



# Validating IoT Devices with Rate-Based Session Types

GRANT IRACI, University at Buffalo, USA
CHENG-EN CHUANG, University at Buffalo, USA
RAYMOND HU, Queen Mary University of London, UK
LUKASZ ZIAREK, University at Buffalo, USA

We develop a session types based framework for implementing and validating rate-based message passing systems in Internet of Things (IoT) domains. To model the indefinite repetition present in many embedded and IoT systems, we introduce a timed process calculus with a periodic recursion primitive. This allows us to model *rate-based* computations and communications inherent to these application domains. We introduce a definition of rate based session types in a binary session types setting and a new compatibility relationship, which we call *rate compatibility*. Programs which type check enjoy the standard session types guarantees as well as rate error freedom — meaning processes which exchanges messages do so at the same *rate*. Rate compatibility is defined through a new notion of type expansion, a relation that allows communication between processes of differing periods by synthesizing and checking a common superperiod type. We prove type preservation and *rate error freedom* for our system, and show a decidable method for type checking based on computing superperiods for a collection of processes. We implement a prototype of our type system including rate compatibility via an embedding into the native type system of Rust. We apply this framework to a range of examples from our target domain such as Android software sensors, wearable devices, and sound processing.

CCS Concepts: • **Software and its engineering** → **Concurrent programming languages**; • **Theory of computation** → **Type structures**; • **Computer systems organization** → Real-time system specification.

Additional Key Words and Phrases: session types, type systems, rate-based systems

**ACM Reference Format:**
Grant Iraci, Cheng-En Chuang, Raymond Hu, and Lukasz Ziarek. 2023. Validating IoT Devices with Rate-Based Session Types. *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 278 (October 2023), 29 pages. https://doi.org/10.1145/3622854

## 1 INTRODUCTION

Many embedded [FreeRTOS 2003; QNX 2001] and Internet-of-Things (IoT) systems [Lin et al. 2017] are inherently concurrent message passing systems, featuring communication protocols with bidirectional, alternative and repeated interactions between their components. Sensitivity to communication rates is commonplace for systems in these domains. The correctness of these systems, thus, relies not only on compatibility of communication structures, but also the rate of message exchanges. This can be seen in IoT applications such as soft/virtual sensors [Madria et al. 2014; Maniscalco and Rizzo 2017; Martin et al. 2021], digital signal processing (DSP), wireless sensor networks [Yick et al. 2008] and sensor fusion [Brooks and Iyengar 1998], collaborative sensing [Tegen et al. 2019], biological software sensors [Wechselberger et al. 2013], and digital twins in cyber-physical systems (CPS) [Alam and El Saddik 2017; Tao et al. 2019].



Authors' addresses: Grant Iraci, grantira@buffalo.edu, University at Buffalo, USA; Cheng-En Chuang, chengenc@buffalo.edu, University at Buffalo, USA; Raymond Hu, r.hu@qmul.ac.uk, Queen Mary University of London, UK; Lukasz Ziarek, lziarek@buffalo.edu, University at Buffalo, USA.


2475-1421/2023/10-ART278
https://doi.org/10.1145/3622854

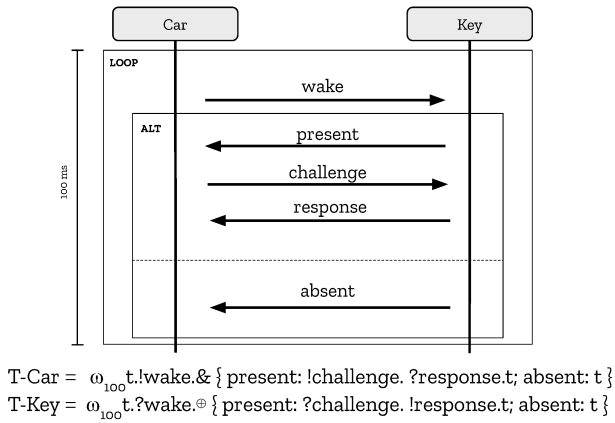


Fig. 1. Proximity Based Car Key Protocol

In DSP applications, differing rates of messages affect the correctness of calculations, causing, e.g., the incorrect frequency to be filtered or other audio distortions [Collins et al. 2018]. Whereas, in wireless protocols, like Bluetooth Low Energy [Bluetooth SIG 2023], adhering to rate is necessary for statically specified powersaving sleep intervals to be safe. Failure to wake up when expected, i.e., receiving or sending at an incorrect rate, causes the receiver to "miss" messages as Bluetooth LE advertisement packets are sent best-effort, meaning receipt is neither guaranteed nor acknowledged.

To better understand the structure of communication protocols in these domains, consider the concrete example given in Fig. 1, which depicts the protocol for an embedded "keyless entry" system in modern cars [Wouters et al. 2019]. The communication structure and rate are important for security and power efficiency. The car must periodically send wake-up message to probe the presence of the key. A branch in the protocol is critical because if the key is absent, the car should not send out cryptographic info. If a key is present, we can then proceed to a cryptographic challenge-response, where a correct answer unlocks the door. This protocol repeats indefinitely while the car engine is not on.

Reasoning about the correctness of rate-based systems, thus requires not only the correctness of computation but also the adherence to rate. Typically such systems are validated empirically, through testing, or via formal methods like model checking. To ease the validation of rate-based systems, we propose a novel type-based approach, which can verify protocol compliance and freedom from rate errors. Our starting point is session types.

Session types [Honda et al. 1998; Yoshida and Vasconcelos 2007] is a type systems approach to specifying and verifying protocols in interaction-oriented systems. The typing discipline ensures that all communication sessions are between processes with *compatible* [Gay and Hole 2005; Vallecillo et al. 2003] communication behaviours. In essence, at every point in a session, a sender must send one of the expected set of possible messages, and the receiver must be prepared to receive any of those messages. A well-typed system is statically guaranteed to be safe, e.g., a process will never receive a message of an unexpected type during execution. Formally, compatibility is *duality* [Honda 1993] of interactions up to session *subtyping* [Gay and Hole 2005], allowing senders to select a subset of the agreed message choices, and receivers to accept a superset. Standard session types can express and verify the communication *structure* of the Car Key protocol.

Unfortunately, this standard notion of compatibility in session types does not capture a critical aspect of the class of systems that we consider in this paper: the *communication rate* of each channel.

Intuitively, the notion of rate can be thought of as the ratio of messages actions performed to a given *period* of time. For example, in addition to the communication structure described above, the Car Key protocol specifies that both processes must complete their interactions from one Wake Up to the next Wake Up within a period of (e.g.) 100 ms – i.e., they must operate at the *same rate*. Reciprocation of message actions alone is not sufficient to guarantee correctness for such systems. It is incorrect to implement either process to operate at a slower rate than the other – e.g., by performing the reciprocal interactions as expected but over a longer period – the execution of systems composed from such rate-incompatible processes will result in errors such as rate mismatches, which standard session types do not verify. The impact of such errors can vary from garbled audio, to lost messages, to systems that produce mathematically incorrect results.

*Contributions.* This paper proposes a session types framework for implementing and validating *rate-based* message passing systems, common in embedded and IoT domains. The Car Key protocol is a simple example involving a single session between two processes operating over the same period, 100ms. We explore rate-based message passing systems that involve multiple sessions between numerous processes operating over *different* periods. To handle these more general applications, we introduce a novel notion of *rate compatibility* (RC) between periodic, communicating processes that may be operating over differing periods. Rate compatibility is decidable through a connection of session compatibility to the notion of *superperiods* (also known as hyper-periods) from real-time scheduling [Liu and Layland 1973]. In addition to standard communication safety, a well-typed system of rate-compatible processes is guaranteed free of *rate errors* – interacting processes safely exchange messages at the same rate.

The rest of the paper is organized as follows. Section 2 gives an overview of our framework and explores making session types rate aware. We examine the heart rate sensor functionality of the PineTime Smartwatch [Pine64 2019] as a concrete example. Section 3 provides an overview of our proposed system and motivates the design through examples. Section 4 formalizes our session calculus and operational semantics. To model the indefinite repetition present in embedded and IoT systems, we introduce a *timed* session calculus with a *periodic recursion* construct. This allows us to model the rate-based processing and communications inherent to our target application domain. Section 5 first defines *rate compatibility* (RC), a new rate-based session compatibility relation that determines when processes with differing periods but matching rates may safely interact. RC is defined by synthesizing and checking a common superperiod type, defined by a new expansion relation, that aligns communication rates between processes. We then present our type system for rate-based session types based on RC. We prove type preservation and *rate error* freedom for well-typed systems, and show a decidable method for type checking based on computing superperiods. Full proof details are in supplemental material. Section 6 presents an implementation of our type system and the RC relation via an embedding in the Rust type system, as part of a framework for safe construction of rate-based systems. We use our framework to implement a range of examples, including Android software sensors, the PineTime Smartwatch, a Bluetooth protocol, the keyless entry protocol given above, and sound processing. A working deployment of the heart rate sensor to FreeRTOS on micro-controller hardware validates the feasibility of applying our implementation to these resource constrained systems. This is available as a corresponding software artifact.

## 2 MOTIVATING EXAMPLE: THE PINETIME SMARTWATCH

Let us consider how we might type the communication protocol for the signal processing in a smartwatch heart rate sensor. The PineTime [Pine64 2019] is a smartwatch with open-source [1] firmware. A graphical representation of this pipeline flow is shown in Fig. 2. Each signal processing

[1]Available at https://github.com/InfiniTimeOrg/InfiniTime/blob/241d364/src/components/heartrate/Ppg.cpp

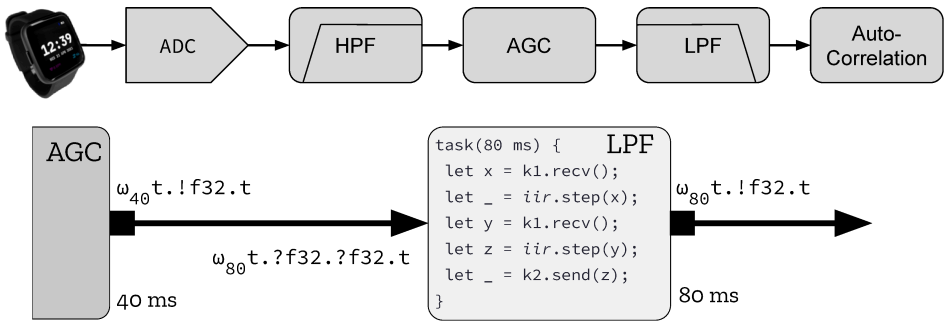


Fig. 2. A heart rate signal processing pipeline and Low Pass Filtering process

block is modeled as a communicating process. We look specifically at a communication based realization of the Low-Pass Filtering (LPF) stage. To illustrate the fundamentals of rate-based reasoning, we consider a variant that performs decimation, a common DSP technique in low-pass filters [Collins et al. 2018]. For every two values (called samples) taken in, the filter produces one output. This is decimation by two, meaning the LPF output rate is half of its input rate. An implementation of the LPF stage is shown in the bottom of Fig. 2.

From a communication view, the Low-Pass Filter process receives two samples as input from the Automatic Gain Control (AGC) stage along a channel $k_1$. From the perspective of the LPF stage, we could give a session type to this channel such as $k_1 : \mu t.?\texttt{f32}.?\texttt{f32}.t$ where `f32` is a 32-bit float. However, the AGC stage only processes one sample at a time. Thus from the AGC perspective, the channel would have a type $k_1 : \mu t.!\texttt{f32}.t$. A standard interpretation of recursive sessions as a fixed point considers these two types to be dual. Though it works for this example, we will see later that the introduction of rates can make this interpretation problematic.

In the systems we consider, communications across the channel happen at a fixed rate. Samples are passed from the AGC to the LPF block at a rate of 25 Hz, meaning one sample per 40 milliseconds. In the implementation, the sample rate is not exposed directly in the process. Instead, these signal processing blocks run at a fixed interval, known as a *period*. Here the AGC block runs for a period of 40 milliseconds and the LPF block for a period of 80 milliseconds. The ratio of the number of communications to the period forms the *rate*. Programmers must document the rate of each channel and manually verify that rates are properly aligned across processes. *Session types capture no information about the rates and are thus unable to help.*

In the case of our LPF process, there are actually two rates: an input rate and an output rate. The block runs at a fixed period, but recall, it performs one send on the output channel per two receives on the input. This decimation is a *rate transformation* that cuts the rate in half. A single process always has a single fixed period, but the rate may vary across channels. *The period is an aspect of the process, while the rate emerges from how a process uses a channel.* To capture information about the period, we replace the standard fixed point $\mu$ with a new recursive construct $\omega_n$ where $n$ represents the period at which the process repeats. Now our input type would be $k_1 : \omega_{80ms} t.?\texttt{f32}.?\texttt{f32}.t$.

Once we include the period in the type, a standard equi-recursive interpretation as a fixed point becomes problematic. Consider now a broken implementation of the AGC block, one in which the period was also 80 ms. The corresponding type $k_1 : \omega_{80ms} t.!\texttt{f32}.t$ appears similar, but captures different behavior. The period matches, but the body differs. Typical interpretations of recursive sessions would allow such a structural difference. However, the difference is more than

just structural, as they have completely different rates. The sample rate is linked to the mathematical correctness of the computation. In our Low-Pass Filter block, there is a set of constants which determine the frequency response of the filter. Using these constants with a wrong sample rate would render our watch unable to detect a heartbeat. The correct AGC type for our example would be $k_1 : \omega_{40ms} t.!\texttt{f32}.t$. These two views of the channel are shown in Fig. 2. Note that while the body is only half of the length, the period is too. Thus the rate is preserved. Clearly, both have a rate of one communication per 40 milliseconds. *Structurally however, they differ.*

Looking only at the infinite behavior as a fixed point is too permissive, incorrectly allowing sessions with mismatched rates. A more restricted view of duality that considered only exact matches in syntactic structure would be safe. When periods differ between processes, however, that view is overly restrictive and would immediately reject the program. The desired type system would allow differences in period, but still reject programs with incompatible rates. Reconciling these types is the key challenge in providing rate-based sessions.

## 3 OUR PROPOSED SYSTEM — COMBINING RATES AND SESSIONS

A type system for rate based sessions must be able to distinguish between processes with the same structure but different rates. Nevertheless, it must also allow processes with the same rates but different structures. Reconciling these observations requires reasoning about the behavior of types over time, not just their structure. To accomplish this our system will need a new notion of *compatibility*. It must be rate aware so it can type check programs where communicating processes have *different periods*, including periods that are *not multiples* of one another. Our type system relies on a definition of *expansion*, which allows us to reason about communications over time. We base compatibility on a *generalization* of expansion which allows us to reason about the type safety of programs which have communicating processes with periods that are not multiples of one another. Programs which type-check in our system will enjoy freedom from *rate errors*. We introduce these concepts intuitively through examples in this section, building on the PineTime example, and formalize them in the following sections.

*Rate Compatibility.* Traditionally, we say that the typings of two processes are compatible if and only if the types of channels in them are *dual*. Duality is a structural property of types. So as a first attempt, let us define duality of these periodic session types, referred to as $duality_\omega$, structurally. To be dual, we require that the annotations of $\omega$ match, enforcing that the tasks have the same period. In our smart watch example, we would say the dual of $k_1 : \omega_{40ms} t.?\texttt{f32}.?\texttt{f32}.t$ is $k_1 : \omega_{40ms} t.!\texttt{f32}.!\texttt{f32}.t$ This guarantees that the types have the same number of communications and same period, and thus the same rate.

This approach captures rate information but is overly restrictive. Consider our smartwatch again. The type in the AGC block was $k_1 : \omega_{40ms} t.!\texttt{f32}.t$. This is not the dual we were looking for. Our loop runs twice as often and performs half the communications per iteration. The sample rate is maintained, thus this is a safe replacement for our prior version. Unfortunately, the type is structurally different. Treating it as just an infinite stream of sends does not resolve our problem now that we have introduced rates. Consider the type $k_1 : \omega_{40ms} t.!\texttt{f32}.!\texttt{f32}.t$. This is not the same behavior. It has twice as many communications in the same period and thus twice the rate. If we were to implement the AGC this way, we would encounter a *rate error*. This hypothetical incorrect AGC block would be producing values faster than the LPF would consume them. If deployed, this could result in the AGC process stalling or unbounded channel growth, eventually resulting in an out-of-memory crash. Our heart rate sensor would fail.

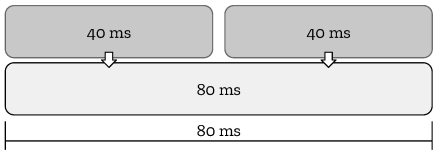


Fig. 3. Expanding to the superperiod

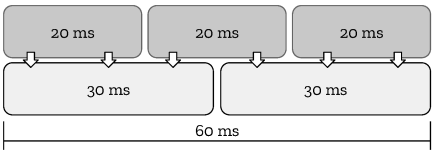


Fig. 4. Generalized Expansion

We can reason about indefinite systems by showing correctness over a time interval in which the entire system returns to an initial state. To do this we leverage the notion of *superperiod*[2] [Liu and Layland 1973], an amount of time where all processes execute at least one iteration and have all returned to their starting state. A superperiod is thus a common multiple of all periods in the system. Leveraging superperiods allows us to compare the behavior over the entire run of the process by only considering a finite window of time. Graphically, this is shown in Fig. 3 as a diagram of the communication pattern across our AGC to LPF channel as viewed over one superperiod.

*Expansion.* Crucial to our system is the observation that, from a communication perspective, statically repeating a task twice is indistinguishable from a process that runs half as often and does twice as many communications. We can thus transform the type for a given process to an equivalent one that executes once per superperiod. Since the superperiod is defined to be a multiple of the period, this amounts to repeating the type multiple times. We refer to this as *expansion*. This is an operation on types that mirrors the compiler optimization of loop unrolling. By leveraging expansion, we can generalize duality to capture a broader range of *rate compatible* processes.

This is formalized via an auxiliary expansion relation. We say a type $T'$ expands into $T$ if it can be transformed into $T$ by repetition of the loop body. We use expansion to define rate-based compatibility. To show two types are rate compatible (RC), $T_1 \bowtie T_2$, we need to find a shared type $T_3$ such that $T_1 <: T_3$ and $T_2 <: \overline{T_3}$. In this definition, $T_3$ has a surprisingly natural interpretation: the type representing the finite interval over which the two processes align. This alignment means we can then check for the presence of duality. This interval is the superperiod, and thus gives rise to a very simple approach to check the existence of such a $T_3$. We need only expand each process to the LCM and then check for duality. We examine these issues in depth via a process calculus that can track timing in Section 4 and the corresponding type system in Section 5.

*Catching Rate Based Errors.* Expansion exactly captures our desired rate-based compatibility. Consider the case where the AGC block in our example was implemented with the wrong period. A simple misreading of the specification could lead to a block with a period of 20 ms instead of the desired 40 ms. This corresponds to a rate double what our LPF block is expecting. Now the type would be $k_1 : \omega_{20ms} t.!\texttt{f32}.t$. The superperiod between this and the LPF block remains 80 ms. To get there, we expand the AGC type 4 times, obtaining $k_1 : \omega_{80ms} t.!\texttt{f32}.!\texttt{f32}.!\texttt{f32}.!\texttt{f32}.t$. After reaching the superperiod, we can structurally check for duality against the LPF type $k_1 : \omega_{80ms} t.?\texttt{f32}.?\texttt{f32}.t$. Crucially, because we do not consider all infinite streams the same now, these are not duals. More abstractly, we can see that once we have expanded so they have the same period, the only way they can have the same rate is to perform the same number of communications in opposite directions.

*Generalized Expansion.* While thus far the periods in question have been multiple, we can also consider cases where they are not. Given a pipeline in which samples have been batched differently, we might have a process with a period of 30 ms trying to communicate with a 20 ms process. We

[2]In real time systems, the superperiod is the *shortest* such time; in this paper, we say a superperiod for any multiple.

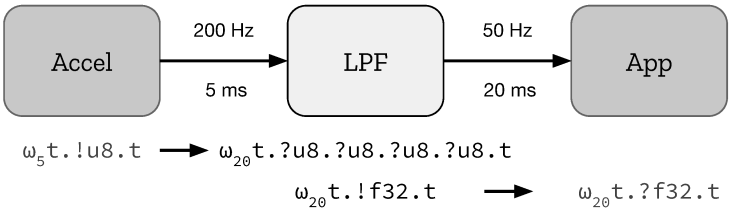


Fig. 5. The Android Gravity Sensor

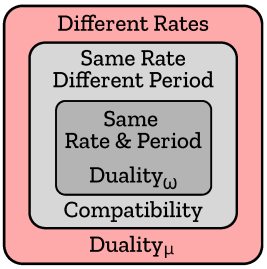


Fig. 6. Acceptable Sessions

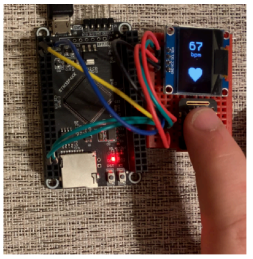

Fig. 7. The heart rate sensor in Rust with rate-aware session types

examine this timeline in Fig. 4. Now a single expansion of a process is not enough. Instead, we must expand both processes to achieve alignment. Here, we expand the 20 ms process to three times in length and the 30 ms process to twice. Both are thus aligned at 60 ms. Again we observe that the communications, averaging one per 10 ms, continue to match.

While the rates across a single channel must be the same, there is no guarantee that the rates on all channels used by a process are the same. The *periods* of the channel end points, however, are tied to the period of the corresponding process and thus match, even if they perform different numbers of communications. In DSP applications, this occurs when there is a *rate transformation*. Fig. 5 shows an overview of the Android gravity sensor, a software sensor built from multiple accelerometer readings. The sensor produces readings at a high rate, here 200 Hz. Due to both security and power considerations, access to the accelerometer and derived data is restricted to 50 Hz. We use a low-pass filter to perform another rate transformation — 4 samples get combined into 1. While the filter has a single period (20 ms) the rates differ between channels.

*Comparing Rate Compatibility and Duality.* We refer to duality in traditional timing-unaware session types as $duality_\mu$. A much more restricted notion of duality we call $duality_\omega$ is based on the syntactic structure of the types. It requires that the period annotations match, along with the exact communication structure of the body. In Fig. 6 we explore the relationship between these concepts. The outer region represents all sessions accepted by $duality_\mu$ where only the communications matter. Once we consider timing, a large portion of these pairings are unsafe and lead to rate errors. The outer region shaded in red encompasses all such sessions where there is a *rate mismatch*. This includes sessions such as the broken AGC type in Sec. 2 with types $\omega_{80ms}t.!\texttt{f32}.t$ and $\omega_{80ms}t.?\texttt{f32}.?\texttt{f32}.t$. The dark gray region in the center is the subset of sessions that have identical structure and whose rates match - exactly defining $duality_\omega$. This does not, however, encompass all safe session pairings. There are more pairings where the rates align, but the periods differ. Those sessions make up the light gray area surrounding it. The correct implementation of our AGC block falls here, pairing its view of the channel as $k_1 : \omega_{40ms}t.!\texttt{f32}.t$ with the $k_1 : \omega_{80ms}t.?\texttt{f32}.?\texttt{f32}.t$ view

taken by the LPF block. We need the *rate-compatibility* relation to generalize $duality_\omega$ and allow these pairings to type check.

*Formalization.* Capturing this behavior formally requires introducing new machinery for reasoning about periodic processes and their types. We first examine how to encode and reason about these processes in Section 4. A new process calculus variant allows us to define what a rate-error means in these systems. To prevent these errors, we then introduce a corresponding type system in Section 5. We prove these types statically prevent timing errors: Section 5.3. The techniques developed in the formal model then allow us to realize rate-based compatibility in a Rust library in Section 6. We revisit these examples and show how they can be translated into Rust and verified with the existing Rust type-checker. Our implementation leads to a working deployment of a heart rate sensor to FreeRTOS on a micro-controller, shown in Fig. 7.

## 4 A SESSION CALCULUS FOR RATE-BASED SYSTEMS

We formalize a process calculus for modeling *rate-based* concurrent message passing systems as found in embedded systems and IoT domains. Our calculus is based on the session calculus of Honda et al. [1998] to which we add two key features: *periodic recursion* and *timing*.

Periodic recursion $\texttt{prd}(n)\ X(\tilde{x}, \tilde{k}) = P\ \texttt{in}\ Q$ describes a recursive process $P$ (with "name" $X$) that is expected to be executed periodically with a *period* of $n$. This construct allows us to model the systems of (indefinitely) periodic processes in our target use cases, as demonstrated in Sec. 2. Conceptually, the period is the length of time between starting one iteration of $P$ and the next – i.e., it determines the frequency at which $P$ repeats. The next iteration will not be started until the current period has elapsed, but a safe execution requires all the work described in $P$, including communications dependent on *other* processes, being completed before then.

Our formalism abstracts from physical timing and details such as the absolute amount of time taken to perform actions by specific implementations or deployments. It focuses on safety in rate-based systems in terms of the *relative* timing between the processes of a system. Thus a period $n$ may be written to mean $n$ milliseconds (assuming milliseconds is a consistent unit across the system), but the crucial point is to capture the *ratio* between the periods of all such interacting process. In general, a period or time $n$ in our formalism denotes $n$ logical *timesteps*.

### 4.1 Syntax

Let $\ell, \ell', \ldots$ range over *labels*; $k, k', \ldots$ range over *channels*; $X, Y, \ldots$ range over *process variables*; $v, v', \ldots$ range over *values* (e.g., integers and booleans); $x, y, \ldots$ range over *variables* (for values); and $e, e', \ldots$ range over *expressions*, i.e., values, variables and computations (definitions omitted) that do not involve communications. The notation (e.g.) $\tilde{k}$ stands for a (possibly empty) sequence of distinct $k_i$; and (e.g.) $P[v/x]$ means substitute $v$ for $x$ in $P$. Fig. 8 defines *processes* $P, Q, R$.

*Periodic recursion* $\texttt{prd}(n)\ X(\tilde{x}, \tilde{k}) = P\ \texttt{in}\ Q$ is the major addition to the syntax. It associates process variable $X$ with the *static definition* (or simply, *definition*) $P$, that takes parameters $\tilde{x}$ and channels $\tilde{k}$, in the scope of the *run-time process* $Q$. It also specifies the *period* of $X$ is $n \in \mathbb{N}_1$. We may simply say process $X$ when referring to the $P$ associated with $X$. Periodic processes may terminate. We do not permit mutually recursive process definitions (restricted by typing, cf. (T-Prd) in Sec. 5.2). This simplification is because our formalism targets systems with periods that are statically specified and non-variant (i.e., constant) during execution.

The definition of a periodic recursion may contain *static process invocations* $X[\tilde{e}, \tilde{k}]$, representing a new iteration of process $X$ with arguments $\tilde{e}$ and channels $\tilde{k}$. A *run-time* process invocation $X^d[\tilde{e}, \tilde{k}]$ represents a pending invocation of $X$ during the execution of some process. The *deadline*

| | | | | | |
|---|---|---|---|---|---|
| $P, Q, R$ | ::= | process | | | |
| | $\mathbf{0}$ | inaction | \| | $\texttt{if}\ e\ \texttt{then}\ P\ \texttt{else}\ Q$ | conditional |
| \| | $k![e];\ P$ | data sending | \| | $P \mid Q$ | parallel composition |
| \| | $k?(x);\ P$ | data reception | \| | $(\nu k)\ P$ | channel scope |
| \| | $k \triangleleft \ell;\ P$ | branch select | \| | $\texttt{prd}(n)\ X(\tilde{x}, \tilde{k}) = P\ \texttt{in}\ Q$ | periodic recursion |
| \| | $k \triangleright \{\ell_i : P_i\}_{i \in 1..n}$ | branch offer | \| | $X[\tilde{e}, \tilde{k}]$ | process invoc. (static) |
| | | | \| | $X^d[\tilde{e}, \tilde{k}]$ | process invoc. (run-time) |

Fig. 8. Syntax of processes. $e$ ranges over values $v$ and expressions of base sorts (e.g., integers and booleans).

$d \in \mathbb{N}_0$ is the *global time* (explained in Sec. 4.2) at which the invocation is scheduled, i.e., a deadline by which all prior work (the proceeding process prefixes) is required to be completed. We often simply say *invocation* as short for process invocation. The definition of a periodic recursion must represent the static definition of a single task in a system. We thus prohibit run-time invocations and parallel processes from appearing in these definitions. Typing enforces this distinction and we formalize it as `static` $P$ in Sec. 5.2.

All other process terms are standard [Honda et al. 1998] with the following notes. For simplicity, values $v$ do not include channels $k$, i.e., we do not model session delegation, as it does not feature in any of our practical use cases. The meaning of communication rates in the presence of channel mobility is a topic of future work. We also omit the terms for session initiation (similarly to, e.g., Gay and Hole [2005]) as orthogonal to the present work; our calculus simply models established communication sessions directly (i.e., session initiation is implicit from session scoping).

Communication occurs as synchronizations between data sending and reception, and between branch select and offer. On synchronizing, data sending will evaluate the expression $e$ and proceed as $P$, while data reception will proceed as $P$ with the evaluation result substituted for bound variable $x$. Branch select will proceed as $P$, while branch offer will proceed with the $P_i$ corresponding to the selected $\ell$. The conditional, parallel composition and channel scoping terms are standard.

The sets of free variables $\texttt{fv}(P)$, channels $\texttt{fc}(P)$, and process variables $\texttt{fp}(P)$ of process $P$ are defined in the expected way. Fig. 13 (bottom) defines the *deadline on a channel* $k$ of $P$, $\texttt{dl}_k(P) \in \mathbb{N}_0 \cup \{\infty\}$, the earliest possible deadline $d$ on run-time invocations that occur within $P$ and pass the channel $k$. We define $\texttt{dl}_k(P)$ for a $k$ with no attached deadline in $P$ to be $\infty$ (representing that $P$ has unlimited time to complete its actions on $k$). Processes without free variables, channels, or process variables are called *programs*.

*Example 4.1.* Recall the Low Pass Filter (LPF) example from Sec. 2. Here, we consider a simplified variant with equal input and output rates.

$$\texttt{prd}(1)\ X(k_\text{i}, k_\text{o}) = k_\text{i}?(x);\ k_\text{o}![\texttt{hpf}(x)];\ X[k_\text{i}, k_\text{o}]\ \texttt{in}\ X^0[k_\text{i}, k_\text{o}]$$

This filter is recursively defined as a process $X$ with a period of *1 timestep*. (As discussed, the absolute value of a period is not important, but how it relates to the periods of processes it interacts with.) $X$ takes two channel parameters $k_\text{i}$ and $k_\text{o}$ . The definition of $X$ specifies that in each iteration (i.e., within each period of 1 timestep): it will input on $k_\text{i}$, do some local computations (the $\texttt{hpf}(x)$ expression) and output the result on $k_\text{o}$, before proceeding to the next iteration $X[k_\text{i}, k_\text{o}]$ when the period ends. The process is bootstrapped by $X^0[k_\text{i}, k_\text{o}]$, i.e., at system time 0 (the initial timestep in a system execution), invoke $X$ to run on (pre-established) channels $k_\text{i}$ and $k_\text{o}$.

$$P \mid \mathbf{0} \equiv P \qquad P \equiv Q \;\text{ if } P \equiv_\alpha Q \qquad (\nu k)\,\mathbf{0} \equiv \mathbf{0}$$
$$P \mid Q \equiv Q \mid P \qquad P \mid (Q \mid R) \equiv (P \mid Q) \mid R \qquad (\nu k)\,P \mid Q \equiv (\nu k)(P \mid Q) \;\text{ if } k \notin \mathtt{fc}(Q)$$
$$\mathbf{0} \equiv \mathtt{prd}(n)\,X\langle\tilde{e},\tilde{k}\rangle = P \;\mathtt{in}\; \mathbf{0} \qquad \big(\mathtt{prd}(n)\,X\langle\tilde{e},\tilde{k}\rangle = P \;\mathtt{in}\; Q\big) \mid R \equiv \mathtt{prd}(n)\,X\langle\tilde{e},\tilde{k}\rangle = P \;\mathtt{in}\; (Q \mid R) \;\text{ if } X \notin \mathtt{fp}(R)$$

Fig. 9. Structural congruence $P \equiv Q$

### 4.2 Operational Semantics

*Modeling time.* We model the advance of time *globally* as a series of logical timesteps, formalized as a monotonically increasing, system-wide counter $c \in \mathbb{N}_0$. Our formalism stratifies system execution ($\longrightarrow$) into (a) steps that perform communication actions and local computations but do *not* advance time ($\longrightarrow_\bullet$), and (b) steps that increment global time. Generally speaking, system execution proceeds by alternating between (a) some number (possibly 0) of $\longrightarrow_\bullet$ steps and (b) one $\longrightarrow_\circ$ step. Our model is designed to exclude scenarios where processes *are* able to perform work, but instead idle and trivially miss deadlines. Specifically, it does not allow a $\longrightarrow_\circ$ step (i.e., global time is not allowed to advance) unless all processes have met all their deadlines up to the current system time (cf. the (Delay) rule); our model treats a system in which a process cannot meet a deadline as stuck. In Sec. 5.3, we formalize systems stuck in this manner as *rate errors*.

As we are interested in the relative rate compatibility between processes, not the absolute time taken to perform communications or local computations, our model assumes that periods span sufficient physical time for (safely rate compatible) processes to perform all the (inter)actions in each step. If a system is *not* safely rate compatible in this model, then its design is inherently flawed regardless of how much physical time a concrete implementation may allocate to the conceptual timestep. In practice, the specific duration of timesteps/periods is often part of the protocol specification. In the case of our Low Pass Filter example, these would be derived from the sample rate of the hardware sensors and must agree with the formula for `hpf`.

*State reduction.* A *state* $\sigma$ is a pair $\langle P, c\rangle$ of a process $P$ and a *system time* (or simply, *time*) $c \in \mathbb{N}_0$. An *initial state* is a state $\langle P, 0\rangle$ where $P$ is a program. *State reduction* $\sigma \longrightarrow \sigma'$ is the union of relations $\sigma \longrightarrow_\bullet \sigma'$ and $\sigma \longrightarrow_\circ \sigma'$ given by the rules in Fig. 10. As noted above, reduction is defined in two parts: steps that perform computation ($\longrightarrow_\bullet$), and steps that advance time ($\longrightarrow_\circ$).

We first explain computations $\longrightarrow_\bullet$. The reduction of process $P$ is mostly standard [Honda et al. 1998], leaving the time $t$ unchanged. Rule (Com) is for synchronous communication of a value between an input process and an output process on a channel $k$. The notation $e \downarrow v$ means evaluate $e$ to a value $v$. Similarly, rule (Bra) is for communication of a branch label. Rule (IfT) is for the true case of a conditional; we omit the corresponding (IfE) rule. Rules (Par), (Chan) and (Str) are standard context rules for parallel composition, channel hiding and structural congruence. Structural congruence for processes (Fig. 9) is also standard [Honda et al. 1998]. The notation $\equiv_\alpha$ means $\alpha$-equivalence. The structural congruence rule for periodic recursion (for syntactically shuffling definitions into redex position) is analogous to that of standard recursive process definitions. Structural congruence is lifted to states by: $\langle P_1, c_1\rangle \equiv \langle P_2, c_2\rangle$ if and only if $P_1 \equiv P_2$ and $c_1 = c_2$.

We now explain the behavior of periodic recursion. Rule (Inv) models the repetition of a periodic process at the desired intervals. The form $X^d[\tilde{e},\tilde{k}]$ denotes a running process that has completed the work (i.e., prior prefixes) of its current iteration and is waiting to commence the next at deadline $d$. When the current time reaches the deadline, the invocation can be replaced by the definition of $X$ with arguments $\tilde{v}$ respectively given by evaluating $\tilde{e}$ and channels $\tilde{k}$. Occurrences of static invocations are replaced by (future) run-time invocations with deadline $d{+}n$, i.e., period $n$ timesteps

$$\text{(Com)}\ \frac{e \downarrow v}{\langle k![e];\ P \mid k?(x);\ Q, c\rangle \rightarrow_\bullet \langle P \mid Q[v/x], c\rangle} \qquad \text{(Bra)}\ \frac{1 \le j \le n}{\langle k \triangleright \{\ell_i : P_i\}_{i\in 1..n} \mid k \triangleleft \ell_j;\ Q, c\rangle \rightarrow_\bullet \langle P_j \mid Q, c\rangle}$$

$$\text{(IfT)}\ \frac{e \downarrow \texttt{true}}{\langle \texttt{if}\ e\ \texttt{then}\ P\ \texttt{else}\ Q, c\rangle \rightarrow_\bullet \langle P, c\rangle} \qquad \text{(Par)}\ \frac{\langle P, c\rangle \rightarrow_\bullet \langle P', c\rangle}{\langle P \mid Q, c\rangle \rightarrow_\bullet \langle P' \mid Q, c\rangle} \qquad \text{(Chan)}\ \frac{\langle P, c\rangle \rightarrow_\bullet \langle P', c\rangle}{\langle (\nu k)\, P, c\rangle \rightarrow_\bullet \langle (\nu k)\, P', c\rangle}$$

$$\text{(Str)}\ \frac{P \equiv P' \quad \langle P', c\rangle \rightarrow_\bullet \langle Q', c\rangle \quad Q' \equiv Q}{\langle P, c\rangle \rightarrow_\bullet \langle Q, c\rangle} \qquad \text{(Prd)}\ \frac{\langle Q, c\rangle \rightarrow_\bullet \langle Q', c\rangle}{\langle \texttt{prd}(n)\, X(\tilde{x}, \tilde{k}) = P\ \texttt{in}\ Q, c\rangle \rightarrow_\bullet \langle \texttt{prd}(n)\, X(\tilde{x}, \tilde{k}) = P\ \texttt{in}\ Q', c\rangle}$$

$$\text{(Inv)}\ \frac{\tilde{e} \downarrow \tilde{v} \qquad d = c}{\langle \texttt{prd}(n)\, X(\tilde{x}, \tilde{k}) = P\ \texttt{in}\ X^d[\tilde{e}, \tilde{k}], c\rangle \rightarrow_\bullet \langle \texttt{prd}(n)\, X(\tilde{x}, \tilde{k}) = P\ \texttt{in}\ P[\tilde{v}/\tilde{x}][X^{d+n}/X], c\rangle}$$

---

$$\text{(Delay)}\ \frac{\forall k.\, \texttt{dl}_k(P) > c}{\langle P, c\rangle \rightarrow_\circ \langle P, c{+}1\rangle}$$

Fig. 10. State reduction: (top) performing computations $\sigma \rightarrow_\bullet \sigma'$, and (bottom) advancing time $\sigma \rightarrow_\circ \sigma'$.

ahead of the current deadline $d$. Rule (Prd) is a context rule analogous to that for standard recursive process definitions.

Finally, $\rightarrow_\circ$ has a single rule (Delay). It allows system time to be advanced globally by one step provided all deadlines that occur in the running system are greater than the current time. As mentioned, our model prevents degenerate scenarios where a process can perform communications but instead trivially misses a deadline by idling. In our model, an invocation with deadline $d$ but which cannot be reduced within timestep $d$ (e.g., due to being blocked under other non-reducible communication prefixes) causes state reduction to become stuck overall. Such states can arise in logically incorrect systems due to rate errors, as we formalize and use to prove *rate safety* in Sec. 5.3.

*Example 4.2.* Let initial $\sigma_0 = \langle (\nu k_i)(\nu k_o)(P \mid Q), 0\rangle$ where $P$ is the process from Ex. 4.1, and $Q$ is $\texttt{prd}(2)\, Y(x, k_\text{i}, k_\text{o}) = k_\text{i}![x];\ k_\text{o}?(y);\ k_\text{i}![y];\ k_\text{o}?(z);\ Y[z, k_\text{i}, k_\text{o}]\ \texttt{in}\ Y^0[42, k_\text{i}, k_\text{o}]$. $\sigma_0$ may reduce by:

$$\sigma_0 \xrightarrow{\ldots\text{Inv}}_\bullet \xrightarrow{\ldots\text{Inv}}_\bullet \xrightarrow{\ldots\text{Com}}_\bullet \xrightarrow{\ldots\text{Com}}_\bullet \xrightarrow{\text{Delay}}_\circ \sigma_1 \xrightarrow{\ldots\text{Inv}}_\bullet \xrightarrow{\ldots\text{Com}}_\bullet \xrightarrow{\ldots\text{Com}}_\bullet \xrightarrow{\text{Delay}}_\circ \sigma_2 \xrightarrow{\ldots\text{Inv}}_\bullet \xrightarrow{\ldots\text{Inv}}_\bullet \quad \ldots$$

The numerical subscript of the $\sigma$ states indicates the timestep in which the reductions occur. The (e.g.) ...Inv label indicates the leaf rule of the reduction step is (Inv), enabled by context rules (Chan), (Par), (Prd), etc. In timestep 0, the two processes are invoked, allowing the first pair of communications to be performed, leaving (the run-time process within) $P$ at the end of its first iteration. The system time is incremented, and $P$ is invoked again, allowing the second pair of communications. Finally, both $P$ and $Q$ are at the end of their current iterations, and the periodic behavior of the system as a whole (cf. the *superperiod*, Sec. 5.1) may repeat.

## 5 RATE-BASED SESSION TYPE SYSTEM

We develop a *rate-based* session type system for our timed session calculus. Our type system is based on that of Honda et al. [1998], employing a linear context for typing process behaviors in terms of channel usages. To reason about safety for rate-based systems, we introduce: (a) *periodic recursive types* (or simply, *periodic types*) for communication behaviors featuring periodic recursion;

$$T ::= !S.T \mid ?S.T \mid \oplus\{\ell_i : T_i\}_{i\in 1..n} \mid \&\{\ell_i : T_i\}_{i\in 1..n} \mid \omega_n t.T \mid \omega_n^d t.T \mid t \mid \mathbf{end} \qquad S ::= \texttt{int} \mid \texttt{bool} \mid \ldots$$

$$\overline{!S.T} = ?S.\overline{T} \qquad \overline{?S.T} = !S.\overline{T} \qquad \overline{\oplus\{\ell_i : T_i\}_{i\in 1..n}} = \&\{\ell_i : \overline{T_i}\}_{i\in 1..n} \qquad \overline{\&\{\ell_i : T_i\}_{i\in 1..n}} = \oplus\{\ell_i : \overline{T_i}\}_{i\in 1..n}$$

$$\overline{\omega_n t.T} = \omega_n t.\overline{T} \qquad \overline{\omega_n^d t.T} = \omega_n^d t.\overline{T} \qquad \overline{t} = t \qquad \overline{\mathbf{end}} = \mathbf{end}$$

Fig. 11. (top) Syntax of binary session types with periodic recursive types; (bottom) duality $\overline{T} = T'$.

(b) *timing-aware* mechanisms for unfolding and *expansion* of periodic types; and (c) a notion of *rate compatibility* to determine when periodic behaviors may be safely composed.

Analogously to the process syntax, our type system integrates two forms of periodic types. One form $\omega_n t.T$ is used to type the static definition of a periodic process with period $n$. The other $\omega_n^d t.T$ carries the constraint of deadline $d$ to type run-time instances of such processes, denoting the timestep by which the process must complete its current iteration and commence the next. In a rate error state, the constraints imposed by the deadlines are unsatisfiable. These processes get stuck when they fail to meet a deadline. Rate safety means a well-typed system is statically guaranteed not to reduce to a rate error. This requires tracking not only the rates of communications but also a phasing constraint. Having the deadline appear in the type of states at run-time allows us to rule out problematic initial states where the execution of processes is misaligned. In the simplest case, two processes may communicate at the same rate, but one may begin at $d_1 = 2$, at which point its partner's deadline $d_2 = 1$ has already passed. All of the systems we have considered start uniformly with all deadlines at 0, avoiding this issue. Tracking the deadlines explicitly in the types of run-time invocations allows us to prove this alignment is preserved throughout execution.

Key to safety is our formulation of rate compatibility $T_1 \bowtie T_2$ between communication behaviors. An intuitive observation is that periodic processes may safely interact despite operating over *heterogeneous* periods provided they are structurally compatible in such a way that their communication rates *align*. We must forgo traditional treatments [Gay and Hole 2005; Honda et al. 1998] of recursive types, subtyping and process subsumption, that seek to freely equate all syntactic (un)foldings of a given recursive type or process. Instead, we formulate notions of expansion and compatibility for periodic types that are additionally sensitive to communication rates. With our types that capture periods and deadlines in their behavioral characterization of periodic processes, this allows us to reason about *rate safety* for rate-based systems.

### 5.1 Rate-Based Session Types and Rate Compatibility

Fig. 11 (top) defines our syntax of *session types* for rate-based systems. $!S.T$ and $?S.T$ are for sending and receiving data of sort $S$. We assume a set of base sorts for values and expressions. $\oplus\{\ell_i : T_i\}_{i\in 1..n}$ and $\&\{\ell_i : T_i\}_{i\in 1..n}$ are for branch select and offer. Let $t, t', \ldots$ range over *recursion variables* (distinct from *time* $t$ in Sec. 4 based on context). As discussed above, $\omega_n t.T$ and $\omega_n^d t.T$ are *periodic recursive types* (or simply, *periodic types*), binding the recursion variable $t$ within $T$. As in the process syntax, $n \in \mathbb{N}_1$ is the *period* and $d \in \mathbb{N}_1$ is the *deadline*. Session termination is denoted **end**. We often say *type* as short for session type.

We assume periodic types are contractive. Our system imposes some further restrictions on periodic types and the definitions of periodic recursions. We require that periodic types are not nested. This means in any periodic type $\omega_n t.T$ (or $\omega_n^d t.T$), the body type $T$ may not contain any occurrence of another periodic type. E.g., in our system $\omega_2 t.!\texttt{int}.t$ is permitted, whereas

$\omega_2 t.\omega_2 t'.!\texttt{int}.t'$ and $\omega_2 t.\omega_3 t'.\&\{\ell_1 : !\texttt{int}.t,\ \ell_2 : !\texttt{int}.t'\}$ are not. Our type system implicitly disallows these types by forbidding a periodic recursion definition from using any other process variable.[3]

Looking ahead (cf. Sec. 5.2), this restriction is enforced by typing rule (T-Prd) (by restricting the process context $\Theta$ when typing the definition $P$). It corresponds with a common restriction in our target application domains that periodic tasks cannot dynamically change their periods during run-time. In embedded system design, allowing dynamically changing periods precludes a static schedule and hampers static guarantees. In our system, allowing general mutually recursive tasks would correspond to allowing dynamically changing periods. In all of the example systems we explore, the period of each task is explicitly fixed and statically known. We note the rate of communications on a channel can still change as our system allows different branches of choice constructs to have differing numbers of communications.

Our system also requires that the definition of a periodic recursion corresponds to a single static definition, as formalized by $\texttt{static}\ P$ (Fig. 14). Notably, a definition cannot contain a parallel composition: this restriction mirrors our target domains where tasks are started at system boot and not created dynamically. A definition cannot contain a nested definition nor *run*-time process invocations $X^d[\tilde{k}, \tilde{e}]$. $\texttt{static}\ P$ on periodic recursion definitions is enforced by typing rule (T-Prd).

*Preliminaries.* We define the key concepts leading up to *rate compatibility* (RC), before presenting state typing based on RC in Sec. 5.2.

First, Fig. 11 (bottom) defines the *duality* relation on types, which represents reciprocation of communication structures. All cases are standard [Honda et al. 1998]; the cases for periodic types are as for standard recursive types $\mu t.T$, yielding the reciprocal structure over the same period $n$.

Next, we define the one-iteration *unfolding* of a type $T$:

$$\texttt{unfold}(\omega_n t.T) = T[\omega_n t.T/t] \qquad \texttt{unfold}(\omega_n^d t.T) = T[\omega_n^{d+n} t.T/t]$$

and $\texttt{unfold}(T) = T$ for all other cases of $T$. The first and third cases are as for standard recursive types. The second case is key: conceptually, $\texttt{unfold}(\omega_n^d t.T)$ represents performing *one additional iteration* of a running periodic process. The *deadlines* of all the *subsequent* invocations are accordingly advanced by one period to $d{+}n$.

Lastly, we express a new notion of type expansion in the form of a subtyping relation $T_1 <: T_2$, signifying that $T_2$ is an *expansion* of $T_1$; i.e., the (shorter) type $T_1$ can be expanded to the (longer) type $T_2$. Expansion represents the *scaling up* of a (periodic) behavior, in terms of its communication structure, over some number of periods. We use an auxiliary function:

$$\texttt{exp}_t(T, 1) = T \qquad \texttt{exp}_t(T, m) = \texttt{exp}_t(T, m{-}1)[T/t] \quad \text{for } m > 1$$

In the context of a periodic recursion with recursion variable $t$, it expands type $T$ by the factor $m$. It is used in the expansion rules to scale the period (and deadline) accordingly. By this definition, an expansion factor of 1 is identity, whereas a factor of, e.g., 3 yields a type that performs triple the number of communications over a three-times-longer period. Expansion thus crucially preserves the *rate* of communication between behaviours, which lies at the heart of rate compatibility (RC).

Fig. 12 (top) defines the expansion relation $\Sigma \vdash T_1 <: T_2$. The set of assumptions $\Sigma$ is standard to subtyping relations over iso-recursive types [Pierce 2002] and tracks assumptions of recursion variables already considered. The notation $T_1 <: T_2$ is a shorthand when $\Sigma$ is empty. Expansion is reflexive and transitive. It would be possible to formulate a symmetric version of expansion, making it an equivalence. We define expansion in this antisymmetric form because the (partial) ordering yields a decidable algorithm for RC, as we explain below. Expansion includes unfolding, as necessary for RC to relate running processes that are at staggered period phases due to *asynchrony*

[3] In the supplemental material, we give an explicit syntactic predicate to clarify the types permitted by this condition.

$$\frac{\Sigma \vdash T_1 <: T_2}{\Sigma \vdash !S.T_1 <: !S.T_2} \qquad \frac{\Sigma \vdash T_1 <: T_2}{\Sigma \vdash ?S.T_1 <: ?S.T_2} \qquad \frac{T_1 <: T_2 \in \Sigma}{\Sigma \vdash T_1 <: T_2} \qquad \frac{}{\Sigma \vdash \mathbf{end} <: \mathbf{end}}$$

$$\frac{\Sigma \vdash T_i <: T_i'}{\Sigma \vdash \oplus\{\ell_i : T_i\}_{i\in 1..n} <: \oplus\{\ell_i : T_i'\}_{i\in 1..n}} \qquad \frac{\Sigma \vdash T_i <: T_i'}{\Sigma \vdash \&\{\ell_i : T_i\}_{i\in 1..n} <: \&\{\ell_i : T_i'\}_{i\in 1..n}}$$

$$\frac{\Sigma, t_1 <: t_2 \vdash T_1 <: T_2}{\Sigma \vdash \omega_n t_1.T_1 <: \omega_n t_2.T_2} \qquad \frac{\Sigma, t_1 <: t_2 \vdash T_1 <: T_2}{\Sigma \vdash \omega_n^d t_1.T_1 <: \omega_n^d t_2.T_2} \qquad \frac{}{\Sigma \vdash \omega_n^d t.T <: \mathtt{unfold}(\omega_n^d t.T)}$$

$$\frac{\Sigma \vdash \omega_{n_1\times m} t_1.\mathsf{exp}_{t_1}(T_1, m) <: \omega_{n_2} t_2.T_2}{\Sigma \vdash \omega_{n_1} t_1.T_1 <: \omega_{n_2} t_2.T_2} \qquad \frac{\Sigma \vdash \omega_{n_1\times m}^{d'} t_1.\mathsf{exp}_{t_1}(T_1, m) <: \omega_{n_2}^{d_2} t_2.T_2 \quad d' = d_1 + n_1(m-1)}{\Sigma \vdash \omega_{n_1}^{d_1} t_1.T_1 <: \omega_{n_2}^{d_2} t_2.T_2}$$

---

$$\text{(RC)}\ \frac{T_1 <: T_3 \qquad T_2 <: \overline{T_3}}{T_1 \bowtie T_2}$$

Fig. 12. (top) Expansion $\Sigma \vdash T <: T'$; (bottom) Rate Compatibility $T \bowtie T'$.

$$\text{Let} \qquad \dagger ::= ! \mid ? \qquad \ddagger ::= \oplus \mid \& \qquad \omega_n^\star t.T ::= \omega_n t.T \mid \omega_n^d t.T$$

$$\mathtt{rl}(\dagger S.T) = 1 + \mathtt{rl}(T) \quad \mathtt{rl}(\ddagger\{\ell_i : T_i\}_{i\in 1..n}) = 1 + \max_{i\in 1..n}(\mathtt{rl}(T_i)) \quad \mathtt{rl}(\omega_n^d t.T) = \mathtt{rl}(t) = \mathtt{rl}(\mathbf{end}) = 0$$

$$\mathtt{bl}(\dagger S.T) = \mathtt{bl}(T) \quad \mathtt{bl}(\ddagger\{\ell_i : T_i\}) = \max_{i\in 1..n}(\mathtt{bl}(T_i)) \quad \mathtt{bl}(\omega_n^\star t.T) = \mathtt{rl}(T) \quad \mathtt{bl}(t) = 0$$

$$\mathsf{pd}(\dagger S.T) = \mathsf{pd}(T) \qquad \mathsf{pd}(\ddagger\{\ell_i : T_i\}) = \max_{i\in 1..n}(\mathsf{pd}(T_i)) \qquad \mathsf{pd}(\omega_n^\star t.T) = n$$

---

$$\mathtt{dl}(\dagger S.T) = \mathtt{dl}(T) \quad \mathtt{dl}(\ddagger\{\ell_i : T_i\}_{i\in 1..n}) = min_{i\in 1..n}(\mathtt{dl}(T_i)) \quad \mathtt{dl}(\omega_n^d t.T) = d \quad \mathtt{dl}(\mathbf{end}) = \infty$$

$$\mathtt{dl}_k(\mathbf{0}) = \infty \qquad \mathtt{dl}_k(k'![e]; P) = \mathtt{dl}_k(P) \qquad \mathtt{dl}_k(k'?(x); P) = \mathtt{dl}_k(P) \qquad \mathtt{dl}_k(k' \triangleleft \ell; P) = \mathtt{dl}_k(P)$$

$$\mathtt{dl}_k(k' \triangleright \{\ell_i : P_i\}_{i\in 1..n}) = min_{i\in 1..n}(\mathtt{dl}_k(P_i)) \qquad \mathtt{dl}_k(\mathtt{if}\ e\ \mathtt{then}\ P\ \mathtt{else}\ Q) = min(\mathtt{dl}_k(P), \mathtt{dl}_k(Q))$$

$$\mathtt{dl}_k(P \mid Q) = min(\mathtt{dl}_k(P), \mathtt{dl}_k(Q)) \qquad \mathtt{dl}_k((\nu k')P) = \mathtt{dl}_k(P) \ \text{ if } k \neq k', \text{otherwise } \infty$$

$$\mathtt{dl}_k(\mathtt{prd}(n)\, X(\tilde{x}, \tilde{k}) = P\ \mathtt{in}\ Q) = \mathtt{dl}_k(Q) \qquad \mathtt{dl}_k(X^d[\tilde{e}, \tilde{k}]) = d \ \text{ if } k \in \tilde{k}, \text{otherwise } \infty$$

Fig. 13. Definitions of (top) remainder length $\mathtt{rl}(T) \in \mathbb{N}_0$, body length $\mathtt{bl}(T) \in \mathbb{N}_0$, and type period $\mathsf{pd}(T) \in \mathbb{N}_1$; (bottom) deadline of types $\mathtt{dl}(T)$, and deadline of processes $\mathtt{dl}_k(P)$ across a channel

.

in execution – process invocation (i.e., unfolding the next iteration for a new period) is a *local* action performed independently by each process.

In summary, expansion paves the way to addressing the essential challenge of safely relating periodic behaviours with *different* periods and deadlines, arising due to both systems with heterogenous periods and the technicalities of asynchronous invocations.

*Communication rates and Rate Compatibility.* We are now ready to formalise the notions of communication rate and rate compatibility. Recall that nesting of periodic types is not permitted.

Fig. 13 (top) defines the *remainder length* $\mathtt{rl}(T)$, *body length* $\mathtt{bl}(T)$, and *period* $\mathsf{pd}(T)$ of a type $T$. Here, $\max_{i\in 1..n}$ is defined as the maximum of the cases that are defined, provided there is at least one such case. $\mathtt{rl}(T)$ signifies the number of communication actions remaining to complete an unfolded iteration of a periodic type, i.e., the prefixes "outside" of the periodic type. For branch

types, it takes that of the longest case. Traditional communication safety ensures that the labels presented in a branch remain compatible. By choosing the maximum length branch, we obtain a single measure of length that is consistently preserved by expansion. $\mathtt{bl}(T)$ correspondingly signifies the number of communications within the body of the periodic type. Choosing the longest branch when determining both $\mathtt{bl}(T)$ and $\mathtt{rl}(T)$ ensures $\mathtt{rl}(\mathtt{unfold}(\omega_n^d t.T)) = \mathtt{bl}(\omega_n^d t.T)$ We use this property to count the number of unfoldings performed in the proof of Theorem 5.9. Note that unfolding preserves $\mathtt{bl}$, and expansion preserves $\mathtt{rl}$. The period of a periodic type $T$ is $\mathtt{pd}(T)$. Fig. 13 (bottom) defines the *deadline* of a type $T$, $\mathtt{dl}(T) \in \mathbb{N}_0 \cup \{\infty\}$. Similarly to $\mathtt{dl}_k(P)$, this function finds the earliest possible deadline $d$ on periodic types that occur within $T$.

We then define the *communication rate* (or simply, *rate*) of a type $T$ as the ratio of its body length to its period, $R(T) = \frac{\mathtt{bl}(T)}{\mathtt{pd}(T)}$. As periods are positive numbers, the rate $R(T)$ is always a positive rational number if defined. It is always, and only, defined for a type that contains a periodic type. Rate is preserved by *both* unfolding and expansion.

We can now define *rate compatibility* (RC) in terms of expansion. Fig. 12 (bottom) defines the RC relation between types $T_1 \bowtie T_2$, which says $T_1$ is rate compatible with $T_2$ if there exists a $T_3$ that is an expansion of $T_1$ such that the dual $\overline{T_3}$ is an expansion of $T_2$. Intuitively, two periodic behaviours are safely compatible if there exists some common protocol to which both behaviours can be scaled up (expansion), preserving their communication *rates*, whereupon they are in direct agreement on the communication *structure* (duality). RC is symmetric, and subsumes duality. RC implies *rate equivalence*, i.e., $T_1 \bowtie T_2$ implies $R(T_1) = R(T_2)$.

We formulate RC in the above way for two related reasons. One is to yield a practical algorithm for checking RC, which we implement in our Rust framework. We discuss the algorithm in Sec. 6.1.1, but we note here that it relies on the fact that RC can expand and unfold in one direction only, as opposed to standard equirecursive session subtyping [Gay and Hole 2005] that can freely fold and unfold recursives types in both directions.

The other reason is expressiveness. Traditional compatibility [Vallecillo et al. 2006] between session types $T_1$ and $T_2$ directly relates $T_1 \leq \overline{T_2}$, where $\leq$ means the standard session subtyping [Gay and Hole 2005]. Since our system permits expansion and unfolding in one direction only, we define RC between $T_1$ and $T_2$ via duality of a common expansion $T_3$. This allows RC to relate a larger set of safely compatible periodic behaviours where both types must be expanded to a common *superperiod* before duality holds (Sec. 6.1.1).[4]

### 5.2 Type System

Following Honda et al. [1998], a *sorting* $\Gamma, \Gamma', \ldots$ is a map from names and variables to sorts; a *typing* $\Delta, \Delta', \ldots$ is a map from channels to types or $\bot$; and a *basis* $\Theta, \Theta', \ldots$ is a map from process variables to sequences of sorts and types. Note $\bot$ is *not* itself a type; it denotes that a session channel is used by two safely compatible and complete types. We write $\Delta$ $\mathtt{completed}$ to mean for all $k \in \mathrm{dom}(\Delta)$, $\Delta(k)$ is **end** or $\bot$. The notation (e.g.) $\tilde{x} : \tilde{S}$ means a (possibly empty) sequence of mappings $x_i : S_i$. We write $\Delta \cdot k : T$ (or $\Theta \cdot X : \tilde{S}, \tilde{T}$ or $\Gamma \cdot x : S$) for the addition of a (disjoint) mapping $k : T$ (or $X : \tilde{S}, \tilde{T}$ or $x : S$) to a typing (or basis or sorting).

We extend RC to typings and define composition of RC typings. Typings $\Delta_1$ and $\Delta_2$ are RC, written $\Delta_1 \asymp \Delta_2$, iff $\Delta_1(k) \bowtie \Delta_2(k)$ for all $k \in \mathrm{dom}(\Delta_1) \cap \mathrm{dom}(\Delta_2)$. Note, $\bowtie$ is defined only for types and *not* for $\bot$. If $\Delta_1 \asymp \Delta_2$, then the *composition* of $\Delta_1$ and $\Delta_2$, written $\Delta_1 \circ \Delta_2$, is the typing such that $(\Delta_1 \circ \Delta_2)(k)$ is (1) $\bot$ if $k \in \mathrm{dom}(\Delta_1) \cap \mathrm{dom}(\Delta_2)$; (2) $\Delta_i(k)$ if $k$ is present in $\Delta_i$ but not $\Delta_{\{1,2\}\setminus i}$; and (3) undefined otherwise.

[4] See the supplementary material for an example derivation of RC for the "2-3 Producer Consumer" process from Fig 4

$$\text{(T-Send)}\ \frac{\Gamma \vdash e \triangleright S \quad \Theta; \Gamma \vdash \langle P, c\rangle \triangleright \Delta \cdot k : T}{\Theta; \Gamma \vdash \langle k![e];\ P, c\rangle \triangleright \Delta \cdot k : !S.T} \qquad \text{(T-Recv)}\ \frac{\Theta; \Gamma \cdot x : S \vdash \langle P, c\rangle \triangleright \Delta \cdot k : T}{\Theta; \Gamma \vdash \langle k?(x);\ P, c\rangle \triangleright \Delta \cdot k : ?S.T}$$

$$\text{(T-Select)}\ \frac{\Theta; \Gamma \vdash \langle P_j, c\rangle \triangleright \Delta \cdot k : T_j \quad 1 \le j \le n}{\Theta; \Gamma \vdash \langle k \triangleleft \ell_j;\ P_j, c\rangle \triangleright \Delta \cdot k : \oplus\{\ell_i : T_i\}_{i \in 1..n}} \qquad \text{(T-Offer)}\ \frac{\Theta; \Gamma \vdash \langle P_i, c\rangle \triangleright \Delta \cdot k : T_i}{\Theta; \Gamma \vdash \langle k \triangleright \{\ell_i : P_i\}_{i \in 1..n}, c\rangle \triangleright \Delta \cdot k : \&\{\ell_i : T_i\}_{i \in 1..n}}$$

$$\text{(T-If)}\ \frac{\Gamma \vdash e \triangleright \texttt{bool} \quad \Theta; \Gamma \vdash \langle P, c\rangle \triangleright \Delta \quad \Theta; \Gamma \vdash \langle Q, c\rangle \triangleright \Delta}{\Theta; \Gamma \vdash \langle \texttt{if}\ e\ \texttt{then}\ P\ \texttt{else}\ Q, c\rangle \triangleright \Delta} \qquad \text{(T-Par)}\ \frac{\Theta; \Gamma \vdash \langle P, c\rangle \triangleright \Delta \quad \Theta; \Gamma \vdash \langle Q, c\rangle \triangleright \Delta' \quad \Delta \asymp \Delta'}{\Theta; \Gamma \vdash \langle P \mid Q, c\rangle \triangleright \Delta \circ \Delta'}$$

$$\text{(T-Inact)}\ \frac{\Delta\ \texttt{completed}}{\Theta; \Gamma \vdash \langle \mathbf{0}, c\rangle \triangleright \Delta} \qquad \text{(T-Bot)}\ \frac{\Theta; \Gamma \vdash \langle P, c\rangle \triangleright \Delta \cdot k : \mathbf{end}}{\Theta; \Gamma \vdash \langle P, c\rangle \triangleright \Delta \cdot k : \bot} \qquad \text{(T-Chan)}\ \frac{\Theta; \Gamma \vdash \langle P, c\rangle \triangleright \Delta \cdot k : \bot}{\Theta; \Gamma \vdash \langle (\nu k)\, P, c\rangle \triangleright \Delta}$$

$$\text{(T-Prd)}\ \frac{\begin{array}{c} X : \tilde{S}, \tilde{T}; \Gamma \cdot \tilde{x} : \tilde{S} \vdash \langle P, c\rangle \triangleright \tilde{k} : \tilde{T}' \qquad \Theta \cdot X : \tilde{S}, \tilde{T}''; \Gamma \vdash \langle Q, c\rangle \triangleright \Delta \qquad d \ge c \\ \texttt{static}\ P \qquad \tilde{T} = (\omega_n t_i.T_i)_{i \in 1..m} \qquad \tilde{T}' = \texttt{unfold}(\tilde{T}) \qquad \tilde{T}'' = (\omega_n^d t_i.T_i)_{i \in 1..m} \end{array}}{\Theta; \Gamma \vdash \langle \texttt{prd}(n)\, X(\tilde{x}; \tilde{k}) = P\ \texttt{in}\ Q, c\rangle \triangleright \Delta}$$

$$\text{(T-Inv-S)}\ \frac{\Gamma \vdash \tilde{e} \triangleright \tilde{S} \quad \tilde{T} = (\omega_n t_i.T_i)_{i \in 1..m} \quad \Delta\ \texttt{completed}}{\Theta \cdot X : \tilde{S}, \tilde{T}; \Gamma \vdash \langle X[\tilde{e}; \tilde{k}], c\rangle \triangleright \Delta \cdot \tilde{k} : \tilde{T}} \qquad \text{(T-Inv-R)}\ \frac{\Gamma \vdash \tilde{e} \triangleright \tilde{S} \quad \tilde{T} = (\omega_n^d t_i.T_i)_{i \in 1..m} \quad \Delta\ \texttt{completed}}{\Theta \cdot X : \tilde{S}, \tilde{T}; \Gamma \vdash \langle X^d[\tilde{e}; \tilde{k}], c\rangle \triangleright \Delta \cdot \tilde{k} : \tilde{T}}$$

$$\text{(T-Fold)}\ \frac{\Theta; \Gamma \vdash \langle P, c\rangle \triangleright \Delta \cdot k : \texttt{unfold}(\omega_n^d t.T)}{\Theta; \Gamma \vdash \langle P, c\rangle \triangleright \Delta \cdot k : \omega_n^d t.T}$$

---

$$\frac{}{\texttt{static}\ X[\tilde{e}, \tilde{k}]} \qquad \frac{}{\texttt{static}\ \mathbf{0}} \qquad \frac{\texttt{static}\ P \qquad \texttt{static}\ Q}{\texttt{static if}\ e\ \texttt{then}\ P\ \texttt{else}\ Q}$$

$$\frac{\texttt{static}\ P}{\texttt{static}\ k![e];\ P} \qquad \frac{\texttt{static}\ P}{\texttt{static}\ k?(x);\ P} \qquad \frac{\texttt{static}\ P}{\texttt{static}\ k \triangleleft \ell;\ P} \qquad \frac{\texttt{static}\ P_1 \ \ldots\ \texttt{static}\ P_n}{\texttt{static}\ k \triangleright \{\ell_i : P_i\}_{i \in 1..n}}$$

Fig. 14. (top) State typing $\Theta; \Gamma \vdash \sigma \triangleright \Delta$. (bottom) Static process definition condition `static` $P$.

Our type system uses two forms of judgement. The judgements on *expressions* $\Gamma \vdash e : S$ type $e$ with sort $S$ under sorting $\Gamma$, and are defined as expected; expressions do not involve any communication behaviour. Fig. 14 defines the judgements on *states* $\Theta; \Gamma \vdash \sigma \triangleright \Delta$.

We first explain the rules that are standard [Honda et al. 1998]. Rule (T-Send) types a send of sort $S$ on channel $k$ followed by a continuation of type $T$. Dually, rule (T-Recv) for a receive of sort $S$. Rule (T-Offer) types a branch offering labels $\ell_{1..n}$ with corresponding continuations $T_{1..n}$. Dually, rule (T-Select) types the branch select of label $\ell_j$ from some $\ell_{1..n}$ options followed by the corresponding continuation $T_j$. Rule (T-If) types the cases of a conditional by the same typing; note, (T-Select) allows the conditional cases to select different labels of the branch.

Rule (T-Inact) types the inactive process $\mathbf{0}$ by requiring all channels in the typing map are complete, i.e., **end** or $\bot$. Rule (T-Par) types the composition of processes $P$ and $Q$ provided their typings are compatible. Although this rule itself is standard, composability of typings in our system is determined by *rate* compatibility of (periodic) behaviours. Composition sets an RC channel pairing to $\bot$ in the resulting typing, enforcing the binary aspect of our sessions. Rule (T-CRes) ensures that channel behaviours are safely completed and composed within their scope.

We now explain our new rules. (T-Prd) types the periodic recursion construct, enforcing several key properties. First, the definition $P$ of the periodic process must satisfy `static` $P$. It must also be typed with a basis mapping process variable $X$ to the sorts $\tilde{S}$ and the *static* periodic types $\tilde{T}$ of its parameters, and the sorting extended with $\tilde{S}$. Note, this basis contains *only* $X$ (it does not extend $\Theta$), restricting $P$ from referring to other processes with potentially different periods. Hence, our periodic processes cannot be mutually recursive. The structure of our rules only allows a type to be unfolded here as part of a processes definition. The prohibition on mutually recursive processes guarantees that only one definition is relevant to the typing of a channel. Thus this restriction also prevents the use of nested periodic types, as unfolding during type checking would require statically nesting mutually recursive processes definitions. Specifically, it is impossible to type a state using nested periodic types (unless that assumption were already to appear in the basis $\Theta$). This reflects the assumption of our formalism that periods are non-variable during execution. The notation $(\ldots)_{1..m}$ means a sequence of $i \in 1..m$ indexed items, and $\texttt{unfold}(\tilde{T})$ stands for the sequence of unfoldings of each $T_i \in \tilde{T}$. The resulting typing of $P$ must map the channel parameters to the corresponding one-iteration unfoldings, i.e., $P$ must implement one iteration of the periodic behaviour.

Second, the run-time process $Q$ is typed with the basis given by replacing the static periodic types with *run-time* periodic types with a deadline $d$. The rule requires that the deadline has not passed, i.e., $d \leq c$. Rule (T-Fold) allows behaviours of $Q$ that can effectively be folded to those run-time periodic types. The overall typing of the periodic recursion is given by the typing of the $Q$.

Within the static definition of a process $X$, rule (T-Inv-S) may be used to type *static* invocations of $X$ with the corresponding parameters. An invocation represents reaching the end of an iteration: all channel parameters are expected to match the periodic types prior to the original unfolding by the (T-Prd) premise. Similarly, rule (T-Inv-R) types *run-time* invocations of $X$ within the run-time process $Q$ of periodic recursions. It matches the invocation deadline to that in the basis – rule (T-Prd) checks that the deadline is not actually missed w.r.t. the current system time. Rules (T-Inv-S) and (T-Inv-R) do not permit partially executed iterations of periodic behaviours to be carried over an invocation, i.e., a periodic behaviour must be fully completed before commencing a new iteration. Incomplete behaviours at an invocation boundary cannot be typed, which we exploit via type preservation to show the execution of well-typed systems is from such errors (Sec. 5.3).

In conjunction with (T-Prd), rule (T-Fold) permits an iso-recursive treatment of periodic types, but only allows subsumption in one direction — there is no converse of (T-Fold). This corresponds to the fact that time is monotonic, and the static definition of a periodic process must implement an iteration to productively follow the advance of time. Limiting (T-Fold) to folding (cf. expansion in RC) imposes a bound on the type checking of individual processes. Folding reduces the remainder length of a type, and thus its structure puts a limit on how many times (T-Fold) can be applied. Combined with the LCM observation for RC, this yields a simple decidable algorithm for type checking overall.

### 5.3 Type System Properties

We establish the key properties of our rate-based session type system. We formalize the notion of *rate error* and prove that well-typed states are safe in terms of *rate safety*, i.e., the execution of well-typed states is guaranteed free of rate errors. First, we present supporting auxiliary results.

Lemma 5.1 (Weakening). *Let* $\Theta; \Gamma \vdash \sigma \triangleright \Delta$. (1) *If* $X \notin \mathrm{dom}(\Theta)$, *then* $\Theta \cdot X : \tilde{S}, \tilde{T}; \Gamma \vdash \sigma \triangleright \Delta$; (2) *if* $x \notin \mathrm{dom}(\Gamma)$, *then* $\Theta; \Gamma \cdot x : S \vdash \sigma \triangleright \Delta$; *and* (3) *if* $k \notin \mathrm{dom}(\Delta)$ *and* $T$ *is* $\bot$ *or* **end**, *then* $\Theta; \Gamma \vdash \sigma \triangleright \Delta \cdot k : T$.

Lemma 5.2 (Substitution). *If* $\Theta; \Gamma \cdot x : S \vdash \langle P, c \rangle \triangleright \Delta$ *and* $\Gamma \vdash v : S$, *then* $\Theta; \Gamma \vdash \langle P[v/x], c \rangle \triangleright \Delta$.

Lemma 5.3 (Congruence). *If* $\Theta; \Gamma \vdash \langle P, c\rangle \triangleright \Delta$ *and* $P \equiv Q$, *then* $\Theta; \Gamma \vdash \langle Q, c\rangle \triangleright \Delta$.

Lemma 5.4 (Continuation). (1) *If* $!S.T_1 \bowtie ?S.T_2$, *then* $T_1 \bowtie T_2$; *and* (2) *if* $\oplus\{\ell_i : T_i\}_{i\in 1..n} \bowtie \&\{\ell_i : T_i'\}_{i\in 1..n}$, *then* $\forall 1 \le i \le n. T_i \bowtie T_i'$.

Lemma 5.5 (Deadline Substitution). *Let* $\tilde{T} = (\omega_n t_i.T_i)_{i\in I}$ *where* $I = 1..m$. *If* $X : \tilde{S}, \tilde{T}; \Gamma \vdash \langle P, c\rangle \triangleright \tilde{k} : \mathtt{unfold}(\tilde{T})$ *with* `static` $P$ *and* $d \ge c$, *then* $X : \tilde{S}, (\omega_n^{d+n} t_i.T_i)_{i\in I}; \Gamma \vdash \langle P[X^{d+n}/X], c\rangle \triangleright \tilde{k} : \mathtt{unfold}((\omega_n^d t_i.T_i)_{i\in I})$.

Lemma 5.6 (Type-Process Deadline Relation). *If* $\Theta; \Gamma \vdash \langle P, c\rangle \triangleright \Delta \cdot k : T$, *then* $\mathtt{dl}(T) \le \mathtt{dl}_k(P)$.

In addition to standard session types properties, we introduce three key lemmas for our system. Lemma 5.4 states if two types are compatible, then their relevant continuations are compatible. Lemma 5.5 states that instantiating a well-typed static definition of a periodic process into a run-time process with deadline $d$ is well-typed provided the deadline is not passed. Lemma 5.6 relates the deadlines that appear in a process with the deadlines that appear in the types of its channels. Specifically, the deadline of a type must be no later than the deadline in any process that uses a channel with that type.

We can now present the first major result, *type preservation*. Our typing ensures pairings of channel behaviors are *rate* compatible, composing them as $\bot$ regardless of their specific behavior. The essence of the proof is thus to show after any reduction step that RC pairings remain RC. Our proof follows that of Yoshida and Vasconcelos [2007], with duality generalized by rate compatibility (and without the cases for session delegation). We outline the proof below, leaving the full details to the supplemental material. A technicality of note, our operational semantics (c.f. Sec. 4.2) models deadline violation via stuck states: reduction by (Delay) is prohibited when it would result in a deadline miss. This prevents trivial violations of type preservation via repeated use of (Delay). Theorem 5.9.1 Rate Error Freedom guarantees well-typed states never get stuck *due to rate errors*.

Theorem 5.7 (Preservation). *If* $\Theta; \Gamma \vdash \sigma_1 \triangleright \Delta$ *and* $\sigma_1 \longrightarrow \sigma_2$, *then* $\Theta; \Gamma \vdash \sigma_2 \triangleright \Delta$

Proof Sketch. By induction on the derivation of the reduction step, proceeding by cases on the last rule applied. We illustrate two key cases.

Case (Com) depends on the property that RC between two types is preserved by the continuations of those types. From (Com), after accounting for context rules, (e.g. (T-Bot)), we have $\Delta = (\Delta_1 \circ \Delta_2) \cdot k : \bot$. From (T-Par), (T-Send) and (T-Recv), we know $\Theta; \Gamma \vdash \langle P, c\rangle \triangleright \Delta_1 \cdot k : T_1$ (and analogous for $Q$). Using Lemma 5.2, we obtain $\Theta; \Gamma \vdash \langle Q\,[v/x]\,, c\rangle \triangleright \Delta_2 \cdot k : T_2$ We must show that the composition is well-typed under the original typing, i.e., $\Theta; \Gamma \vdash \langle P \mid Q\,[v/x]\,, c\rangle \triangleright \Delta$ for (T-Par). In particular, we need $\Delta_1 \cdot k : T_1 \asymp \Delta_2 \cdot k : T_2$. This follows from Lemma 5.4 Continuation and the definition of typing compatibility.

Case (Inv) handles process invocation by connecting the types of static definitions and run-time iterations of periodic processes. From (T-Inv-R) and (T-Prd), we know $\Delta = \Delta_0 \cdot \tilde{k} : \tilde{T}'$ where $\tilde{T}' = (\omega_n^d t_i.T_i)_{i\in I}$ and $\Delta_0$ `completed`. The typing of the static definition in (T-Prd) and Lemma 5.2 gives us

$$X : \tilde{S}, \tilde{T}; \Gamma \cdot \tilde{x} : \tilde{S} \vdash \langle P[\tilde{v}/\tilde{x}], c\rangle \triangleright \tilde{k} : \mathtt{unfold}(\tilde{T}) \qquad \tilde{T} = (\omega_n t_i.T_i)_{i\in I} \qquad I = 1..m$$

By applying Lemma 5.5 with $d = c$ and then using (T-Prd):

$$\Theta; \Gamma \vdash \langle \mathtt{prd}(n)\, X(\tilde{x}, \tilde{k}) = P \ \mathtt{in}\ P[\tilde{v}/\tilde{x}][X^{d+n}/X], c\rangle \triangleright \tilde{k} : \mathtt{unfold}((\omega_n^d t_i.T_i)_{i\in I})$$

By applying Lemma 5.1 and (T-Fold), the above is also well-typed under the original typing:

$$\Theta; \Gamma \vdash \langle \mathtt{prd}(n)\, X(\tilde{x}, \tilde{k}) = P \ \mathtt{in}\ P[\tilde{v}/\tilde{x}][X^{d+n}/X], c\rangle \triangleright \Delta_0 \cdot \tilde{k} : \tilde{T}'$$

□

We use type preservation to show the second major result, *rate safety*. Consider this run-time scenario of a critical rate compatibility error. Process $P$ is trying to communicate with process $Q$, but $Q$ is at a process invocation (i.e., $Q$ has completed its current iteration). $P$ must wait for $Q$ to reach its deadline $d_2$ (i.e., for its current period to expire) and commence its next iteration, before $P$ can complete its blocked communication action. If $P$ has a deadline $d_1$ (i.e., on an invocation after its currently blocked communication prefix) for an *earlier* timestep than $Q$ is waiting for ($d_1 < d_2$), then it is impossible for $P$ to meet its deadline. The following formalizes this notion of rate error.

*Definition 5.8 (Errors).* A process $P$ is *communicating on* $k$ if it is one of the following forms: (1) $k![e];\ P'$, (2) $k?(x);\ P'$, (3) $k \triangleleft \ell;\ P'$, or (4) $k \triangleright \{\ell_i : P'_i\}_{i \in 1..n}$.

- A $k$-*redex* is a parallel composition of two processes congruent to (a) $k![e];\ P \mid k?(e);\ P'$, or (b) $k \triangleleft \ell;\ P \mid k \triangleright \{\ell_i : P'_i\}_{i \in 1..n}$. A state $\sigma$ is a *communication error* if $\sigma \equiv \langle \mathtt{prd}(n)\, X(\tilde{x}, \tilde{k}) = P \ \mathtt{in}\ (\nu \tilde{k'})\,(Q \mid R), c\rangle$ and $Q$ is parallel composition of two or more processes communicating on $k$ that is not a $k$-redex.
- A state $\sigma$ is a *rate error* if

$$\sigma \equiv \langle (\nu \tilde{k})((\mathtt{prd}(n_p)\, X(\tilde{x_p}, \tilde{k_p}) = P' \ \mathtt{in}\ P) \mid (\mathtt{prd}(n_q)\, Y(\tilde{x_q}, \tilde{k_q}) = Q' \ \mathtt{in}\ Y^{d_q}[\tilde{e}, \tilde{k_q}]) \mid R), c\rangle$$

  where $P$ is communicating on $k \in \tilde{k_q}$ and $\mathtt{dl}_k(P) < d_q$.

A state $\sigma$ is an *error* if it is a communication error or a rate error.

Theorem 5.9 (Rate Safety). *If $\Theta; \Gamma \vdash \sigma \triangleright \Delta$ for a state $\sigma$, then $\sigma$ is not an error.*

Proof Sketch. Assume a well-typed state $\sigma$ is a rate error. Then $\sigma \equiv \sigma' = \langle (\nu \tilde{k})(P \mid Q \mid R), c\rangle$ by Def. 5.8. Assume the case where $P$ is communicating on $k$ as an external choice; the other cases are analogous or simpler. Thus let $P = \mathtt{prd}(n_p)\, X(\tilde{x_p}, \tilde{k_p}) = P' \ \mathtt{in}\ k \triangleright \{\ell_1 : P'_1; \ldots; \ell_n : P'_n\}$, $Q = \mathtt{prd}(n_q)\, Y(\tilde{x_q}, \tilde{k_q}) = Q' \ \mathtt{in}\ Y^{d_q}[\tilde{e}, \tilde{k_q}]$, and $k \in \tilde{k_q}$. Let $T_P$ and $T_Q$ be the typings of $P$ and $Q$ on channel $k$. Since $\sigma'$ is well-typed (Lemma 5.3), by (T-Par) there exists a type $T_3$ such that $T_P \bowtie T_Q$ by (RC). We then show such an $\sigma'$ cannot be well-typed by analyzing the relationship between $T_P$, $T_Q$ and $T_3$ from two separate perspectives, and finding a contradiction. These perspectives stem from analyzing the run-time state $\sigma'$ in terms of (i) the constraints on *deadlines* present in $P$ and $Q$, versus (ii) the number of remaining *communications* in $P$ and $Q$ before the next superperiod begins.

In both cases, we consider the expansions of $T_P$ and $T_Q$ to $T_3$ and $\overline{T_3}$; these expansions involve zero or more unfoldings for each expansion to reach superperiod type $T_3$. In each case, we derive an inequality that relates the periods $n_p$ of $T_P$ and $n_q$ of $T_Q$, to factors $a_1$ and $a_2$ which count the number of times $T_P$ and $T_Q$ must be unfolded to match the type $T_3$ in order for RC to hold. We find that by deadline, the number of unfoldings multiplied by the period in $T_P$ ($a_1 \cdot n_p$) must be greater than in $T_Q$ ($a_2 \cdot n_q$). This stems from $Q$ having a later deadline than $P$, so it will take more unfolding for $P$ to catch up. Looking at the remaining communications, we find the opposite: the number for $T_P$ ($a_1 \cdot n_p$) must be less than or equal to that for $T_Q$ ($a_2 \cdot n_q$). Process $Q$ is waiting to repeat, so it has no remaining communications. Thus we need to unfold $T_Q$ at least as far into the future as $T_P$ for $Q$ to catch up on communications. These inequalities are in direct contradiction, and thus we can conclude $\sigma$ cannot be well-typed. In more detail:

(i) *Deadlines.* From Def. 5.8, $\mathtt{dl}_k(P) < \mathtt{dl}_k(Q)$, and by Lemma 5.6, $\mathtt{dl}(T_P) \leq \mathtt{dl}_k(P)$. By (T-Prd), $\mathtt{dl}_k(Q) = \mathtt{dl}(T_Q)$, and we can infer $\mathtt{dl}(T_P) < \mathtt{dl}(T_Q)$. By duality, $\mathtt{dl}(T_3) = \mathtt{dl}(\overline{T_3})$, hence we can equate the deadline $\mathtt{dl}(T_p) + a_1 \cdot n_p = \mathtt{dl}(T_q) + a_2 \cdot n_q$, and can conclude (1) $a_1 \cdot n_p > a_2 \cdot n_q$.

(ii) *Number of communications.* We first determine the remaining communications in the current iterations from types $T_P$ and $T_Q$ when scaled up and unfolded to the superperiod type $T_3$. For $P$, let $r_3 = \mathtt{rl}(T_3) = a_1 \cdot \mathtt{bl}(T_P) + \mathtt{rl}(T_P)$. Similarly for $Q$ and by duality, $r_3 = \mathtt{rl}(\overline{T_3}) = a_2 \cdot \mathtt{bl}(T_Q) + \mathtt{rl}(T_Q)$.

Looking at the structure of $Q$, we need to apply (T-Inv-R) to $Y^{d_q}[\tilde{e}, \tilde{k_q}]$ to get $T_Q$. Thus we can conclude $T_Q$ starts with a periodic construct and $\mathtt{rl}(T_Q) = 0$. Remainder length is non-negative, so we also know $\mathtt{rl}(T_P) \geq 0$. Since $a_1 \cdot \mathtt{bl}(T_P) + \mathtt{rl}(T_P) = a_2 \cdot \mathtt{bl}(T_Q) + \mathtt{rl}(T_Q)$, we have (2) $a_1 \cdot \mathtt{bl}(T_P) \leq a_2 \cdot \mathtt{bl}(T_Q)$. Let the communication rate $R(T_3) = u$. Since rates are preserved by expansion and duality, we know $\frac{\mathtt{bl}(T_P)}{n_p} = \frac{\mathtt{bl}(T_Q)}{n_q}$. With (2), we obtain $a_1 \cdot \frac{\mathtt{bl}(T_P)}{n_p} \cdot n_p \leq a_2 \cdot \frac{\mathtt{bl}(T_Q)}{n_q} \cdot n_q$, and we can conclude (3) $a_1 \cdot n_p \leq a_2 \cdot n_q$.

Conclusions (1) and (3) are contradictions and show that no such well-typed $\sigma'$ exists.

□

Corollary 5.9.1 (Rate Error Freedom). *Let $\longrightarrow^*$ be the reflexive and transitive closure of state reduction $\longrightarrow$. A well-typed state $\sigma$ never reduces to a state $\sigma \longrightarrow^* \sigma'$ where $\sigma'$ is an error.*

## 6 IMPLEMENTATION AND APPLICATIONS IN RUST

We demonstrate an implementation of our framework as a Rust library (known in Rust as a *crate*). The core of this implementation is an embedding of our session type system into the native type system of Rust. We encode the RC relation as a Rust trait and leverage the Rust compiler's type checker to check whether two types are RC. This allows the unmodified Rust compiler to statically verify rate safety of periodic systems without requiring external preprocessing or code generation steps. By working within the type system of Rust, we gain support for programming of periodic processes with generic periods, and inference of periods in certain cases. A key motivation of using Rust is the existing support and tooling for embedded systems: as an example, we highlight an implementation of the PineTime Heart Rate Sensor from Sec. 2 using our framework. We reimplemented all stages of the pipeline in Rust based on the open-source C++ implementation[5], and run it on FreeRTOS [FreeRTOS 2003], the same real-time operating system used by the PineTime smartwatch. We have deployed our implementation on an STM32F407 ARM Cortex-M4 microcontroller[6] using 64 KiB of RAM (Fig.7).

This section also summarizes a range of other application protocols from the embedded and IoT domains expressed in our framework. The artifact [Iraci et al. 2023] accompanying this paper includes our framework as a Rust library, the examples showcasing its functionality, a port of our framework to FreeRTOS, and the full implementation of the Heart Rate Sensor.

### 6.1 Overview: Smartwatch Heart Rate Sensor

Recall the Automatic Gain Control (AGC) stage of the Heart Rate Sensor (Sec. 2). Fig. 15 lists the session-typed Rust code for the AGC task programmed and type checked using our framework. The full implementation uses existing open-source Rust crates for various hardware interfaces, such as the heart-rate sensor and display; we omit those details here, but include them in the artifact.

*Programming periodic tasks in Rust.* Fig. 15 gives a Rust function `task_agc` that constructs a new AGC task. We summarise how our framework is used to define this task. For convenience, our framework provides helper macros that expand into the full constructs and types of our embedding. In Rust, a macro call is distinguished by a '`!`' symbol after the macro name.

The return type of `task_agc` is coded using the macro `Task![N; S; T]`. In general, the macro defines an underlying Rust type that represents a periodic task $\mathtt{prd}(n)\, X(\tilde{x}, \tilde{k}) = P$ `in` $Q$, where `N` is a (zero-sized) type representing $n$, `S` is the sort of $\tilde{x}$, and `T` are the types of $\tilde{k}$. For `task_agc`

[5]Available at https://github.com/InfiniTimeOrg/InfiniTime/blob/241d364/src/components/heartrate/Ppg.cpp.
[6]PineTime development units were unavailable at the time of implementation – we restricted our deployment to 64 KiB RAM (of the 192 KiB on the STM32F407) to match the 64 KiB RAM on the nRF52832 used in the PineTime.

```
fn task_agc() -> Task![ONE; AGC; Prd<ONE, P, Recv<f32, P>>, Prd<ONE, P, Send<f32, P>>] {
    task![mut agc : AGC; k1, k2 : Recv<f32, P>, Send<f32, P> => {
        let (x, k1) = k1.recv();  // Note: the macro implicitly types k1 as Chan<Recv<f32, P>>
        let y = agc.step(x);
        let ((), k2) = k2.send(y); // ...and k2 as Chan<Send<f32, P>>
        Continue(chans![k1, k2], agc)}]
}
```

Fig. 15. Session-typed Rust code implementing the PineTime AGC process from Sec. 2.

specifically, `N` is `ONE`, and `S` is a Rust struct `AGC` containing the data for the computations performed by this task. The two session types are explained next.

Our framework takes session types encoded by the programmer into a set of provided Rust types:

$$[\![\omega_n t.T]\!] = \texttt{Prd<}[\![n]\!]\texttt{, P, }[\![T]\!]\texttt{>} \quad [\![!\texttt{S}.T]\!] = \texttt{Send<S, }[\![T]\!]\texttt{>} \quad [\![?\texttt{S}.T]\!] = \texttt{Recv<S, }[\![T]\!]\texttt{>} \quad [\![t]\!] = \texttt{P}$$

$$[\![\&\{\ell : T_L,\ r : T_R\}]\!] = \texttt{Offer<}[\![T_L]\!]\texttt{, }[\![T_R]\!]\texttt{>} \quad [\![\oplus\{\ell\ : T_L,\ r : T_R\}]\!] = \texttt{Select<}[\![T_L]\!]\texttt{, }[\![T_R]\!]\texttt{>} \quad [\![\mathbf{end}]\!] = \texttt{End}$$

An AGC task uses two channels: one for input and one for output. Specifically, the input channel has the associated session type $\omega_1 t.?\texttt{f32}.t$, encoded as `Prd<ONE, P, Recv<f32, P>>`, where `P` is a special type used to denote recursion variables. Similarly, the output channel has the session type $\omega_1 t.!\texttt{f32}.t$, encoded as `Prd<ONE, P, Send<f32, P>>`. The encoding of send/receive types is similar to that of Jespersen et al. [2015]; periodic types are new to this work.

The macro `task![s: S; k̃: T̃]` encodes the static definition $P$ of a task $\texttt{prd}(n)\ X(\tilde{x}, \tilde{k}) = P$ `in` $Q$, with `s: S` corresponding to $\tilde{x}$ and its sort, and `k̃: T̃` to $\tilde{k}$ and the bodies of their periodic types. Specifically, `agc: AGC`, and `k1, k2` with the body types of the above `Prd` types. Uses of this macro can be read as defining the main loop to be executed by the periodic task. The session type annotation $T$ in the macro is used to specify the channel type `Chan<T>`, which is parameterised by the session type $T$. The Rust keyword `mut` specifies the internal state `agc` is mutable.

Our Rust *channels* offer functions for performing the I/O actions of their corresponding sessions. Each function returns both its result and a channel with the appropriate continuation type. E.g., a `Chan<Send<T, K>>` type offers a function `send(v: T) -> ((), Chan<K>)` – note the constraints on `T` and `K` between the channel and function types. Similarly, a `Chan<Recv<T, K>>` offers a function `recv() -> (T, Chan<K>)`. The `agc.step(x)` function call performs the local computation of the AGC (details omitted). For branching, `Chan<Select<L, R>>` exposes two functions: `left() -> ((),Chan<L>)` and `right() -> ((),Chan<R>)`. On the offering side, `Chan<Offer<L, R>>` provides the function `offer() -> ((), Either<Chan<L>, Chan<R>>)`, where pattern matching on the value of `Either<Chan<L>, Chan<R>>` determines which branch to take.

A task ends its current iteration and waits to commence the next by returning a special `Continue` value. `Continue` requires the type of every channel parameter to be `P`, ensuring the implementation performs a complete iteration of the task. The macro `chans![...]` constructs (session) typed sets of channel values, similar to Rust's `vec!` macro for vectors. To terminate, a task can return an analogous `Terminate`, whose constructor requires the type of every channel to be `End`. Typing thus ensures all channel behaviors are completed.

Our framework does *not* include Rust representations of the *run-time* entities in our theory, i.e., run-time invocations and periodic types that carry *deadline* information. Users program the static definitions of periodic tasks, which are typed by *static* typing entities (cf. rule (T-Prd)); our framework effectively types user programs as initial states (Sec. 4.2, i.e. at system time 0), and with run-time processes of all periodic tasks that initially invoke at time 0. Our framework does *not* perform any dynamic checks for rate safety. The run-time entities in our theory are only for the theoretical representation of run-time states and their typing, for proving the theorems in Sec. 5.3.

*Linear usage of session channels.* The above illustrates how our Rust channel types statically ensure that I/O actions performed on any channel correspond to its associated session type. For example, static type checking would not permit performing a `send` on channel `k1` at any point in Fig. 15. The next key aspect of a session typing system is to enforce *linear* usage of channels, corresponding to the linear treatment of the $\Delta$ environment in our type system.

A notable feature of Rust is support for *affine* typing. All the variables in Fig. 15 – crucially, the channel variables – are affinely typed by Rust, i.e., every variable may be used *at most* once. In Rust, the affine property of type checking would not permit using the channel `k1` to `recv` twice.

To obtain linearity, we structure the typing constraints of `Continue` and `Terminate` to ensure every channel variable is used *at least* once to construct the return value. Since tasks are implemented as Rust functions, failing to return one of these return values will lead to a type error. In Fig. 15, static type checking requires the `Continue` to be provided a collection of two channels, where the type of each channel must correspond to the periodic recursion variable of its associated session type: the type of `k1` on line 6 corresponds to the `P` of `Recv<f32, P>`, and the type of `k2` corresponds to the `P` of `Send<f32, P>`. Crucially, the *only* way to obtain values that satisfy the typing requirements of `Continue` is by using channel variables (i.e., by performing I/O operations) at least once. Starting from the initial `k1` declared on line 2, the only way to obtain the `k1` corresponding to `P` for the `Continue` on line 6 is to peform the `recv`. Similarly, a `Terminate` must be provided `End` channels, which can only be obtained by performing I/O operations at least once. Since `P` and `End` are distinct, the protocol will only allow one of `Continue` or `Terminate` to be constructed. Together, these mechanisms statically ensure *linearity* of channel usages, i.e., *exactly* once usage of every channel variable, in accordance with our type system.

*6.1.1 Rate Compatibility in Rust.* Key to safety in our practical framework, and a key difference from previous implementations of session types in Rust [Jespersen et al. 2015], is our embedding of RC checking into Rust type checking. We first outline how our formulation of RC from Sec. 5.1 yields a decidable algorithm for checking whether two periodic session types are RC. We then demonstrate how this decision procedure is used in our framework.

*An algorithm for checking RC.* Recall Rule (RC) from Fig. 12, for which we will now provide an algorithmic decision procedure. To determine whether session types $T_1$ and $T_2$ are RC, we must either (i) find a $T_3$ where $T_3$ is an expansion of $T_1$ and $\overline{T_3}$ is an expansion of $T_2$ ($T_1 <: T_3$ and $T_1 <: \overline{T_3}$), or (ii) show no such $T_3$ exists. We observe the following properties of expansion:

- Applying the $\mathsf{exp}$ by $m$ operation to a type with period $n$ yields a type with period $m \cdot n$. Duality between types is preserved by applying $\mathsf{exp}$ by $m$ to both, and no other rule changes the period.
- The period of any such $T_3$, $\mathsf{pd}(T_3)$, is thus a multiple of $\mathsf{pd}(T_1)$, and $\mathsf{pd}(\overline{T_3})$ is a multiple of $\mathsf{pd}(T_2)$. Duality preserves periods, so $\mathsf{pd}(T_3) = \mathsf{pd}(\overline{T_3})$ and is a *common* multiple of $\mathsf{pd}(T_1)$ and $\mathsf{pd}(T_2)$.
- Any common multiple of two numbers must be a multiple of their least common multiple. A corresponding property holds for expansion. Any expansion to a common multiple period must itself be an expansion of one to the least common multiple period.

From these observations, we can conclude any $T_3$, such that $T_1 <: T_3$ and $T_1 <: \overline{T_3}$, must thus be an expansion of a $T_3'$ where $\mathsf{pd}(T_3')$ is the least common multiple of $\mathsf{pd}(T_1)$ and $\mathsf{pd}(T_2)$. Since this transformation preserves duality, $T_3$ meets the requirements for (RC) if and only if $T_3'$ also does. This logical equivalence extends to any multiple of $T_3'$.

Note that the $\mathsf{exp}$ operation (used in expansion) and duality are purely syntactic operations and decidable. Thus we can use an expansion to any common period as a representative $T_3$ to test. This leads to the following algorithm for determining whether two types $T_1$ and $T_2$ are RC:

```
1  fn main() {
2      // ... Parse the input data and create the channels ...
3      let (kai, kao) = new_chan!(Prd<ONE, P, Recv<f32, P>>, Prd<ONE, P, Send<f32, P>>);
4      let (kbi, kbo) = new_chan!(Prd<ONE, P, Recv<f32, P>>, Prd<ONE, P, Send<f32, P>>);
5      let _agc = spawn!(task_agc(), || chans!(kai, kbo), || AGC::new(400.0, 0.971, 2.0));
6      // ... Instantiate the rest of the pipeline and run indefinitely ...
7  }
```

Fig. 16. Allocation of rate-compatible channels to safely compose AGC with its pipeline neighbors.

> Let $i \in \{1, 2\}$, $n_i = \mathsf{pd}(T_i)$, and $n$ be a common multiple of $n_1$ and $n_2$, i.e., $n = m_i \cdot n_i$ for some $m_i$. Let $T_i' = \mathsf{exp}(T_i, m_i)$, i.e., the transformation of $T_i$ into a type with period $m_i \cdot n_i = n$. If $T_1' = \overline{T_2'}$, then $T_1$ and $T_2$ are RC. Otherwise $T_1$ and $T_2$ are not RC.

This approach works for any common multiple of $n_1$ and $n_2$. For simplicity, we choose $n_1 \cdot n_2$ in our Rust embedding. Choosing the least common multiple would give the minimal expansion.

*Session channels in Rust.* Fig. 16 is an extract of the Rust code that creates and checks two RC channels for AGC to communicate with its pipeline neighbours. Our framework provides a macro `new_chan!<L, R>` to create channel endpoint pairs analogous to $(\nu k)$ construct in the session calculus. This macro expands to a call to a channel creation function with a type annotation that requires a type to implement the `Compat<L: Protocol, R: Protocol>` trait. There exists such a type only if the RC relation holds on the corresponding session types. We elaborate on the implementation of this mechanism below. If session types `L` and `R` are not RC, this type annotation fails to check and the Rust compiler statically rejects the program as badly-typed. Lines 3 and 4 each create a pair of RC channels, one endpoint for repeated input and one for output.

On line 5, the `spawn` operation provided by our framework spawns a new concurrent periodic task. It takes three arguments representing: a new instance of an AGC task (created by `task_AGC` from Fig. 15), a channel set containing the appropriate endpoint of each channel required by the task, and an initial data struct. (Note, the Rust notation `|| expr` forms a closure from `expr`; it is not to be confused with parallel composition in process calculi.) The channel endpoints supplied to this `spawn` are accordingly (see the definition of the AGC task in Fig. 15)) the input endpoint for the AGC task to receive from the preceding pipeline task (HPF), and send to the next (LPF). The Rust compiler statically checks that appropriate channels are supplied.

In this way, a complete system is formed by creating the concurrent periodic tasks and session typed channels, and allocating the appropriate channel endpoints to each task. The rest of the Heart Rate Sensor pipeline is implemented similarly to the above, producing a rate-safe periodic system that can be run indefinitely.

*Checking RC via Rust trait embedding.* The static RC check performed by `new_chan` is implemented as an embedding into Rust type checking. In Rust, *traits* are sets of behaviors that types can implement, similar to the concept of a typeclass in languages like Haskell. Traits in Rust define a collection of functions available on values of implementing types. As part of type checking, the Rust compiler will determine if the types provided implement the traits required by their usage in a *trait resolution* step. A common technique in Rust is the use of *marker traits*, which have no functions but where implementing the trait is a marker of some property. In our framework, the Rust type encodings of session types (as summarized earlier) implement a shared `Protocol` marker trait. Only the types implementing `Protocol` are allowed to be used as session type encodings.

Our framework encodes the RC relation via the trait `Compat<L: Protocol, R: Protocol>`. This is a marker trait, where a type implementing `Compat<L: Protocol, R: Protocol>` indicates proof that

the corresponding session types are RC. We encode our decision procedure for RC via a collection of helper traits that combine to form an implementation of `Compat<L: Protocol, R: Protocol>`. Like typeclasses in Haskell, a Rust type can either implement a trait directly, or by implementing another trait which in turn provides an implementation. Notably, this indirection through auxillary traits allows us to transform our decision procedure into a search by the Rust compiler for a chain of trait implementations ending in `Compat<L: Protocol, R: Protocol>`. If this search succeeds and a type implements `Compat<L, R>`, then there is a corresponding execution of our decision procedure that shows the types represented by `L` and `R` are RC.

Given two types `L` and `R`, the macro asserts the existence of a type `CompatProof<L, R, ...>` that implements the `Compat<L, R>` trait. Here, `CompatProof<L, R, ...>` is a zero-sized type, meaning it has no run-time representation. It is still treated like any other type during type checking and will force the expansion of `L` and `R` during trait resolution. The type `CompatProof<L, R, ...>` can implement the `Compat<L, R>` trait via series of auxiliary traits which implement operations such as $\mathsf{exp}_t(T, m)$, substitution, and duality. The omitted type params (...) above must be filled by zero-sized types which encode the steps of the decision procedure for RC. The procedure is fixed, so the macro always expands to the same types with `L` and `R` inlined appropriately. Verifying these types implement the corresponding traits acts as an execution of the encoded decision procedure. Interested readers can find the full encoding in the corresponding software artifact.

*Summary of the Rust framework.* Our framework puts all the above pieces together to statically ensure rate-safe session programming – i.e., freedom from session communication and rate errors (Theorem 5.9.1) – through native Rust type checking. Systems are constructed in two phases:

**Implement individual tasks.** The programmer encodes our session types as Rust types, and implements the individual periodic tasks that operate on session-typed channels. Static Rust type checking ensures that I/O actions performed on a channel variable/instance correspond to the associated session type. Moreover, a task implementation does not permit any form of folding/unfolding or expansion of the associated session type – the period of the task derives solely from its declared type. Correspondingly, the number of I/O actions on a given channel in a task implementation must be the same as the number of syntactic I/O prefixes in the associated type. This restriction corresponds to the fact that our formal typing system does not allow rule (T-Fold) to be applied to the *static* definitions of periodic recursions. The rule (T-Prd) allows exactly one unfolding to be used to type the task body, and (T-Fold) applies only to *run-time* periodic types. Rust type checking ensures that each channel variable/instance in a task is used exactly once.

**Compose tasks using RC channels.** The programmer creates pairs of channel endpoints using the declared session types. Static Rust type checking ensures the declared session types in each pair of channel endpoints are RC. They then instantiate a set of periodic tasks, compose the tasks by giving the appropriate channel endpoints to each task, and spawn the tasks to run concurrently. The RC check performed by `new_chan` is the only construct in the framework that allows expansion and unfolding of periodic types. In contrast, the `task` definition allows exactly one unfolding, where the body is typed against the once unfolded session type as per rule (T-Prd).

*6.1.2 Composable DSP Blocks With Generic Sessions.* An advantage of building within existing Rust type system is we can reuse Rust type checking features. Importantly, we get the entire Rust generics system. We can use this for making tasks generic over values and even periods. This is useful for creating a reusable task definition, as some tasks only care about ratios of rates, not the concrete period. We use this to define a library of useful generic tasks that perform common signal processing and stream manipulation operations. The function `printer` in Fig. 17 shows a task that prints values — with any period and type desired.

```
fn printer<V, N>() ->
    Task![N; (); Prd<N, P, Recv<V, P>>]
where V: RustSend+Debug, N: Nat+NonZero {
    task![_; c : Recv<V, P> => {
        let (v, c) = c.recv();
        println!("{:?}", v);
        (chans!(c), ())}]
}

type ProdOut = Prd<TWO, P, Send<i32, P>>;
type PairIn = Prd<FOUR, P,
                Recv<i32, Recv<i32, P>>>;
```

```
type PairOut = Prd<FOUR, P, Send<(i32, i32), P>>;
type PrintIn = Prd<FOUR, P, Recv<(i32, i32), P>>;

fn main() {
 let (ka1, ka2) = new_chan!(ProdOut, PairIn);
 let (kb1, kb2) = new_chan!(PairOut, PrintIn);

 let  p1 = spawn!(counter(), || chans!(ka1), || 1);
 let _p2 = spawn!(pair(), || chans!(ka2, kb1), || ());
 let _p3 = spawn!(printer(), || chans!(kb2), || ());
 let _ = p1.join(); // Run forever
}
```

Fig. 17. The Generic Printer.

We can then instantiate these blocks in a larger system, for example, Fig. 17. We observe three advantages of the type encoding. First, as protocols are simply Rust types, we can define type aliases for convenience. Thus we name the various sessions e.g. `PrintIn`. Second, we do not need to specify the type of value the printer will be receiving when we instantiate it. The Rust compiler is able to infer the value type from the type we gave for the channel. Since the channel is passing `(i32, i32)`, that must be the value type `V` of the printer. Third, this type inference even extends to task periods. We defined `printer` to have a generic period `N`. Because the period of a process must match the period of all its channel endpoints, Rust can infer the printer must have a period of 4. We specified in the channel type that `PrintIn` had period `FOUR`, so too must the printer. By creating an encoding of sessions as Rust types, we find that these programs only need channel type annotations. From there, Rust can infer the rest — even when the blocks are generic.

### 6.2 Examples and Features

Table 1 lists application protocols covering various features of our framework, including a small set of reusable DSP blocks used to implement them. Most of the examples feature heterogeneous periods and thus exercise full RC including expansions; some feature simpler protocols that require only $duality_\omega$ to type. "Generic" means the protocols and processes are defined with generic periods or sorts. Each example is classified as a pipeline (P) with one-way communications (common in our target domains), or a system with bidirectional (B) communications. In addition to basic indefinite

Table 1. Example protocols expressed using rate-based session types in our framework.

| Example | Domain | $duality_\omega$ or full-**RC** | Generic | **P**ipe. or **B**idi. | **I**ndef., **B**ranch. or (Pot.) **T**erm. |
|---|---|---|---|---|---|
| PineTime Heart Rate [Pine64 2019] | Wearables | $duality_\omega$ | ✓ | P | I |
| Watch Rate Transform | Wearables | RC | ✓ | P | I |
| Periodic Bluetooth [Bluetooth SIG 2023] | IoT Sensors | RC | | **B** | I |
| Keyless Entry [Wouters et al. 2019] | Embedded | RC | | **B** | I, B |
| Gravity [Android 2009] | Android | RC | | P | I |
| Linear Acceleration [Android 2009] | Android | RC | | P | I |
| Orientation [Android 2009] | Android | RC | | P | I |
| Wake Word [Warden 2020] | Audio | RC | | P | I |
| IIR Filter [Collins et al. 2018] | DSP | $duality_\omega$ | ✓ | P | I |
| File Reader | Utility | $duality_\omega$ | | P | I, B, T |
| DSP Blocks | DSP | RC | ✓ | P | I |

(I) periodic processes, our framework also supports (binary) labeled branching (B), and (potentially) terminating (T) processes where a branch may offer both a recursive and terminating case. The protocols in our examples are safely RC. In the case of a rate mismatch error, our framework rejects the program as badly-typed. For example, these types cause a static type error if given to `new_chan!`:

```
type P2X = Prd<TWO,   P, Send<i32, Send<i32, Send<i32, P>>>>;
type C3  = Prd<THREE, P, Recv<i32, Recv<i32, Recv<i32, P>>>>;
```

### 6.3 Distribution Across Devices

Our theory represents processes executing concurrently subject to a single global measure of time. With sufficient support from networking libraries, our Rust crate could be extended to cover both communications between threads on a single device and communications across the network between devices. Embedded systems typically impose constraints on the networking hardware and protocol that make this possible. To illustrate this, let's consider an example. In wireless systems, rate compatibility is necessary for statically specified power-saving sleep intervals to be safe. Bluetooth Low Energy (BLE) [Bluetooth SIG 2023] contains a feature known as "periodic advertisements" which allows receivers to go into a sleep mode between communications. During the sleep interval, the receiver stops listening for messages and will miss any communications sent to it. For this to work properly, the receiver must know when to expect the next message and wake up in time to receive it. The BLE standard specifies the exact parameters of the radio communications channel. This means messages take a fixed, known time in transit. Thus clock synchronization with accuracy is possible and messages are physically guaranteed to arrive in order. Our theory models the passage of time as a single global clock. The periodic advertisement protocol in BLE faithfully implements this model by requiring all participants to maintain an accurate and stable local time reference and injecting timekeeping communications at the beginning of each period to keep them synchronized and prevent drift. In BLE, a rate error would mean the receiver and transmitter disagree on when the next message is coming. In our theory, a rate error results in a process term getting stuck on a deadline. In a BLE system, that error would manifest as communications occurring while the receiver was asleep. These communications would be ignored and thus lost, comparable to a hardware failure.

## 7 RELATED WORK

**Session types** Our system is based on the theory of binary session types. First proposed in [Honda 1993] and [Honda et al. 1998], we utilize the refined formulation and proofs given in [Yoshida and Vasconcelos 2007]. The important difference from the classic approach to binary session types is our treatment of recursion. Typically, the recursive construct $\mu t.T$ is viewed as a fixed point. Crucially, the types $\mu t.!\texttt{int}.t$ and $\mu t.!\texttt{int}.!\texttt{int}.t$ would be considered equivalent. This corresponds to an equi-recursive view of typing in which these types actually represent infinite trees. We depart from this approach and do not treat $\omega_n^d t.T$ as a fixed point. This is critical to our approach for two reasons. First, we want to distinguish a type $\omega_n^d t.!\texttt{int}.t$ as different from $\omega_n^d t.!\texttt{int}.!\texttt{int}.t$. These types have different rates, as they perform a different number of communications over the same period. Second, viewing these types as infinite trees loses the period annotation on the $\omega$. Thus our approach to reasoning about the periods of processes mandates an iso-recursive approach.

For the systems introduced in Section 2, we examine pipelines and sensor fusion problems. For these applications, the communication model is acyclic. Thus for these systems, we are able to use multiple binary channels without encountering deadlocks. In future work, we plan to support cyclic process topologies with deadlock freedom in a multiparty session type system [Bettini et al. 2008; Honda et al. 2008; Scalas and Yoshida 2019].

**Timed session types.** Existing work has explored adding timing to session types [Bocchi et al. 2019, 2014]. Similar to these approaches, we separate delays from computations. The critical difference lies in where we encode timing. Prior work [Bocchi et al. 2014] ties timing to communication via the addition of timing constraints over local clocks. Instead, we attach timing to a control flow construct – namely the periodic construct. Timing constraints are relative to a global clock, mirroring the approach taken in real-time system design [Liu and Layland 1973].

Prior work adding timing to binary types [Bocchi et al. 2019] introduces a subtyping relation over timed sessions. This subtyping relation allows faster processes to replace slower ones. As this model is asynchronous, it is viewed as safe for a message to be enqueued earlier than necessary. When applied to indefinite processes, however, this approach breaks down. While it would maintain progress, a new issue arises: unbounded channel size. If we allowed a process with period one, $\omega_1 t.!\texttt{int}.t$, to replace one with period two, $\omega_2 t.!\texttt{int}.t$, messages would be produced at double the expected rate. In an asynchronous system, this would mean the channel queue would grow without bound, as one message is removed for every two inserted. In a synchronous model, this becomes even more problematic. Now the faster process is blocked, waiting for the slower one. Thus it is unable to meet its stated deadline, leading to an error. While our system can prevent these timing errors and corresponds to bounded channels, we lose the ability to model communication latency. We operate under a model where all communications and computations are assumed to have zero cost. The reintroduction of delay in processes used in prior work [Bocchi et al. 2019, 2014] would allow more expressive cost models. This is important when the computation costs between communications mean an expanded process may have different timing behavior.

Work on intuitionistic linear logic based session types [Balzer and Pfenning 2017; Caires and Pfenning 2010] has also considered notions of timing. [Das et al. 2018] extends session types with temporal modalities. This system explicitly uses a next modality to specify the exact time for communications. Applied to our domain, this would require a statically known schedule to be encoded in the type. In our system, we define the notion of deadline and allow communications to happen any time up to the deadline. Allowing this flexibility enables reuse of a task between different systems without defining new types for it. We leverage this in examples from Sec. 6.

**Session types in Rust.** We extend the original formulation of Rust session types presented in [Jespersen et al. 2015; Munksgaard and Jespersen 2015]. Ferrite [Chen et al. 2022] is a more recent session type implementation based on an encoding of intuitionistic linear logic session types in Rust's type system. This linear logic formulation lacks an explicit notion of duality. Instead, channels are oriented along client-server roles. Without a duality relation, it is unclear where to add compatibility. In contrast, classical linear logic session types [Wadler 2012] do utilize a duality relation and may provide some initial insight. We consider this an interesting area for future work.

## 8 CONCLUSION AND FUTURE WORK

We introduce a variant of session types capable of tracking communication rates — an important concept in many IoT signal processing domains. A new timed process calculus with periodic tasks allows us to define rate errors. We then provide a corresponding type system and show it can statically prevent such errors by rejecting programs with rate mismatches. We consider extending rate-based reasoning to multiparty session types an important area for future work. The corresponding software artifact, including Rust crate, is available from ACM DL [Iraci et al. 2023].

## ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant No. SHF 2211997, SHF 1749539, and the Graduate Research Fellowships Program.

## REFERENCES


Kazi Masudul Alam and Abdulmotaleb El Saddik. 2017. C2PS: A Digital Twin Architecture Reference Model for the Cloud-Based Cyber-Physical Systems. *IEEE Access* 5 (2017), 2050–2062. https://doi.org/10.1109/ACCESS.2017.2657006

Android. 2009. Motion Sensors. https://developer.android.com/guide/topics/sensors/sensors_motion

Stephanie Balzer and Frank Pfenning. 2017. Manifest Sharing with Session Types. *Proc. ACM Program. Lang.* 1, ICFP, Article 37 (aug 2017), 29 pages. https://doi.org/10.1145/3110281

Lorenzo Bettini, Mario Coppo, Loris D'Antoni, Marco De Luca, Mariangiola Dezani-Ciancaglini, and Nobuko Yoshida. 2008. Global Progress in Dynamically Interleaved Multiparty Sessions. In *CONCUR 2008 - Concurrency Theory, 19th International Conference, CONCUR 2008, Toronto, Canada, August 19-22, 2008. Proceedings (Lecture Notes in Computer Science, Vol. 5201)*, Franck van Breugel and Marsha Chechik (Eds.). Springer, 418–433. https://doi.org/10.1007/978-3-540-85361-9_33

Bluetooth SIG 2023. *Bluetooth Core Specification.* Bluetooth SIG. 5.4.

Laura Bocchi, Maurizio Murgia, Vasco Thudichum Vasconcelos, and Nobuko Yoshida. 2019. Asynchronous Timed Session Types. In *Programming Languages and Systems*, Luís Caires (Ed.). Springer International Publishing, Cham, 583–610.

Laura Bocchi, Weizhen Yang, and Nobuko Yoshida. 2014. Timed Multiparty Session Types. In *CONCUR 2014 – Concurrency Theory*, Paolo Baldan and Daniele Gorla (Eds.). Vol. 8704. Springer Berlin Heidelberg, Berlin, Heidelberg, 419–434. https://doi.org/10.1007/978-3-662-44584-6_29 Series Title: Lecture Notes in Computer Science.

Richard R. Brooks and S. S. Iyengar. 1998. *Multi-Sensor Fusion: Fundamentals and Applications with Software.* Prentice-Hall, Inc., USA.

Luís Caires and Frank Pfenning. 2010. Session Types as Intuitionistic Linear Propositions. In *CONCUR 2010 - Concurrency Theory*, Paul Gastin and François Laroussinie (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 222–236.

Ruo Fei Chen, Stephanie Balzer, and Bernardo Toninho. 2022. Ferrite: A Judgmental Embedding of Session Types in Rust. In *36th European Conference on Object-Oriented Programming (ECOOP 2022) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 222)*, Karim Ali and Jan Vitek (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 22:1–22:28. https://doi.org/10.4230/LIPIcs.ECOOP.2022.22

Travis F. Collins, Robin Getz, Di Pu, and Alexander M. Wyglinski. 2018. *Software-Defined Radio for Engineers.* Artech House. https://www.analog.com/en/education/education-library/software-defined-radio-for-engineers.html

Ankush Das, Jan Hoffmann, and Frank Pfenning. 2018. Parallel Complexity Analysis with Temporal Session Types. *Proc. ACM Program. Lang.* 2, ICFP, Article 91 (jul 2018), 30 pages. https://doi.org/10.1145/3236786

FreeRTOS. 2003. *FreeRTOS Kernel.* Amazon Web Services. https://www.freertos.org/

Simon Gay and Malcolm Hole. 2005. Subtyping for session types in the pi calculus. *Acta Informatica* 42, 2 (Nov. 2005), 191–225. https://doi.org/10.1007/s00236-005-0177-z

Kohei Honda. 1993. Types for dyadic interaction. In *CONCUR'93*, Eike Best (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 509–523.

Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. 1998. Language primitives and type discipline for structured communication-based programming. In *Programming Languages and Systems*, Gerhard Goos, Juris Hartmanis, Jan van Leeuwen, and Chris Hankin (Eds.). Vol. 1381. Springer Berlin Heidelberg, Berlin, Heidelberg, 122–138. https://doi.org/10.1007/BFb0053567 Series Title: Lecture Notes in Computer Science.

Kohei Honda, Nobuko Yoshida, and Marco Carbone. 2008. Multiparty Asynchronous Session Types. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Francisco, California, USA) *(POPL '08)*. Association for Computing Machinery, New York, NY, USA, 273–284. https://doi.org/10.1145/1328438.1328472

Grant Iraci, Cheng-En Chuang, Raymond Hu, and Lukasz Ziarek. 2023. *Rate Based Session Types: Rust Implementation.* https://doi.org/10.1145/3580415

Thomas Bracht Laumann Jespersen, Philip Munksgaard, and Ken Friis Larsen. 2015. Session Types for Rust. In *Proceedings of the 11th ACM SIGPLAN Workshop on Generic Programming* (Vancouver, BC, Canada) *(WGP 2015)*. Association for Computing Machinery, New York, NY, USA, 13–22. https://doi.org/10.1145/2808098.2808100

Jie Lin, Wei Yu, Nan Zhang, Xinyu Yang, Hanlin Zhang, and Wei Zhao. 2017. A Survey on Internet of Things: Architecture, Enabling Technologies, Security and Privacy, and Applications. *IEEE Internet of Things Journal* 4, 5 (2017), 1125–1142. https://doi.org/10.1109/JIOT.2017.2683200

C. L. Liu and James W. Layland. 1973. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *J. ACM* 20, 1 (jan 1973), 46?61. https://doi.org/10.1145/321738.321743

Sanjay Madria, Vimal Kumar, and Rashmi Dalvi. 2014. Sensor Cloud: A Cloud of Virtual Sensors. *IEEE Software* 31, 2 (2014), 70–77. https://doi.org/10.1109/MS.2013.141

Umberto Maniscalco and Riccardo Rizzo. 2017. A virtual layer of measure based on soft sensors. *Journal of Ambient Intelligence and Humanized Computing* 8, 1 (2017), 69–78. https://doi.org/10.1007/s12652-016-0350-y

Dominik Martin, Niklas Kühl, and Gerhard Satzger. 2021. Virtual Sensors. *Business & Information Systems Engineering* 63, 3 (2021), 315–323. https://doi.org/10.1007/s12599-021-00689-w

Philip Munksgaard and Thomas Bracht Laumann Jespersen. 2015. *Practical Session Types in Rust.* Master's thesis. Department of Computer Science, University of Copenhagen.

Benjamin C. Pierce. 2002. . The MIT Press, Chapter 21.11, 311–312.

Pine64. 2019. *PineTime.* Pine64. https://www.pine64.org/pinetime/

QNX. 2001. *QNX Neutrino.* QNX. https://www.qnx.com/products/intl/neutrino_rtos/

Alceste Scalas and Nobuko Yoshida. 2019. Less is more: multiparty session types revisited. *Proc. ACM Program. Lang.* 3, POPL (2019), 30:1–30:29. https://doi.org/10.1145/3290343

Fei Tao, He Zhang, Ang Liu, and A. Y. C. Nee. 2019. Digital Twin in Industry: State-of-the-Art. *IEEE Transactions on Industrial Informatics* 15, 4 (2019), 2405–2415. https://doi.org/10.1109/TII.2018.2873186

Agnes Tegen, Paul Davidsson, Radu-Casian Mihailescu, and Jan A. Persson. 2019. Collaborative Sensing with Interactive Learning using Dynamic Intelligent Virtual Sensors. *Sensors* 19, 3 (2019). https://doi.org/10.3390/s19030477

Antonio Vallecillo, Vasco T. Vasconcelos, and António Ravara. 2003. Typing the Behavior of Objects and Components using Session Types. *Electronic Notes in Theoretical Computer Science* 68, 3 (2003), 439–456. https://doi.org/10.1016/S1571-0661(05)80382-2 Foclasa 2002, Foundations of Coordination Languages and Software Architectures (Satellite Workshop of CONCUR 2002).

Antonio Vallecillo, Vasco T Vasconcelos, and Antonio Ravara. 2006. Typing the Behavior of Software Components using Session Types. (2006), 16.

Philip Wadler. 2012. Propositions as Sessions. *SIGPLAN Not.* 47, 9 (sep 2012), 273–286. https://doi.org/10.1145/2398856.2364568

Pete Warden. 2020. *TinyML: Machine learning with tensorflow on Arduino, and ultra-low Power Micro-controllers.* O'REILLY MEDIA.

Patrick Wechselberger, Patrick Sagmeister, and Christoph Herwig. 2013. Real-time estimation of biomass and specific growth rate in physiologically variable recombinant fed-batch processes. *Bioprocess and Biosystems Engineering* 36, 9 (2013), 1205–1218. https://doi.org/10.1007/s00449-012-0848-4

Lennert Wouters, Eduard Marin, Tomer Ashur, Benedikt Gierlichs, and Bart Preneel. 2019. Fast, Furious and Insecure: Passive Keyless Entry and Start Systems in Modern Supercars. *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2019, 3 (May 2019), 66–85. https://doi.org/10.13154/tches.v2019.i3.66-85

Jennifer Yick, Biswanath Mukherjee, and Dipak Ghosal. 2008. Wireless sensor network survey. *Computer Networks* 52, 12 (2008), 2292–2330. https://doi.org/10.1016/j.comnet.2008.04.002

Nobuko Yoshida and Vasco T. Vasconcelos. 2007. Language Primitives and Type Discipline for Structured Communication-Based Programming Revisited: Two Systems for Higher-Order Session Communication. *Electronic Notes in Theoretical Computer Science* 171, 4 (July 2007), 73–93. https://doi.org/10.1016/j.entcs.2007.02.056

Received 2023-04-14; accepted 2023-08-27