k \in \mathbb{N}_+$: $k \leq \#(y{\downarrow}(t'+1)) \Rightarrow$ cqa(x[k], y[k]))
$\wedge\ \#(y{\downarrow}(t'+1)) \leq \#(x{\downarrow}t'))$
$\wedge\ (\forall\ t \in \mathbb{N}$: $\#(x{\downarrow}(t+1) \leq 1+\#(y{\downarrow}t)) \Rightarrow \#x = \#y)$

Once more, we may introduce abbreviations

One_Question_at_a_Time_Until(t) = $(\forall\ t' \in \mathbb{N}$: $t' < t \Rightarrow \#x{\downarrow}(t'+1) \leq 1+\#y{\downarrow}t')$

Questions_Correctly_Answered_Until(t) = $(\forall\ k \in \mathbb{N}_+$: $k \leq \#y{\downarrow}t \Rightarrow$ cqa

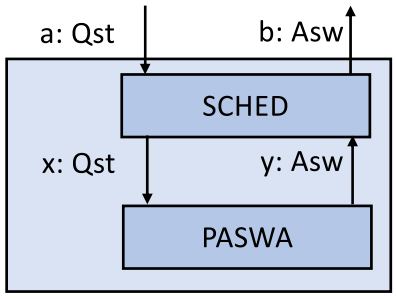

Fig. 11. Architecture for the system SCHED ⊗ PASWA.

[2] This figure clearly illustrates the global synchronous time model assumption. Note that the 2 time-frame delay is based on timing information kept locally and which is presumed to be perfectly synchronized with whatever serves as the global time reference (Selic, 2020).

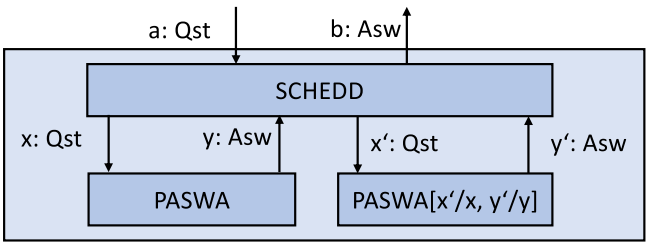

Fig. 12. Architecture of SCHEDD ⊗ PASWA ⊗ PASWA[*x'/x, y'/y*], the Scheduler for Two Question Answering Systems.

(x[*k*], y[*k*]))

No_Answer_before_Question_Until(t) = (#y↓($t+1$) ≤ #x↓t) and get:

PASWA

in x: Tstr Qst
out y: Tstr Asw
∀ t ∈ ℕ:
One_Question_at_a_Time_Until(t) ⇒ (Questions_Correctly_Answered_Until($t+1$)
∧ No_Answer_before_Question_Until(t))

This specification expresses that as long as at every time there is at most one unanswered question in the input stream x all questions are eventually correctly answered.

We now define a scheduler (see Fig. 10) that guarantees the assumption only one question at a time, but requires the assumption that each question is answered to guarantee $\#a = \#b$. We give the following specification.

SCHED

We compose SCHED with PASWA (see Fig. 11) and get

SCHED × PASWA

in a: Tstr Qst
out b: Tstr Asw, x: Tstr Qst, y: Tstr Asw
$\bar{x} \sqsubseteq \bar{a} \wedge \bar{b} = \bar{y}$
∧ ∀ t ∈ ℕ:
#x↓($t+1$) = min(#(a↓t), 1+#(y↓t)) ∧ #(b↓($t+1$)) = #(y↓t)
∧ (∀ t‘ ∈ ℕ: t‘ < t ⇒ #x↓t‘ ≤ 1+#(y↓(t‘+1))) ⇒ (∀ k ∈ $\mathbb{N}_+$: $k \le$ #(y↓($t+1$) ⇒ cqa(x[*k*], y[*k*]))
∧ #(y↓($t+1$)) ≤ #(x↓t))
∧ (∀ t ∈ ℕ: #(x↓($t+1$) ≤ 1+#(y↓t)) ⇒ $\#x = \#y$)

SCHED guarantees the assumption of PASWA:

SCHED ⇒ ∀ t ∈ ℕ: #x↓t ≤ 1+#y↓($t+1$)) since (if $t = 0$ the assumption trivially holds)

#(x↓($t+1$)) = min(#(a↓t), 1+#(y↓t)) ≤ 1+#(y↓t) ≤ 1+#(y↓($t+2$))

Under this assumption PASWA guarantees (proof by induction):

$\#x = \#y \wedge$ (∀ k ∈ $\mathbb{N}_+$: $k \le$ #y ⇒ cqa(x[*k*], y[*k*]) and thus since $\bar{x} \sqsubseteq \bar{a} \wedge \bar{b} = \bar{y}$ we get by $\#x = \#y$ and ∀ t ∈ ℕ: #(x↓($t+1$)) = min(#a, #(y↓($t+1$))) the assertion $\#x = \#a$ and finally

$\#a = \#b \wedge$ (∀ k ∈ $\mathbb{N}_+$: $k \le$ #(b↓($t+1$)) ⇒ cqa(a[*k*], b[*k*]) which is the interface assertion of PASW[*a/x, b/y*]

We get (see Fig. 11) by simplifying the interface assertion

SCHED × PASWA

in a: Tstr Qst

(*continued on next column*)

(*continued*)

out b: Tstr Asw, x: Tstr Qst, y: Tstr Asw
$\bar{x} = \bar{a} \wedge \bar{b} = \bar{y} \wedge \#x = \#y$
∧ ∀ t ∈ ℕ: #(x↓($t+1$)) = min(#a↓t, 1+#(y↓t)) ∧ #(b↓($t+1$)) ≤ #(y↓t) ∧ #(y↓($t+1$)) ≤ #(x↓t)
∧ (∀ k ∈ $\mathbb{N}_+$: $k \le$ #y ⇒ cqa(x[*k*], y[*k*]))

and, moreover, for SCHED ⊗ PASWA (hiding the internal feedback streams x and y and as a result also the involved timing constraints)

SCHED ⊗ PASWA

in a: Tstr Qst
out b: Tstr Asw
$\#a = \#b \wedge \forall$ k ∈ $\mathbb{N}_+$: $k \le$ #b ⇒ cqa(a[*k*], b[*k*])
∧ ∀ t ∈ ℕ: #(b↓($t+1$)) ≤ #(a↓t)

This proves

SCHED ⊗ PASWA ⇒ PASWC[*a/x, b/y*]

Note that we might also take this equation as the requirement specification for SCHED and then understand its specification as the design. Then the proof verifies the design.

Finally, we consider a scheduler SCHEDD which manages two subsystems PASWA (see Fig. 12):

SCHEDD ⊗ PASWA ⊗ PASWA[*x'/x, y'/y*] where SCHEDD is specified as follows: its syntactic interface is specified by

SCHEDD: ({a: Qst, y, y': Asw}▶{b: Asw, x, x': Qst})

We specify the behavior of SCHEDD in a number of steps.

We include requirements of PASWA as assumptions for SCHEDD about x and y and x' and y' as well:

ASU(x, y) = (∀ t ∈ ℕ: (∀ t' ∈ ℕ: t' ≤ t ⇒ #(x↓(t'+1) ≤ 1+#(y↓t')) ⇒ #(y↓$t+1$) ≤ #(x↓t)))

∧ (∀ t ∈ ℕ: #(x↓($t+1$)) ≤ 1+#(y↓t)) ⇒ $\#y = \#x$)

We introduce functions h: {n ∈ $\mathbb{N}_+$: $n \le$ #a} → {$\mathbb{N}_+$} and p: {n ∈ $\mathbb{N}_+$: $n \le$ #a} → {1, 2} where h and p define the behavior of the "scheduler" for the system SCHEDD. Let n be the n-th question on input stream a. If p(n) = 1 then the question number n is forwarded on channel x at time h(n), if p(n) = 2 the question is forwarded on channel x' at time h(n). The number h(n) denotes the time at which the n-th question is forwarded such that the following propositions hold:

∀ n, n' ∈ [1:#a]: $n < n' \Rightarrow$ (h(n) < h(n') ∨ (h(n) = h(n') ∧ p(n) < p(n')))

Question a[n] is scheduled as soon as possible on channel x or x' but after time t if #a↓$t < n$:

∀ n ∈ $\mathbb{N}_+$: $n \le$ #a ⇒
(h(n) ≥ 1+min{t ∈ $\mathbb{N}_+$: #(a↓t) ≥ n}
∧ (∀ t ∈ ℕ: min{t ∈ $\mathbb{N}_+$: #(a↓t) ≥ n} < t < h(n) ⇒ busy(t)))
where busy(t) = (#x↓t > #(y↓(t-1)) ∧ #x'↓t > #(y'↓(t-1)))

p indicates which of the two subsystems are selected to calculate the answer for question a[n]:

(p(n) = 1 ∧ x(h(n)) = ⟨a[n]⟩ ∧ #(x↓(h(n)−1)) = #(y↓(h(n)−1)))
∨ (p(n) = 2 ∧ x'(h(n)) = ⟨a[n]⟩ ∧ #(x'↓(h(n)−1)) = #(y'↓(h(n)−1)))

A question is never sent on x or x' before the previous question has been answered on y or y' resp.:

#(x↓($t+1$)) ≤ 1+#(y↓t) ∧ #(x'↓($t+1$)) ≤ 1+#(y'↓t)

Answers are forwarded on b one after the other in the same order in which the question are asked on a and only after the corresponding answers are available on y or y'

(p(n) = 1 ∧ b[n] = y[#(x↓h(n))] ∧ (#(b↓($t+1$)) = n ⇒ #(y↓t) ≥ #(x↓h(n))))
∨ (p(n) = 2 ∧ b[n] = y'[#(x'↓h(n))] ∧ (#(b↓($t+1$)) = n ⇒ #(y‘↓t) ≥

#(x'↓h(n))))

In addition, we require that answers are forwarded from y and y' as soon as possible:

#(b↓(*t* + 1)) = #(y↓t) + #(y'↓t)

SCHEDD

in a: Tstr Qst, y, y': Tstr Asw
out b: Tstr Asw, x, x': Tstr Qst
ASU ∧ ASU[x'/x, y'/y] ⇒
∃ h: {n ∈ $\mathbb{N}_+$: *n* ≤ #x} → {$\mathbb{N}_+$}, p: {n ∈ $\mathbb{N}_+$: *n* ≤ #x} → {1, 2}:
(∀ n, n' ∈ $\mathbb{N}_+$: (*n* + 1 ≤ #a ∧ n'+1 ≤ #a ∧ *n*<n' ⇒ (h(n) < *h*(n') ∨ (h(n) = *h*(n') ∧ p(n) < *p*(n')))
∧ h(n) > 1+min {t ∈ $\mathbb{N}_+$: #a↓*t* ≥ *n*} ∧ (∀ t ∈ $\mathbb{N}$: min {t ∈ $\mathbb{N}_+$: #a↓*t* ≥ *n*} < *t* < *h*(n) ⇒ busy(t))
∧ ((p(n) = 1 ∧ x[h(n)] = ⟨a[*n*]⟩ ∧ #x↓h(n)−1 = #y↓h(n)−1)
∨ (p(n) = 2 ∧ x'[h(n)] = ⟨a[*n*]⟩ ∧ #x'↓h(n)−1 = #y'↓h(n)−1))
∧ (∀ t ∈ $\mathbb{N}$: #(x↓(*t* + 1)) ≤ 1+(#y↓t) ∧ #(x'↓(*t* + 1)) ≤ 1+#(y'↓t)
∧ ((p(n) = 1 ∧ b[n] = *y*[#(*x*↓*h*(*n*))]) ∨ (p(n) = 2 ∧ b[n] = y'[#(*x*'↓*h*(*n*))]))
∧ #(b↓(*t* + 1)) = #(y↓t) + #(y'↓t)

The specification SCHEDD is quite difficult to express precisely in formal language. We get

SCHEDD ⊗ PASWA ⊗ PASWA[x'/x, y'/y] ⇒ PASW[*a/x, b/y*]

This can be proved along the lines demonstrated for SCHED above. By SCHEDD we get a more comprehensive specification of a system using a scheduler which uses the service of two identical question answering systems working in parallel on a stream of tasks.

Note that the system SCHEDD is an example of a layer as we find it in layered architectures.

4. On methodology

Some of the most significant steps in the development of systems including specifications dealing with time are demonstrated in Section 3 by small examples. On these basis, we briefly describe how these steps contribute to major principles in system development.

4.1. Interface specification and refinement

The theoretical foundation, as worked out in (Broy, 2023), is designed with the motivation to address basic practical goals. The most important one is *interface abstraction* with the goal to describe by interface models the behavior of systems as it is relevant for their interaction with their contexts. The theory provides an abstract interface model to describe the functional behavior of systems. By this model both specifications of interfaces of systems are captured as well as the interface behavior of systems. Interface models are the basis of functional specification and functional requirements. The model follows the principles of encapsulation and information hiding (see Section 4.3 below).

By the specification technique the classical concept of refinement is supported. For given specifications, by strong causality and full realizability a specification is refined to a predicate that defines the interface behavior of the most general implementation of the specification by a Moore machine, provided the specification is implementable, otherwise the interface specification cannot be implemented and is equivalent to false.

A system interface specification defines the *extensional* behavior of a Moore machine since it does not specify the states but just properties of the relation between the streams of input and output data. In fact, very general and incomplete interface specifications may be written initially. The resulting specification does not have to be strongly causal nor fully realizable for using the specified system as part of a design. Then the specification can be refined and improved until all required properties are captured. However, refinement by strong causality and full realizability is always a possible and insightful step for discovering and verifying further system interface properties that hold for all implementations.

4.2. Concurrent composition and modularity

Concurrent composition of syntactically composable systems is defined in terms of their interface specifications by logical conjunction. Given two syntactically composable systems with interface behavior specified by the predicates Q_1 and Q_2 as well as two specifications R_1 and R_2 of these systems that are refinements of Q_1 and Q_2 resp.; formally

$R_i \Rightarrow Q_i$ for $i = 1, 2$

Then we have

$R_1 \times R_2 \Rightarrow Q_1 \times Q_2$ since $R_1 \wedge R_2 \Rightarrow Q_1 \wedge Q_2$

This shows that in an architecture with component specifications the refinement of the component specifications leads to refinement of the architecture specification. The approach is *modular* since independent refinements of component specifications leads to refinements of the specifications of the composed system.

4.3. Encapsulation and information hiding

What should be underlined is that the approach follows classical principles of software and systems engineering:

- Interfaces provide an abstract description of the functional behavior of systems in a modular way (including "functional" timing properties, see (Broy, 2015)) that follows the principle of *encapsulation*.
- *Encapsulation*[3] is a principle to get access to systems only through their specified interfaces. In the *interface specification*, it is also described what the reaction is for certain input to the system.
- The approach does not only follow the principle of encapsulation, but also that of *information hiding*. Information hiding means here that we do not need to look inside systems. It is enough to know the specification their interface behavior.
- The approach is *modular* in the sense that the specification of the interface behavior of a composed system is derived from the specifications of the interface behavior of the systems to be composed and that (specifications of) subsystems can be refined independently (see Section 4.2 and Section 4.3).

This methodology follows also the principle of contracts. It is possible to work with assertions that represent contracts consisting of assumptions and commitments which are just special patterns of interface assertions (see (Broy, 2018)).

4.4. Formal and informal specifications

As we have demonstrated by fairly simple completely formal examples, we can specify all kinds of systems behaviors in terms of their interfaces, including timing, in our approach using interface assertions. From a practical point of view, of course, a fully formal treatment of large systems might be too demanding and more pragmatic approaches may be preferred. It is obvious, however, that the presented approach may be combined with more pragmatic techniques such as specifications by natural language (see Section 3.7) or interaction diagrams or diagrams from SysML and that it can be also supported by case tools. However, we believe that also for a pragmatic approach, a formal foundation with a scientifically justified theory is indispensable for understanding the task of specification (see (Broy, 2024)). A formal foundation gives justifications for pragmatic development steps and provides the necessary certainty that the overall approach is scientifically justified – a justification which is required anyhow by good

[3] In (Selic, 2020), Bran Selic argues that perfect encapsulation cannot be realized in practice, because implementations of software components inevitably leak into the platform.

engineering practice. However, we must understand the limitations of formal foundations – knowing and recognizing them (as opposed to unquestionably trusting any formal conclusions drawn by such methods).

In the discussions about so-called formal methods, we find two extreme points of view:

- Practitioners often say that formal methods do not really help, are too demanding, too difficult to use, too time consuming, overstrain practitioners, do not scale, and cannot be applied in a practical context. Moreover, there are always important aspects which are not captured by the formal models (see (Selic, 2020), private communication).
- Protagonists of formal methods state that the only way to ensure that a program works correctly is to use formal techniques where everything which is constructed is justified on formal grounds.

In some sense, both positions sound reasonable. The formal method point of view has to consider, however, the fact that in many cases bugs and errors are not only in the implemented code, but rather in the specifications. This shows that in the end there cannot be a perfect formal justification of software and software-intensive systems since verification can be carried out completely formally, while for capturing functional requirements and their validation formal methods may help but these steps depend to a large extend on informal judgement based on common sense. The result of applying formal techniques is only as good as our ability to grasp, to identify and to formulate valid requirements. In fact, this holds also for very practical work. For instance, let us look at safety issues in cars which are regulated, for instance, by the ISO standard 26,262. It took quite some time until practical experts realized that functional safety is not sufficiently achieved by only showing that a system behaves as specified even in situations where failures may occur in the execution platform. Only later, it was realized that by these arguments systems are only as safe as their specification is safe. Today, engineers are working towards a standard called SOTIF (safety of the intended functionality) with the goal to make sure that systems are safe as specified which, by the way, can never be proved formally in the end. Thus, sufficient safety needs a systematic development of requirements and discussions of all relevant use cases.

It is unrealistic to believe in formal methods as a silver bullet, but it is not wise to believe in “pragmatic” methods for which a formal justification has not been or even cannot be given. This leads in the end to the following principle:

> A method for software and systems engineering should always be justified on the ground of an adequate scientific theory.

It should be proved that methods fulfill all the expectations assumed for them, however, they should be “simple-to-use” methods with manifestations in a more informal or semi-formal style. In any case, it should be ensured that the general concept including the method is scientifically justified. Moreover, advanced tool support including partial automation of development steps requires a sound and sufficiently complete calculus, anyway.

4.5. Methodological essentials

Besides refinement we consider abstraction in system development. For non-time-critical systems, which are systems where the output data do not depend on the timing of the input data, time abstraction is a helpful step in development. Although not a refinement step, in general, it allows us to concentrate on certain properties of data-flow and getting rid of timing considerations.

However, when working towards system composition, we may do a refinement adding strong causality and – in special cases – even full realizability for system components to get a specification for the composed system (for a proof see (Broy, 2023)) that is strong enough to show all relevant properties of a composite system. Actually, in the case of fully realizable specifications, the specifications include all the properties that hold for all implementations. We can get rid of these properties again by time abstraction, but we may also stick to these properties, because they give us additional hints about the specific timing of systems according to the question, how they are composed.

The result is a very flexible, powerful and expressive method[4] for the specification and design of systems in terms of their interface behaviors, of composition, and of dealing with time-critical properties. In addition, the method allows for a flexible combination of time-critical and time insensitive systems, introducing time whenever appropriate and convenient or needed according to the application or to do certain proofs, but also to be able to get rid of time by abstraction.

What is very significant in the system model is that we do not use artificial theoretical concepts whose purpose is only to manage certain theoretical aspects. Examples of those concepts are monotonicity of functions for dealing with feedback and recursion as well as continuous functions to be able to prove that fixpoints coincide with the least upper bound of function iteration.

What is definitely needed in the end is a carefully designed notation, easy to write and easy to read, expressive enough to formulate specifications and to support all kinds of development steps. Furthermore, as always, adequate tool support is indispensable.

4.6. Expressive power

Which properties of systems can be described by a specification and modeling language determines its expressive power. In our case, we use higher order predicate logic – based on specification of abstract data types – and a interface model to specify extensionally system behaviors. This is a very powerful description technique that includes the power to express all kinds of computable functions and this way all classical properties of programming languages. Interface assertions describe relations between input and output streams. Thus underspecification and nondeterminism can be addressed. According to the timed model, also real time properties can be expressed.

A specific technique, which we have demonstrated in Section 3.2 and in more detail in (Broy, 2023, 2024) is the use of recursive definitions of assertions and predicates. This is per se a very powerful technique and closely related to recursive functions. This technique can be complemented by graphical illustrations and descriptions. However, in contrast to approaches like SysML, where first the graphics is selected and then its meaning is discussed, we suggest to work the other way around, to start with a textual language based on a modeling theory providing a formally defined meaning and then to enhance the language by graphical constructs (see (Broy and Rumpe, 2023)).

4.7. Interfaces and architectures

The introduced approach to specify, refine, implement, compose, and verify interfaces is, in particular, useful when designing and refining

[4] In fact, it may be worthwhile putting more emphasis on this aspect of the approach; i.e., that it is best suited to specifying and validating system *specifications* as opposed to system *implementations*. A separate – and much needed – methodology is required showing how to approach the problem of realizing such specifications given the imperfect nature of implementation technologies. This could prove to be very useful and a very fruitful line of research (Selic, 2020).

architectures. Based on concurrent composition, all kinds of architectural patterns can be captured and formally specified such as, for instance, layered architectures (see (Herzberg and Broy, 2005)[5]). This supports in a very straightforward, direct way the composition of subsystems into architectures in terms of their interface specifications leading to interface specifications for the composed systems (hiding feedback channels) as well as specifications for the architectures (leaving feedback channels visible).

This design steps can be applied both in top down and in bottom up development and mixtures thereof. Systems and their architectures can be specified this way at several levels of abstraction which are formally related.

5. Concluding remarks

We have presented an approach for designing distributed interactive systems including real time properties for the specification, refinement, concurrent composition, and verification of software-intensive systems. Distributed interactive systems are well-known for their difficulties in finding appropriate models. Starting from early work, more or less motivated by operating systems, shared memory had to be considered with all its problems concerning non-determinism, as caused by interleaving, critical regions and so on. It is obvious that for distributed systems shared memory[6] does not seem to be the most appropriate concept not only for its difficulties from a modelling and methodological point of view, but also since important concepts in software and systems engineering are distribution, explicit concurrency, encapsulation of states, and access onto systems by specified interfaces.

To give precise semantics for systems poses a number of problems that are related to inherent underspecification and the nondeterminism one has to deal with for software-intensive, concurrent, distributed systems. The approach that we have presented has the advantage that the techniques for modelling interaction behavior and for characterizing the principle of causal feedback is expressed with the help of time. As a result, we get a model where time is integrated into the model in a coherent way. The reasoning about the properties of feedback communication is characterized in a homogenous way by the timing of systems of that type. And, most important, the calculus is sound and relatively complete (see (Broy, 2023)).

Following significant remarks by Bran Selic (Selic, 2020, private communication), we discuss some issues that have to do with the difference between the formal model and its practicality in terms of engineering practice.

1. For the formalism described in this paper, we must be aware of its limitations. In particular, we must be aware that these are models that are distinct from the reality that they intend to represent, and therefore, we must understand the nature of the relationship between a model of software intensive systems and its corresponding implementations in real life systems.
2. The abstract time model implies a global synchronous model of time, similar to that proposed by Berry (see (Berry, 2000)). On its own, this is not necessarily an issue – one can always choose a convenient time reference that can serve as a global reference point. However, there seems to be an implication that this view of time is perceived equally and simultaneously by all (physically distributed) subsystems under consideration. However, *from a practical viewpoint*, in a physically distributed system, each subsystem needs to have a local "replica" of the global time model, which means dependence on a local source of time. That local source of time could, of course, get out of step with the global time model, which means that there is no true global perception of time across all subsystems in a composition. Nevertheless, such a case can be modelled by the presented approach. This aspect is partially treated in (Broy, 2009).
3. A further major concern relates to the notion that each subsystem is *fully encapsulated*, such that its functional behavior is fully defined by its interface specifications. This effectively implies that every subsystem has its own dedicated computing engine. Given economic considerations (as found in, for instance, in primary motivations behind AUTOSAR by the need to share ECUs), as well as other practical factors, this is a significant step away from today's reality. Therefore, the key question is: can we change today's reality for the benefit of a more powerful methodology.

However, neither point 2 nor point 3 are arguments against our approach, which we interpret primarily as a method for validating system *specifications*, as opposed to a method for validating *implementations*.

It would be interesting and fruitful to initialize a research effort that would explore the relationship between formal system specifications and their practical implementations. How should we bridge the gap between these two, given the finiteness and imperfections of implementation technologies? The two are mutually interdependent: we may, for example, find that a perfectly valid formal system specification may need to be adjusted to account for technological and other feasibility limitations associated with implementations.[7]

CRediT authorship contribution statement

Manfred Broy: Conceptualization, Formal analysis, Investigation, Methodology, Visualization, Writing – original draft, Writing – review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

No data was used for the research described in the article.

Acknowledgement

It is a pleasure to thank Bran Selic for a substantial number of stimulating remarks.

References

Abadi, M., Lamport, L., 1993. Composing specifications. ACM Trans. Program. Lang. Syst. 15 (1), 73–132. January.

[5] According to (Selic, 2020, Herzberg and Broy, 2005) is missing one important aspect: The fact that communication resources at lower levels that realize higher-level channels, are invariably shared by multiple higher-level channels, which leads to the problem of unintended interactions due to shared resources (see (Selic, 2011)).

[6] However, it is not just memory that is shared. The CPU is also shared as well as other platform elements, such as busses, etc. The only time this can be avoided is if every concurrent subsystem was single threaded (i.e., no multitasking) and had its own dedicated computer. But, even if that was feasible, we would still run into problems of sharing network resources, unless we had dedicated communications resources for every possible interconnection between subsystems (Selic, 2020). Since we have to live with shared resources we need a theory of shared resources, in addition.

[7] Perhaps, Selic's "forgotten interfaces" (Selic, 2020) may provide a convenient starting point for such a study. Surely it is possible to formalize the impact that platforms can have through such interfaces? Perhaps the streams approach can be extended to go beyond data-based interactions?

de Bakker, J.W., van Breugel, F., 2000. From Banach to Milner: Metric Semantics for Second Order Communication and Concurrency. Proof, Language, and Interaction 99–132.
Banach, S., 1922. Sur les opérations dans les ensembles abstraits et leur application aux équations intégrales. Fundamenta Mathematicae 3, 133–181.
Berry, G., 2000. The foundations of Esterel. Proof, Language, and Interaction 425–454.
Berry, G., Boudol, G., 1992. The chemical abstract machine. Theoretical Computer Science 96 (1), 217–248.
Broy, M., Stølen, K., 2001. Specification and Development of Interactive Systems: Focus on Streams, Interfaces, and Refinement. Springer.
Broy, M., 2009. Relating Time and Causality in Interactive Distributed Systems". In Engineering Methods and Tools for Software Safety and Security. In: Broy, M., Sitou, W., Hoare, T. (Eds.), NATO Science for Peace and Security Systems, D:22, Information and Communication Security. IOS Press, pp. 75–130.
Broy, M., 2010. A Logical Basis for Component-Oriented Software and Systems Engineering. Comp. J. 53 (10), 1758–1782.
Broy, M., 2015. Rethinking Non-functional Software Requirements. IEEE Computer 48 (5), 96–99.
Broy, M., 2015. Computability and Realizability for Interactive Computations. Information and Computation 241, 277–301. April.
Broy, M., 2018. Theory and Methodology of Assumption/Commitment Based System Interface Specification and Architectural Contracts. Formal Methods in System Design. Springer Science & Business, Berlin, pp. 33–87, 52,1/2018.
Broy, M., 2023. Specification and Verification of Concurrent Systems by Causality and Realizability. Theoretical Computer Science 974. VolumeSeptember.
Broy, M., 2024. An Axiomatic Basis for System Design. Accepted for publication.
Broy, M., Rumpe, B., 2023. Development Use Cases for Semantics-Driven Modeling Languages. Communications of the ACM 66 (05), 62–71.
Chandy, K.M., Misra, J., 1988. Parallel Program Design: a Foundation. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.
de Alfaro, L., Henzinger, Th.A., 2001. Interface automata. In: Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering (ESEC/FSE-9). New York, NY, USA. ACM, pp. 109–120.
Dennis, J.B., 1974. First version of a data flow procedure language. Symposium on Programming 362–376.
Dijkstra, E.W., 1963. Over de sequentialiteit van procesbeschrijvingen (EWD-35).
Dijkstra, E.W., 1968. Cooperating sequential processes (EWD-123).
Geisberger, E., Broy, M., 2012. agendaCPS – Integrierte Forschungsagenda Cyber-Physical Systems (acatech STUDIE). Springer Verlag, Heidelberg u.a.
Gurevich, Y., 2000. Sequential Abstract State Machines capture Sequential Algorithms. ACM Transactions on Computational Logic 1 (1). July77-11.
Henzinger, Th.A., Jagadeesan, R., 1997. Robust Timed Automata. Proceedings of the International Workshop on Hybrid and Real-Time Systems (HART). In: Lecture Notes in Computer Science, 1201. Springer, pp. 331–345.
Henzinger, Th.A., Horowitz, B., Kirsch, Ch.M., 2003. Giotto: A Time-triggered Language for Embedded Programming. Proceedings of the IEEE 91, 84–99.
Herzberg, D., Broy, M., 2005. Modeling layered distributed communication systems. In: In Formal Aspects of Computing, 17. Springer, Berlin, pp. 1–18. May.
Hoare, C.A.R., 1985. Communicating Sequential Processes. Prentice-Hall.
Kahn, G., 1974. The Semantics of Simple Language for Parallel Programming. IFIP Congress 471–475.
Kahn, G., MacQueen, D.B., 1977. Coroutines and Networks of Parallel Processes. IFIP Congress 993–998.
Lamport, L., 1994. The Temporal Logic of Actions. ACM Toplas 16 (3), 872–923. May.
Lee, E.A. 2009. Computing Needs Time. Communication of the ACM, 52(5) 70–79.
Leeuwen, J.V. (Ed.), 1990. Handbook of Theoretical Computer Science (vol. A): Algorithms and Complexity. MIT Press.
Milner, R., 1980. A Calculus of Communicating Systems. Springer Verlag.
Milner, R., 1999. Communicating and Mobile Systems: the Pi-Calculus. Cambridge University Press.
Misra, J., 2005. Computation Orchestration. In: Broy, M., Grünbauer, J., Harel, D., Hoare, T. (Eds.), Engineering Theories of Software Intensive Systems. NATO Advanced Study Institute on Engineering Theories of Software Intensive Systems. NATO Science Series. 195. Marktoberdorf. Springer, Germany, pp. 285–330.
Neubeck, P., 2012. A Probabilitistic Theory of Interactive Systems. Technische Universität München. Dissertation.
Owicki, S., Gries, D., 1976. An Axiomatic Proof Technique for Parallel Programs I. Acta Informatica 6, 319–340.
Park, D.M.R., 1981. Concurrency and Automata on Infinite Sequences. Theoretical Computer Science 167–183.
Petri, C.A., 1962. Fundamentals of a Theory of Asynchronous Information Flow. In: IFIP Congress, pp. 386–390.
Proceedings of the First International Conference on Foundations of Software Science and Computation Structure Cardelli, L., Gordon, A.D., 1998. Mobile Ambients. In: Nivat, M. (Ed.), Lecture Notes in Computer Science. Springer-Verlag, pp. 140–155. March 28 - April 4.
Rich, E., 2008. Automata, Computability and complexity: Theory and Applications. Pearson Prentice Hall, Upper Saddle River, NJ.
Selic, B., 2011. A Short Catalogue of Abstraction Patterns for Model-Based Software Engineering. Int. J. Software and Informatics 5, 313–334.
Selic, B., 2020. Private Communication.
Selic, B., 2020. The Forgotten Interfaces: A Critique of Component-based Models of Computing and a Remedy. To appear in: Journal of Object Technology. Published by AITO. Association Internationale pour les Technologies Objets. http://www.jot.fm/.
Tripakis, S., Lickly, B., Henzinger, Th.A., Lee, E.A., 2011. A Theory of Synchronous Relational Interfaces. ACM Trans. Program. Lang. Syst. 33 (4). July14:1–14:41.

Manfred Broy's research is in software and systems engineering both in theoretical and practical aspects. This includes system models, specification and refinement of system and software components, specification techniques, development methods and verification. He is leading a research group working in a number of industrial projects that apply mathematically based techniques to combine practical approaches to software engineering with mathematical rigor. His main topics are requirements engineering, software and system architectures, componentware, software development processes, software evolution, and software quality. In his group the CASE tool AutoFocus was developed.

One of the main themes of Manfred Broy is the role of software in a networked world. As a member of acatech under his leadership the study Agenda Cyber-Physical Systems was created for the Federal Ministry of Research to comprehensively investigate the next stage of global networking through the combination of cyberspace and embedded systems in all their implications and potential.

From January 2016 till April 2019 Professor Broy was founding president of the Bavarian Center for Digitization. There he was working on the topics of digital transformation and digital innovation.