Semantic Logical Relations for Timed Message-Passing Protocols

YUE YAO, Carnegie Mellon University, USA
GRANT IRACI, University at Buffalo, USA
CHENG-EN CHUANG, University at Buffalo, USA
STEPHANIE BALZER, Carnegie Mellon University, USA
LUKASZ ZIAREK, University at Buffalo, USA

Many of today's message-passing systems not only require messages to be exchanged in a certain order but also to happen at a certain *time* or within a certain *time window*. Such correctness conditions are particularly prominent in Internet of Things (IoT) and real-time systems applications, which interface with hardware devices that come with inherent timing constraints. Verifying compliance of such systems with the intended *timed protocol* is challenged by their *heterogeneity*—ruling out any verification method that relies on the system to be implemented in one common language, let alone in a high-level and typed programming language. To address this challenge, this paper contributes a *logical relation* to verify that its inhabitants (the applications and hardware devices to be proved correct) comply with the given timed protocol. To cater to the systems' heterogeneity, the logical relation is entirely *semantic*, lifting the requirement that its inhabitants are syntactically well-typed. A semantic approach enables two modes of use of the logical relation for program verification: *(i) once-and-for-all* verification of an *arbitrary* well-typed application, given a type system, and *(ii) per-instance* verification of a specific application / hardware device (*a.k.a.,* foreign code). To facilitate mode *(i)*, the paper develops a refinement type system for expressing timed message-passing protocols and proves that any well-typed program inhabits the logical relation (fundamental theorem). A type checker for the refinement type system has been implemented in Rust, using an SMT solver to check satisfiability of timing constraints. Then, the paper demonstrates both modes of use based on a small case study of a smart home system for monitoring air quality, consisting of a controller application and various environment sensors.

CCS Concepts: • **Theory of computation** → **Linear logic**; **Type theory**; *Process calculi*.

Additional Key Words and Phrases: Semantic logical relations, Timed message-passing protocols, Instant-based temporal session types, Intuitionistic linear logic

ACM Reference Format:
Yue Yao, Grant Iraci, Cheng-En Chuang, Stephanie Balzer, and Lukasz Ziarek. 2025. Semantic Logical Relations for Timed Message-Passing Protocols. *Proc. ACM Program. Lang.* 9, POPL, Article 59 (January 2025), 32 pages. https://doi.org/10.1145/3704895

1 Introduction

The computing landscape has gradually been shifting to applications targeting distributed and heterogeneous systems, including Internet of Things (IoT) and real-time systems applications. Such applications are predominantly *concurrent*, employ *message-passing*, and often interface with *foreign*

Authors' Contact Information: Yue Yao, Carnegie Mellon University, Pittsburgh, USA, yueyao@cs.cmu.edu; Grant Iraci, University at Buffalo, Buffalo, USA, grantira@buffalo.edu; Cheng-En Chuang, University at Buffalo, Buffalo, USA, chengenc@buffalo.edu; Stephanie Balzer, Carnegie Mellon University, Pittsburgh, USA, balzers@cs.cmu.edu; Lukasz Ziarek, University at Buffalo, Buffalo, USA, lziarek@buffalo.edu.

ACM 2475-1421/2025/1-ART59
https://doi.org/10.1145/3704895

code–code inaccessible to the application developer. For example, consider a smart home system for monitoring air quality. The controller of the system, an IoT application, bases its decision on readings it receives from various environment sensors (*e.g.,* BME680 [Bosch 2024]), spread across the home, measuring the surrounding air temperature, humidity, pressure, *etc.* Such sensors are hardware devices that the developer of the controller must interact with through a *protocol* defined by the manufacturer in a specification ("datasheet"). Another characteristic of such applications is their need to comply with the *timing constraints* dictated by the hardware devices' protocol. For example, when measuring air quality, the BME680 sensor can only provide readings after heating up an internal component for 30ms, after which it requires an additional 20ms to cool down. Any application that uses the sensor must account for its protocol and timing requirements.

This naturally leads to the *research question* that we address in this paper:

> *How to enable application developers to write timed message-passing programs that comply with the timing constraints of the underlying hardware devices' protocols?*

This question raises the following *challenges*; the techniques to overcome them are the core contribution of our paper:

(1) distillation of a common *specification language* for *timed message-passing protocols* to prescribe the sequencing and timing of interactions between applications and devices;
(2) distillation of a common *operational model*, capturing the execution behavior *both* of running applications and devices;
(3) development of a *compositional verification framework*, allowing the proof that the operational models of the applications and devices satisfy their specifications. Compositionality (*a.k.a.,* modularity) allows applications and devices to be verified separately and guarantees that they can be combined to a verified whole (without the need to re-verify the whole).

To address Challenge 1, we use *types* as a specification language. In particular, we build on the types developed for process calculi [Igarashi and Kobayashi 2001; Kobayashi 1997] and specifically on *session types* [Honda 1993; Honda et al. 1998, 2008]. Session types are behavioral types that prescribe the protocols of message-passing concurrent programs and enjoy strong theoretical foundations, including a Curry-Howard correspondence between the session-typed π-calculus and linear logic [Caires and Pfenning 2010; Kokke et al. 2019; Lindley and Morris 2015; Toninho 2015; Toninho et al. 2013; Wadler 2012]. The connection to linear logic endows programming languages developed for logic-based session types with various desirable properties, such as protocol adherence and deadlock freedom. The latter ensures global progress and is a result of linearity, which imposes a tree structure on the runtime configurations of processes.

Among these logic-based session types, we chose the family based on *intuitionistic linear logic* [Caires and Pfenning 2010; Toninho 2015; Toninho et al. 2013], which distinguishes the *provider* from the *client* side of a protocol. This distinction naturally accommodates the separation between hardware devices (*i.e.,* providers) and application programs (*i.e.,* clients), present in our target domain. To facilitate expression of timing constraints, we extend intuitionistic linear logic session types (ILLST) with *temporal predicates*, resulting in *timed intuitionistic linear logic session types* (TILLST). TILLST adopts an *instant-based* model of time [SEP 1999], treating points in time (*i.e.,* instants) as primary notions, and allows temporal predicates to quantify over such points in time to prescribe at which instant(s) communications must happen relative to a *global clock*. TILLST sets itself apart from session types based on timed automata [Bartoletti et al. 2017; Bocchi et al. 2019, 2014] not only in terms of its logical foundation, but also in the underlying model of time. Automata-based session types come equipped with local clocks, which require explicit synchronization (*i.e.,* clock resets) to simulate a global notion of time.

To address Challenge 2, we use a *labelled transition system (LTS)* [Milner 1980, 1999; Sangiorgi and Walker 2001] to express how applications and hardware devices exchange messages and thus compute. The fundamental notion underlying an LTS is the one of a transition labelled with an *action*, conveying the readiness of a participating entity to engage in an exchange. Actions range over outputs (*i.e.,* sends), inputs (*i.e.,* receives), and the empty action. An empty action denotes an actual computation, *i.e.,* a *reduction*, which happens when two entities with complementary, *i.e., dual*, actions meet and exchange a message as a result. We chose an LTS as our operational model because it is agnostic of the literal syntax used to represent communicating entities, catering to the heterogeneity of our target domain. Moreover, the duality of actions mirrors the provider-client distinction in ILLST: when a provider is ready to output, its client will (eventually) be ready to input, and vice versa. To accommodate timing considerations, our LTS denotes the instant at which an entity is ready to engage in an exchange, in addition to the action. We refer to the resulting LTS as *timed LTS*.

To address Challenge 3, we use *logical relations* [Girard 1972; Pitts and Stark 1998; Plotkin 1973; Statman 1985; Tait 1967] as our verification framework. Logical relations are a device to prescribe the properties of valid programs in terms of their *computational behavior* and are defined by considering the types of the underlying language. As such, a logical relation can be viewed as defining "inhabitance" of valid terms in a type. Logical relations enjoy *compositionality*, ensuring that any two inhabitants can be composed to a compound inhabitant, as dictated by the type structure. An important characteristic of logical relations is their *constructive* standpoint: logical relations define the semantics of a program in terms of how it runs. This insight fueled the *semantic typing* approach [Constable et al. 1986; Martin-Löf 1982; Timany et al. 2024], which lifts the requirement that terms inhabiting the logical relation ought to be (syntactically) well-typed. Such a semantic approach allows proving inhabitance not only of well-typed terms—via the "fundamental theorem" of the logical relation—but also of untyped terms, provided they exhibit the computational behavior prescribed by the logical relation. Because of this property, cross-language logical relations have been successfully employed, for example, for compiler correctness proofs [Benton and Hur 2009; Chlipala 2007; Minamide et al. 1996] and soundness of language interoperability [Patterson et al. 2022].

We chose logical relations as our verification framework precisely because of semantic typing. The logical relation that we contribute enables semantic typing of *both* the applications and devices in our target domain and is the core contribution of this paper. Next, we summarize the **(1)** key *features* of our logical relation and then highlight **(2)** two *modes of use* of our logical relation: **(a)** once-and-for-all verification of an arbitrary well-typed application, given a type system, and **(b)** per-instance verification of a specific application / hardware device (*a.k.a.,* foreign code). Because logical relations are compositional, both modes of use work *synergistically*, allowing us to combine components verified by either method to a verified whole. The subsequent sections should be read as a "teaser", conveying the main ideas, which we will formally develop in §3–§5.

1.1 A Timed Semantic Session Logical Relation (TSSLR)

Our logical relation is defined by structural induction on timed intuitionistic linear logic session types (TILLST), our specification language for timed message-passing protocols (Challenge 1), and prescribes the computational behavior of its inhabitants in terms of a timed LTS, our operational model for applications and devices in our target domain (Challenge 2). We refer to our logical relation as *timed semantic session logical relation* (TSSLR), to convey its purely semantic nature and grounding in TILLST. To the best of our knowledge, TSSLR is the first logical relation facilitating the verification of timing constraints for message-passing protocols. It is also the first *entirely* semantic logical relation for session types; prior logical relations for session types [Balzer et al.

2023; Caires et al. 2013; Derakhshan et al. 2021, 2024; DeYoung et al. 2020; Pérez et al. 2012, 2014; van den Heuvel et al. 2024] demand inhabitants to be syntactically well-typed. Besides facilitating *semantic typing* for *timed message-passing protocols*, TSSLR has the following unique features:

1.1.1 Nameless Families of Configurations. The handling of names—identifiers such as channels and locations—can become tedious in formal developments. All too often, we end up either renaming existing names, for example upon receiving a channel, or maintaining equivalence classes of names. These strategies are not only cumbersome but also error-prone. But even more so, formal developments usually do not even depend on a *particular* choice of a name! Linear logic can come to help, because it guarantees that a name is only shared between *two* runtime entities (*e.g.,* sensor and controller), facilitating local reasoning about names. Intuitionistic linear logic—due to its provider-client distinction–moreover allows a provider (*e.g.,* sensor) to be *polymorphic* in the name it is referred to by its client (*e.g.,* controller). Our development takes advantage of these properties and introduces the notion of a *nameless family of configurations*. Configurations are the terms inhabiting our logical relation, nameless families of configurations are polymorphic in the choice of a name by their clients. Our TSSLR defines inhabitance in terms of nameless families of configurations to accommodate arbitrary choices of names.

1.1.2 Computable Trajectories. Labelled transition systems (LTS) [Milner 1980, 1999; Sangiorgi and Walker 2001] describe concurrent interactions in a *local* way, isolating a particular entity that is ready to engage in an action, while "framing off" entities unaffected by the action. The understanding is that any number of ready entities with mutually complementary actions can reduce concurrently. For timed message-passing protocols, however, the term "concurrent" is too liberal, because it does not prescribe which reductions among the concurrent ones must happen *simultaneously*. Transitions in our timed LTS therefore are additionally annotated with the *instant* at which an exchange may happen. To complement this local description of a potential computation with a *global* perspective, we introduce the notion of a trajectory. A *trajectory* is a function that returns for each instant in time the configuration of all entities at that particular instant. We can thus think of a trajectory as a description of *how a configuration evolves over time*. To assert that a trajectory is the result of applying a sequence of timed LTS reductions, which validates that trajectory, we introduce the notion of a computable trajectory. A *computable trajectory* is a pair, consisting of the trajectory and a validating sequence of reductions. In case of simultaneous reductions, there exists a validating sequence of reductions for each permutation of simultaneous reductions, but any of these suffices to assert computability. The value of the notion of a trajectory is precisely that it "collapses" all simultaneous local LTS reductions to one global reduction step. Our TSSLR is phrased in terms of computable trajectories and is thus polymorphic in the equivalence class of instantaneous LTS reduction sequences for a configuration's trajectory.

1.1.3 Algebra for Computable Trajectories. With the definition of computable trajectories in our hands, it is convenient to define *operations* on computable trajectories, resulting in an *algebra* for computable trajectories. To gather an intuition for what operations may be suitable, it is helpful to remind ourselves that a computable trajectory essentially describes how a configuration evolves over time. With that intuition in mind, it is sensible to expect that trajectories can be: **(a)** *interleaved*, describing the evolution of the concurrent composition of two configurations; **(b)** *partitioned*, to truncate a trajectory relative to a given instant; and **(c)** *concatenated*, to sequentially compose two trajectories. As we will see in §4.2, our computable trajectory algebra facilitates an elegant definition and proof of the usual forward and backwards closure properties of logical relations, generalized to account for the passage of time.

1.2 Two Modes of Use of TSSLR

Thanks to its semantic formulation, our TSSLR facilitates program verification in two ways:

(1) *once-and-for-all* verification of an *arbitrary* well-typed application, given a type system;
(2) *per-instance* verification of a *specific* application / hardware device (*a.k.a.,* foreign code).

Mode 1 requires development of a suitable type system that is strong enough to ensure that any well-typed term computes as prescribed by the logical relation. This proof, *i.e.,* that any well-typed term inhabits the logical relation, is referred to as the *fundamental theorem* of the logical relation (FTLR). The benefit of Mode 1 is, in a sense, its "economy of scale": By carrying out, once and for all, a difficult proof (proof of the FTLR), per-program verification reduces to a type checking problem. If a decidable type checking algorithm exists, then program verification becomes automatic.

Type systems are by design recursively enumerable and thus never complete with respect to the intended semantics (*i.e.,* inhabitance does not necessarily imply well-typedness). As a result, they may reject perfectly good programs. This is where Mode 2 comes to help. Here, inhabitance of a term is proved directly, without a type system as an intermediary. The benefit of Mode 2 is its impartiality: Any "computational object" can be certified, as long as it can be shown to compute as prescribed by the logical relation. Mode 2 is therefore indispensable to the verification of systems in our application domain.

1.2.1 Mode of Use 1: Refinement Type System for TILLST. To facilitate Mode of Use 1 of our TSSLR, we contribute a refinement type system for TILLST. The dependency of TILLST types on temporal predicates manifests in the typing judgment of our type system: a term is typed relative to a context of temporal propositions, in addition to the usual variable context. Temporal dependencies between communications are expressed using temporal variables, whose free occurrences are collected in a separate context as well. To guarantee that any term that has a valid derivation using our type system also inhabits our TSSLR, we prove the corresponding *fundamental theorem.*

We have implemented a *type checker* for our type system. The type checker is implemented in Rust and uses an SMT solver to check satisfiability of temporal predicates. Type checking is thus incomplete; our benchmark suite of examples, however, type checks successfully.

1.2.2 Mode of Use 2: Per-Device Inhabitance Proof. We illustrate Mode of Use 2 of our TSSLR, by providing a proof that the BME680 environment sensor inhabits our TSSLR. To carry out this proof, we translate the BME680's specification [Bosch 2024] given by the manufacturer to a corresponding TILLST type as well as timed automaton, a term that can stepped using our timed LTS. As we show in §5, this translation is straightforward because the manufacturer's datasheet effectively defines a timed automaton.

1.2.3 Mode of Use 1 + Mode of Use 2: Whole System Verification. A semantic logical relation really comes to shine when combining both modes of use. For example, given the timed protocol as a TILLST type, we can develop the controller of our smart home device using our refinement language and prove it correct using our type checker (Mode of Use 1). Then, we compose our controller with the BME680 sensor, certified to be correct by Mode of Use 2. Compositionality of the logical relations method guarantees correctness of the entire system as a result. As we will see in §4.2, compositionality is guaranteed by the statement of the fundamental theorem, which takes an open well-typed term (*e.g.,* controller), closes it with values assumed to inhabit the logical relation (*e.g.,* sensor), and asserts inhabitance of the resulting closed term (*e.g.,* sensor + controller). Because the definition of our TSSLR does not require its inhabitants to be well-typed, the substituted values (*e.g.,* sensor) do not have to be well-typed, accommodating foreign code.

Our example of a smart home device also showcases that semantic logical relations are *synergistic* with other verification methods. Although we provide ourselves a proof of inhabitance of the BME680 sensor in § 5, a proof of inhabitance carried out by any other method will suffice. For example, the manufacturer could utilize the UPPAAL tool suite [Bengtsson et al. 1995] to prove that the BME680 sensor complies with the timed automaton specified [Bosch 2024] and thus assert its inhabitance in our TSSLR.

1.3 Summary and Contributions

In the remainder of this paper, we first give a motivating example (§ 2), the air quality monitoring system, detailing its specification and introducing the reader to the protocol specification language TILLST, contributed by this paper, as well as the implementation of the controller in a process language. While our motivating example is an instance of a *real-time* and (or) *embedded system*, our work can see applications beyond real-time and embedded systems; our methods and results are not specialized to those domains. Communicating systems with timing concerns also include web servers and email servers. For instance, the SMTP protocol [Klensin 2001] for email exchange prescribes timeouts in addition to message format specifications.

In § 3, we then develop our semantic logical relation TSSLR, featuring our *timed LTS*, *nameless families of configurations*, *computable trajectories*, and *algebra for computable trajectories*. The two modes of use of our logical relation are developed in § 4 and § 5, which contribute a refinement type system for TILLST to facilitate Mode of Use 1 (§ 4.1) and proof of inhabitance of the BME680 sensor in our logical relation, showcasing Mode of Use 2 (§ 5). The proof of correctness of the whole system, showcasing Mode of Use 1 + Mode of Use 2, is given in § 5, as a consequence of the proof of the fundamental theorem, given in § 4.2. § 6 details our Rust type checker implementation, § 7 comments on related work, and § 8 concludes with an outlook on future work.

Artifact and Technical Report. Our type checker for the TILLST refinement type system is available as an associated artifact. An extended version of this paper, which includes an appendix with formalization and proofs, is available as a technical report [Yao et al. 2024b].

2 Protocol Specification in TILLST

This section revisits the air quality monitoring system mentioned in § 1 and illustrates how to specify the underlying protocol using timed intuitionistic linear logic session types (TILLST) and how to implement the controller in a process language. This process language coincides with the term language of our refinement type system, which we introduce in § 4.1 and prove to inhabit our logical relation in § 4.2, for well-typed terms. § 5 completes the example by giving a representation of the environment sensors and a proof of their inhabitance in the logical relation.

2.1 Air Quality Monitoring System

Let's revisit the air quality monitoring system mentioned in § 1 in more detail. The system has three components: two environment sensors, x and y, connected to one central controller. Sensor x is placed in the bedroom and y is placed in the living room. The task of the controller is to decide in a timely manner, whether it needs to run the air-conditioner, by sending a Boolean representing its decision. To complete the task, the controller must configure both sensors and then collect data from them, according to the protocols dictated by the sensor specification. The controller should report `true` if the temperature in either room gets too high or the air quality in the bedroom degrades too much. The exact conditions are not important here. The aggregation of multiple data sources to synthesize a clearer, more informative signal is known as *sensor fusion* [Das et al. 2012].

Sensor fusion requires a controller to juggle multiple time sensitive communications, challenging to program correctly.

The sensor is modeled against the BME680 4-in-1 environment sensor [Bosch 2024]. The BME680 sensor measures surrounding temperature, humidity, pressure, and air quality. Interaction with the BME680 is via a hardware message-passing bus called I2C [Semiconductors 2021], and the sensor operates as a state machine. Initially, the sensor is in a low-power stand-by state. In this mode, the controller can send configuration messages to choose the desired measurement type. Additionally, the sensor can be set to either report continuously and periodically (*normal mode*) or to perform just one set of measurements (*forced mode*). After configuration, the sensor reports data for each enabled measurement sequentially by sending corresponding messages. It then returns to the standby state, potentially after a cooldown period.

Let's consider how we can specify the forced mode operation of the sensor in TILLST, required by our controller. We model two options that have very different timing requirements: (1) temperature only and (2) temperature followed by air quality. While the sensor measures temperature effectively instantaneously, in order to measure air quality, the sensor needs to first heat-up an internal component before taking a measurement. The heating takes 30 *ms*. After that, the sensor must cool down for 20 *ms* before it returns to standby. If both temperature and air quality data are requested (option 2), heating cannot start before the temperature results are read to prevent interference and inaccurate measurements. This means that our controller must wait an appropriate duration between temperature and air quality measurements.

The timing constraints involved in these protocols complicates programming even with just a singular sensor. The controller in our example, however, interleaves operations on both the bedroom and living room sensors. In the following sections we will see how TILLST types and processes allow us to specify and verify protocols cleanly and effectively.

2.2 Protocol Specification Language and Process Term Language

Before being in a position to specify the protocol for our example and implement the controller, we first must acquaint ourselves with timed intuitionistic linear logic session types (TILLST) and a suitable process language. TILLST enrich intuitionistic linear logic session types [Caires and Pfenning 2010; Toninho 2015; Toninho et al. 2013] with *temporal predicates*. The syntax of TILLST and the process language is given below. The most distinguishing feature of our language is the superscript $t.p$, where t is a time variable and p serves as a predicate on t. Table 1 conveys the protocol semantics. We first focus on the core process calculus constructs and discuss support for the exchange of functional values in §2.2.3.

$$
\begin{aligned}
\text{Session Types} \quad & A ::= \mathbf{1}^{t.p} \mid A_1 \otimes^{t.p} A_2 \mid A_1 \oplus^{t.p} A_2 \mid A_1 \mathbin{\&}^{t.p} A_2 \mid A_1 \multimap^{t.p} A_2 \\
\text{Time} \quad & T ::= t \mid \mathtt{init} \mid T + i \\
\text{Prop} \quad & p ::= \top \mid \bot \mid p_1 \wedge p_2 \mid p_1 \vee p_2 \mid p_1 \supset p_2 \mid T_1 = T_2 \mid T_1 \leq T_2 \\
\text{Process} \quad & P, Q ::= \mathtt{close}^{t.p} \mid \mathtt{wait}^{T} x \,;\, Q \mid \lambda^{t.p}(x : A_1)P \mid \mathtt{app}^{T} x\,(P) \,;\, Q \mid P_1 \otimes^{t.p} P_2 \\
& \quad \mid \mathtt{split}^{T}\, x \,;\, y \,.\, Q \mid \mathtt{switchL}^{t.p} \,;\, P \mid \mathtt{switchR}^{t.p} \,;\, P \\
& \quad \mid \mathtt{case}^{T}\, x \,\{Q_1 \mid Q_2\} \mid \mathtt{offer}^{t.p}\, \{P_1 \mid P_2\} \mid \mathtt{selL}^{T} x \,;\, Q \mid \mathtt{selR}^{T} x \,;\, Q \\
& \quad \mid \mathtt{fwd}^{T} x \mid \mathtt{spawn}^{T} P \,;\, x \,.\, Q
\end{aligned}
$$

2.2.1 Timed Process Term Language. Our language is a process language, where run-time processes of the form $\mathtt{proc}[a](\,P\,)$ compute by communicating with each other over named channels. Each process executes some code P called its *process term*. When it is clear from the context, we refer to

the process executing P as just the process P. Among the channels by which a process $\texttt{proc}[a](P)$ is connected to other processes, we designate one channel, a, as its *offering channel*. Process P is said to *provide for*, or *to be the provider of*, its offering channel a. For all the other channels, P assumes the role of a *client*. The distinction of provider and client roles of a process is the hallmark of an intuitionistic system. It has a profound impact on the design (§4.1) and semantics (§3 and §4.2) of the system. The separation supports a wide range of metatheoretic properties, including and not limited to deadlock freedom (Thm. 4.7).

We begin with the process terms, denoted by P, Q. Process terms can have occurrences of free and bound variables (denoted by x, y), for which we adopt the usual conventions to denote binding. Variables range over names of runtime channels. Process terms can be classified into three groups, depending on the channel they immediately operate on. *Provider processes*, such as $\texttt{close}^{t.p}$ and $\texttt{offer}^{t.p}\ \{P_1 \mid P_2\}$, act on the offering channel. Because the offering channel of a process is distinguished, provider processes do not name the channel explicitly. *Client processes*, such as $\texttt{wait}^T x\ ;\ Q$ and $\texttt{case}^T\ x\ \{Q_1 \mid Q_2\}$, act on the channel variable x. In either case, process terms are typically *sequences*, consisting of the next communication action and a continuation process. For example, provider process $\texttt{switchL}^{t.p}\ ;\ P$ ($\texttt{switchR}^{t.p}\ ;\ P$) sends message `left` (`right`) and then continues with P. Dually, the client process $\texttt{case}^T\ x\ \{Q_1 \mid Q_2\}$ receives the message over x and continues with Q_1 (Q_2). An overview of all process terms with description is given in Table 1. Processes $\texttt{fwd}^T x$ and $\texttt{spawn}^T P\ ;\ x\ .\ Q$ have judgmental roles and constitute the third group. The process $\texttt{fwd}^T x$ forwards any actions from and to channel x to it's offering channel, therefore identifying them. Process $\texttt{spawn}^T P\ ;\ x\ .\ Q$ spawns P as a separate process and binds its offering channel to x for use in the continuation Q.

In addition to specifying what actions a process should take, processes also carry timing related information as superscripts. In this regard, providers and clients again take on distinct roles. A provider process has a predicate $t.p$ as a superscript, indicating at what time it is willing to take the action. Here t is a variable ranging over points in time (denoted T) and p is a proposition involving t. The exact definition is inessential and will be clarified shortly. A predicate $t.p$ constrains a message exchange to only occur at times T such that the proposition p holds true for t substituted with T (*i.e.,* $\{T/t\}p$ is true). In other words, provider processes limit the message exchange to a time window, while promising to engage in the exchange at any valid choice of time. Client processes, on the other hand, carry a concrete choice of time T as superscripts. The chosen T is expected to satisfy the predicate set up by the provider of the referenced channel, in addition to some other requirements guaranteeing that time moves forward. For example, the process $\texttt{offer}^{t.p}\ \{P_1 \mid P_2\}$ offers to receive either `left` or `right` at any time satisfying p. Its client $\texttt{selL}^T x\ ;\ Q$ chooses to send `left` at exactly time T.

2.2.2 TILLST: Protocol Specification Language. The sequences of actions that a process takes over its offering channel constitutes its *protocol*. *Session types* (denoted by A, B) are behavioral types that prescribes such protocols. Generally, the superscript of a session type indicates the time at which a communication action may occur, while the connective indicates the nature of the action itself. The connectives include *sequencing* and *branching* constructs. Sequencing is expressed by operators $\otimes$ and $\multimap$. The type $A_1 \multimap A_2$ indicates that, after the receipt of a channel of type A_1, the protocol transitions to behaving as A_2. Conversely, the type $A_1 \otimes A_2$ denotes the sending of a channel of type A_1. Branching is expressed by the types $A_1\ \&\ A_2$ and $A_1 \oplus A_2$, offering a choice between the sessions A_1 and A_2 and making a choice between the sessions A_1 and A_2, *resp.* The choice is conveyed by receiving and sending labels `left` or `right`, reminiscent of products and sums in functional languages. The type $\mathbf{1}$ denotes the end state of a protocol. The connectives come from intuitionistic linear logic, given ILLST's foundation. Table 1 lists all connectives, with

Table 1. TILLST types and process terms.

Role	Type (A)	Process Term (P)	Time	Action	Cont.
Provider	$\mathbf{1}^{t.p}$	$\texttt{close}^{t.p}$		send closing signal `cls`	none
	$A_1 \multimap^{t.p} A_2$	$\lambda^{t.p}(x : A_1)P$		accept channel, bind to x	P
Acting on	$A_1 \otimes^{t.p} A_2$	$P_1 \otimes^{t.p} P_2$	any T	spawn P_1 and send it	P_2
providing	$A_1 \oplus^{t.p} A_2$	$\texttt{switchL}^{t.p}\ ; P$	s.t. $\{T/t\}p$	send `left`	P
channel	$A_1 \oplus^{t.p} A_2$	$\texttt{switchR}^{t.p}\ ; P$		send `right`	P
P @ $T :: A$	$A_1 \,\&^{t.p}\, A_2$	$\texttt{offer}^{t.p}\ \{P_1 \mid P_2\}$		accept `left` or `right`	P_1 or P_2
	$\mathbf{1}^{t.p}$	$\texttt{wait}^{T}\, x\ ; Q$		wait for closing signal `cls`	Q.
Client	$A_1 \multimap^{t.p} A_2$	$\texttt{app}^{T}\, x\ (P)\ ; Q$		spawn P and send it	Q
	$A_1 \otimes^{t.p} A_2$	$\texttt{split}^{T}\ x\ ; y\ .\ Q$	fixed T	receive channel, bind to y	Q
Acting on	$A_1 \oplus^{t.p} A_2$	$\texttt{case}^{T}\ x\ \{Q_1 \mid Q_2\}$	s.t. $\{T/t\}p$	receive `left` or `right`	Q_1 or Q_2
chan. $x : A$	$A_1 \,\&^{t.p}\, A_2$	$\texttt{selL}^{T}\, x\ ; Q$		send `left`	Q
	$A_1 \,\&^{t.p}\, A_2$	$\texttt{selR}^{T}\, x\ ; Q$		send `right`	Q
Judgmental rules		$\texttt{fwd}^{T}\, x$	fixed T	forward x to offerring channel	
		$\texttt{spawn}^{T}\, P\ ; x\ .\ Q$		spawn P and connect to it on x	

corresponding process terms. Due to the provider-client distinction, each connective has a term for the provider and client.

The Logic and Model of Time. The language presented uses a straightforward model for time. Every point in time is essentially an integer offset i away from one distinguished *initial* point in time `init`. This model of time is discrete, linear, and unbounded. Points in time can be compared for equality and satisfy less-or-equal-to relations. Propositions p include the whole quantifier-free fragment of first order logic. This choice of logic is expressive enough for our examples and many real-world applications.

2.2.3 Exchange of Functional Values. Our language may be easily extended to support evaluating, sending, and receiving functional expressions. Let e be the syntactic sort of expressions in a (typed) functional language of your choice. For the sake of simplicity, we assume e is at least terminating and pure. Lifting those restrictions presents difficulties largely orthogonal to the development of this paper. Let τ be the sort of types in said language. For the sake of our sensor example, let us assume that the language includes at least Booleans (`bool`) and integer numbers (`int`).

$$
\begin{aligned}
\text{Functional Types} \quad & \tau ::= \texttt{bool} \mid \texttt{int} \mid \dots \\
\text{Functional Expressions} \quad & e ::= x \mid \texttt{true} \mid \texttt{false} \mid n \mid \dots \\
\text{Session Types} \quad & A ::= \cdots \mid\ !^{t.p}_{\tau}.A \mid\ ?^{t.p}_{\tau}.A \\
\text{Process} \quad & P ::= \cdots \mid \texttt{produce}^{t.p}\, e\ ; P \mid \texttt{consume}^{T}\, y\ ; x\ .\ Q \\
& \qquad \mid \texttt{query}^{t.p}\ ; x\ .\ P \mid \texttt{supply}^{T}\, x(v)\ ; Q
\end{aligned}
$$

We extend the session types and process terms accordingly. Sessions types $!^{t.p}_{\tau}.A$ and $?^{t.p}_{\tau}.A$ allow sending a functional value of type τ and receiving a value of type τ, *resp.* Provider process $\texttt{produce}^{t.p}\, e\ ; P$ evaluates e to a value and then sends it over the offering channel. The message is intended for the client process $\texttt{consume}^{T}\, y\ ; x\ .\ Q$, which binds the message to a variable x. Be aware that we are overloading variable names x, y for both channel and functional variables. The

pair $\texttt{query}^{t.p}\ ; x\ .\ P$ and $\texttt{supply}^{T}\ x(v)\ ; Q$ perform analogous actions, with provider/client roles switched.

2.3 Code for the Controller

We now have the tools to prescribe the protocols for the sensor ($A_{\texttt{BME680}}$) and the controller ($A_{\texttt{Hub}}$):

$$
\begin{aligned}
A_{\texttt{BME680}} &\triangleq A_{\texttt{T}}\ \&^{t_1.t_0 \le t_1}\ A_{\texttt{TG}} \\
A_{\texttt{T}} &\triangleq\ !^{t_2.t_1 \le t_2}_{\tau_{\texttt{Temp}}}.\ \mathbf{1}^{t_3.t_2 \le t_3} \\
A_{\texttt{TG}} &\triangleq\ !^{t_2.t_1 \le t_2}_{\tau_{\texttt{Temp}}}.\ !^{t_3.t_2+30\ ms \le t_3}_{\tau_{\texttt{Gas}}}.\ \mathbf{1}^{t_4.t_3+20\ ms \le t_4} \\
A_{\texttt{Hub}} &\triangleq A_{\texttt{BME680}}\ \multimap^{t_1.t_0 \le t_1}\ A_{\texttt{BME680}}\ \multimap^{t_2.t_2 = t_1}\ !^{t_3.t_1+50\ ms \le t_3}_{\texttt{bool}}.\ \mathbf{1}^{t_4.t_4 = t_3}
\end{aligned}
$$

Suppose we start the protocol at time t_0. Notice that t_0 is free everywhere. The protocol of the sensor starts by accepting a configuration message of either `left` or `right`, choosing between measuring just the temperature ($A_{\texttt{T}}$) or both temperature and air quality [1] ($A_{\texttt{TG}}$). The message simultaneously puts the sensor into (forced) operational mode [2]. The message can happen at anytime t_1 after initial time t_0 ($t_1.t_0 \le t_1$). If $A_{\texttt{T}}$ is selected, the temperature result (a functional value of type $\tau_{\texttt{temp}}$) is immediately available for collection. This is conveyed by the predicate $t_2.t_1 \le t_2$, licensing the client to receive the functional value at any time t_2 in the future of t_1. This illustrates an important feature of the language, we are now in a position to appreciate:

> TILLST supports *binding* the actual time of communication for future reference.

A more precise reading of the type $A_{\texttt{BME680}}$ is that a process may accept a `left` or `right` at any time that follows t_0, and *let* t_1 *be the actual time at which the exchange occurs*. This now-bound point in time t_1 may be used both for specifying protocols in types, which we are seeing now, and for choosing the time of future actions, which we will see when we examine the process term for the controller. Wrapping up this branch, because only temperature is collected, the sensor may be shut-down any time after the temperature is received, a point in time now-bound at t_2, conveyed by the predicate $t_2 \le t_3$. On the other hand, if $A_{\texttt{TG}}$ is selected, then the sensor temperature result is immediately available, as before. After the temperature result is collected at t_2, the subsequent air quality result ($\tau_{\texttt{gas}}$) is only available after 30 *ms*, allowing enough time for the sensor to warm up. This is expressed by $t_2 + 30\ ms \le t_3$. Finally, after the collection of both results at time t_3, the sensor takes 20 *ms* to cool down before it can be closed at t_4.

The controller type $A_{\texttt{Hub}}$ uses *higher-order channels*, which are channels whose messages are names of other channels. At some time t_1 (or t_2, since $t_1 = t_2$) after t_0, the controller simultaneously gets hold of two channels, each connected to a sensor. Upon receiving the pair of sensors, the controller has 50 *ms* to compute and make available a `bool` response, ready for collection at t_3. Then, the controller is immediately terminated at t_3 ($t_4 = t_3$).

The ability to prescribe the relative timing between actions is critical for this specification. Many protocols, including this one, require a client to "`wait`" or "`sleep`" for a fixed or variable amount of time between consecutive actions. Existing modeling tools and languages based on *timed automata* (*e.g.,* [Bocchi et al. 2019]) address this via imperative *clock resets* (as shown in §5). However, these are not the only kinds of timing requirements. Other common requirements include the action takes place "no-later" or "no earlier" than some other action, or that the action "leaves enough room" for some other action. These timing requirements can be logically complicated to express

[1] The BME680 datasheet [Bosch 2024] refers to the feature that measures air quality as "gas" measurement, hence the letter "G".

[2] We combine the configuration of measurement and mode selection, which have no timing requirements between them.

indirectly via clock resets. TILLST accommodates these and introduces a rich declarative language for timing specification, allowing for intuitive and natural timing specifications.

We turn now to the process term for the controller, using Table 1 as a guide:

$$
\begin{aligned}
P \mathbin{@} t_0 :: A_{\mathsf{Hub}} \triangleq\ & \lambda^{t_1.\, t_0 \le t_1}(x : A_{\mathtt{BME680}})\, \lambda^{t_2.\, t_2 = t_1}(y : A_{\mathtt{BME680}}) P_{\mathsf{Hub}} \\
& \mathtt{selR}^{t_1} x\ ;\ u_1 \leftarrow \mathtt{consume}^{t_1}\ x; \\
& \mathtt{selL}^{t_1} y\ ;\ u_2 \leftarrow \mathtt{consume}^{t_1}\ y\ ;\ \mathtt{wait}^{t_1} y\ ; \\
& v_1 \leftarrow \mathtt{consume}^{t_1 + 30\ ms}\ x\ ;\ \mathtt{wait}^{t_1 + 50\ ms} x\ ; \\
& \mathtt{produce}^{t_3.\, t_1 + 50 \le t_3}\ \mathtt{NeedAC}(u_1, u_2, v_1)\ ;\ \mathtt{close}^{t_4.\, t_4 = t_3}
\end{aligned}
$$

As dictated by its protocol, the controller initializes at t_0, then simultaneously accepts two channels x and y, each connected to a sensor. Recall that y connects to the living room sensor, from which we only require temperature, and x connects to the bedroom sensor, from which we require both the temperature and air quality data. After receiving the channels, the controller *interleaves* actions to the sensors. It first sends `right` to x, selecting both measurements and receives the temperature results, all done at t_1 at which the channels where received. Instead of waiting on the subsequent air quality results, it turns its attention to y. It configures, collects from, and then closes y. It then reads the air quality result from x at $t_1 + 30\ ms$ and closes it at $t_2 + 50\ ms$, namely 20 ms after the previous step. Finally, it computes the `bool` through some defined algorithm `NeedAC` in the functional language and then closes itself.

The ability of a process to interleave actions on different channels as a client, without revealing this interleaving as part of its offering protocol, preserves abstraction and is one of the key benefits of intuitionism. To meet timing requirements, applications need to be able to put multiple processes in motion. If we were to modify the example to collect temperature and air quality data from both x and y, then both sensors must be configured to warm up in parallel to meet the output deadline. Intuitionism again helps here, as the session type only prescribes what happens over the offering channel, processes have the freedom to re-order client actions as they see fit.

3 Semantics through Timed Semantic Session Logical Relation (TSSLR)

The heterogeneity of our target domain requires the semantics of programs, *i.e.,* how programs run (*a.k.a.,* communicate), to be the front and centerpiece of the development. In this section, we begin this exploration by looking at the *dynamic semantics*, which concerns the stepping of programs through time using a timed labelled transition system. We will then distill the meaning of *session types* based on their role as classifiers for program behavior through the lens of the *logical relations* method. In §5, we build on this foundation to show that the implementation of the BME680 environment sensor discussed in §2 inhabits our logical relation. Thanks to semantic typing and the use of a timed labelled transition system, these "foreign objects" can be easily accommodated by simply extending the list of transition rules while maintaining an almost identical proof structure.

3.1 Dynamic Semantics

The dynamics describes a transition system that models computation as evolving with time in a way that is consistent with our instant-based model of time. In this section, we will set up a *timed labeled transition system*, where transitions are labeled by both the action (if any) taken and the time at which it takes place.

At runtime, a program amounts to a configuration Ω of processes, defined as:

$$\text{Conf} \quad \Omega ::= \mathbf{1} \mid \mathtt{proc}[a](P) \mid \mathtt{fwd}[a](b) \mid \Omega_1 \otimes \Omega_2$$

The nullary configuration is written as "$\mathbf{1}$". The runtime incarnation of a process term P is $\texttt{proc}[a](P)$, where a is the providing channel of the process. The forwarding configuration $\texttt{fwd}[a](b)$ identifies runtime channels a and b by forwarding all messages between them. Finally, $\Omega_1 \otimes \Omega_2$ denotes the concurrent composition of two configurations.

As usual, structural congruence rules identify configurations up to reordering. Nullary configurations may be dropped silently, and forwarding configurations may be dropped after renaming the channels referenced accordingly. Structural congruence is set up to be a congruent equivalence relation. Below we are showing the critical rules. For the complete set of rules see the full paper [Yao et al. 2024b].

$$\frac{}{\mathbf{1} \otimes \Omega \equiv \Omega}\ [\text{C-Stop}] \qquad \frac{}{\texttt{proc}[a](P) \otimes \texttt{fwd}[b](a) \equiv \texttt{proc}[b](P)}\ [\text{C-Fwd}]$$

$$\frac{}{\Omega_1 \otimes \Omega_2 \equiv \Omega_2 \otimes \Omega_1}\ [\text{C-Comm}] \qquad \frac{}{\texttt{fwd}[c](b) \otimes \texttt{fwd}[b](a) \equiv \texttt{fwd}[c](a)}\ [\text{C-Ctr}]$$

Next, we define the *timed labelled transition system*. The judgment $\Omega \overset{\alpha}{\underset{T}{\longmapsto}} \Omega'$ asserts that at time T the configuration Ω is ready to step to Ω' taking action α. Actions α are defined as follows:

$$\text{Action} \quad \alpha ::= \varepsilon \mid a\,!\,b \mid a\,?\,b \mid a\,!\,\texttt{left} \mid a\,!\,\texttt{right} \mid a\,?\,\texttt{left} \mid a\,?\,\texttt{right} \mid a\,!\,\texttt{cls} \mid a\,?\,\texttt{cls}$$

The action ε is the *nullary, silent* [3] action, corresponding to an actual reduction step of the configuration. Silent actions are usually omitted. Non-silent actions always carry a message, a channel over which the message propagates, and a direction. The symbol "!" means send and the symbol "?" means receive. Messages may be labels, channel names, or the closing signal. For example, $\Omega \overset{a!\texttt{left}}{\underset{T}{\longmapsto}} \Omega'$ means that configuration Ω at time T may send the label $\texttt{left}$ over channel a and become Ω'.

Our transition system allows any pair of compatible processes to communicate anywhere within the configuration. This is achieved by structural congruence in conjunction with the *framing* and communication rules.

$$\frac{\Omega_1 \overset{\alpha}{\underset{T}{\longmapsto}} \Omega_1'}{\Omega_1 \otimes \Omega_2 \overset{\alpha}{\underset{T}{\longmapsto}} \Omega_1' \otimes \Omega_2}\ [\text{D-Frame}] \qquad \frac{\Omega_1 \overset{\alpha}{\underset{T}{\longmapsto}} \Omega_1' \qquad \Omega_2 \overset{\overline{\alpha}}{\underset{T}{\longmapsto}} \Omega_2'}{\Omega_1 \otimes \Omega_2 \underset{T}{\longmapsto} \Omega_1' \otimes \Omega_2'}\ [\text{D-Comm}]$$

The premise of rule [D-Frame] "frames off" the surrounding configuration Ω_2, allowing us to isolate Ω_1 for a local transition.

Two processes are compatible to communicate if they are willing to take *complementary* (*a.k.a.,* dual) actions. Complementary actions are defined as follows, with $\overline{\overline{\alpha}} = \alpha$:

$$\overline{a\,?\,\texttt{left}} \triangleq a!\texttt{left} \qquad \overline{a\,?\,\texttt{right}} \triangleq a!\texttt{right} \qquad \overline{a\,!\,\texttt{left}} \triangleq a?\texttt{left} \qquad \overline{a\,!\,\texttt{right}} \triangleq a?\texttt{right}$$

$$\overline{\varepsilon} \triangleq \varepsilon \qquad \overline{a\,?\,b} \triangleq a\,!\,b \qquad \overline{a\,!\,b} \triangleq a\,?\,b \qquad \overline{a\,?\,\texttt{cls}} \triangleq a\,!\,\texttt{cls} \qquad \overline{a\,!\,\texttt{cls}} \triangleq a\,?\,\texttt{cls}$$

In [D-Comm], if a pair of processes is willing to take complementary actions, then they communicate and both transition. The overall step involving both processes is silent because the communication happens "internally".

[3] Commonly known as τ transitions in the π-calculus literature. We choose ε to avoid conflicting with functional types.

$$\text{[D-fwd]} \quad \mathtt{proc}[a](\,\mathtt{fwd}^T b\,) \underset{T}{\longmapsto} \mathtt{fwd}[a](\,b\,)$$
$$\text{[D-spawn]} \quad \mathtt{proc}[a](\,\mathtt{spawn}^T P\,;x\,.\,Q\,) \underset{T}{\longmapsto} \mathtt{proc}[b](\,P\,) \otimes \mathtt{proc}[a](\,\{b/x\}Q\,)$$

$$\text{[D-}\mathbf{1}\text{ L]} \quad \mathtt{proc}[a](\,\mathtt{wait}^T b\,;Q\,) \xmapsto[T]{b?\mathtt{cls}} \mathtt{proc}[a](\,Q\,)$$
$$\text{[D-\& L1]} \quad \mathtt{proc}[a](\,\mathtt{selL}^T b\,;Q\,) \xmapsto[T]{b!\mathtt{left}} \mathtt{proc}[a](\,Q\,)$$
$$\text{[D-\& L2]} \quad \mathtt{proc}[a](\,\mathtt{selR}^T b\,;Q\,) \xmapsto[T]{b!\mathtt{right}} \mathtt{proc}[a](\,Q\,)$$
$$\text{[D-}\oplus\text{ L1]} \quad \mathtt{proc}[a](\,\mathtt{case}^T\, b\,\{Q_1 \mid Q_2\}\,) \xmapsto[T]{b?\mathtt{left}} \mathtt{proc}[a](\,Q_1\,)$$
$$\text{[D-}\oplus\text{ L2]} \quad \mathtt{proc}[a](\,\mathtt{case}^T\, b\,\{Q_1 \mid Q_2\}\,) \xmapsto[T]{b?\mathtt{right}} \mathtt{proc}[a](\,Q_2\,)$$
$$\text{[D-}\otimes\text{ L]} \quad \mathtt{proc}[a](\,\mathtt{split}^T\, b\,;y\,.\,Q\,) \xmapsto[T]{b?c} \mathtt{proc}[a](\,\{c/y\}Q\,)$$
$$\text{[D-}\multimap\text{ L]} \quad \mathtt{proc}[a](\,\mathtt{app}^T b\,(P)\,;Q\,) \xmapsto[T]{b!c} \mathtt{proc}[a](\,Q\,) \otimes \mathtt{proc}[c](\,P\,)$$

With premise $\{T/t\}p$:

$$\text{[D-}\mathbf{1}\text{]} \quad \mathtt{proc}[a](\,\mathtt{close}^{t.p}\,) \xmapsto[T]{a!\mathtt{cls}} \mathbf{1}$$
$$\text{[D-\& 1]} \quad \mathtt{proc}[a](\,\mathtt{offer}^{t.p}\,\{P_1 \mid P_2\}\,) \xmapsto[T]{a?\mathtt{left}} \mathtt{proc}[a](\,\{T/t\}P_1\,)$$
$$\text{[D-\& 2]} \quad \mathtt{proc}[a](\,\mathtt{offer}^{t.p}\,\{P_1 \mid P_2\}\,) \xmapsto[T]{a?\mathtt{right}} \mathtt{proc}[a](\,\{T/t\}P_2\,)$$
$$\text{[D-}\oplus\text{ R1]} \quad \mathtt{proc}[a](\,\mathtt{switchL}^{t.p}\,;P\,) \xmapsto[T]{a!\mathtt{left}} \mathtt{proc}[a](\,\{T/t\}P\,)$$
$$\text{[D-}\oplus\text{ R2]} \quad \mathtt{proc}[a](\,\mathtt{switchR}^{t.p}\,;P\,) \xmapsto[T]{a!\mathtt{right}} \mathtt{proc}[a](\,\{T/t\}P\,)$$
$$\text{[D-}\otimes\text{]} \quad \mathtt{proc}[a](\,P_1 \otimes^{t.p} P_2\,) \xmapsto[T]{a!c} \mathtt{proc}[a](\,\{T/t\}P_2\,) \otimes \mathtt{proc}[c](\,\{T/t\}P_1\,)$$
$$\text{[D-}\multimap\text{]} \quad \mathtt{proc}[a](\,\lambda^{t.p}(x:A)P\,) \xmapsto[T]{a?c} \mathtt{proc}[a](\,\{T,c/t,x\}P\,)$$

Fig. 1. Timed labelled transitions $\Omega \xmapsto[T]{\alpha} \Omega'$.

Fig. 1 lists the remaining timed labelled transition rules of our system. Clients can only step at a time T, dictated by the client, or in the case of [D-fwd] and [D-spawn], specified by the process term. All provider rules require $\{T/t\}p$, meaning that the time of interaction must satisfy the demand imposed. The rules of Fig. 1 give our instant-based model of time computational meaning: computation happens *at an instant*. Messages between processes are sent and received at exactly the same instant. In particular, there is no notion of an in-flight message.

To describe the computation of a configuration over a period of time, silent labelled transitions can be "chained" together, each happening at progressing times. For example, the reductions $\Omega \underset{T_1}{\longmapsto} \Omega_1 \cdots \underset{T_n}{\longmapsto} \Omega_n \underset{T_{n+1}}{\longmapsto} \Omega'$ take an initial configuration Ω, starting at T, to a configuration Ω' at T', such that $T \le T_i \le T_{i+1} \le T'$ for all i. This chaining is expressed by the judgment

$$⏱_T, \Omega \longmapsto^* ⏱_{T'}, \Omega'$$

asserting that an initial configuration Ω at T computes and reaches Ω' at T'. Configurations Ω and Ω' are called the *initial and terminal configurations, resp.* The judgment is a generalization of the usual

$$\frac{}{\mathtt{refl}^{\Omega}_{T} : \clock_T, \Omega \longmapsto^* \clock_T, \Omega}\ [\text{G-refl}] \qquad \frac{T_1 \leq T_2 \qquad \sigma : \clock_{T_2}, \Omega \longmapsto^* \clock_{T_3}, \Omega'}{\mathtt{stepT}^{\Omega}_{T_1;T_2}(\sigma) : \clock_{T_1}, \Omega \longmapsto^* \clock_{T_3}, \Omega'}\ [\text{G-stepT}]$$

$$\frac{\Omega \underset{T_1}{\longmapsto} \Omega' \qquad \sigma : \clock_{T_1}, \Omega' \longmapsto^* \clock_{T_2}, \Omega''}{\mathtt{stepC}^{\Omega;\Omega'}_{T_1}(\sigma) : \clock_{T_1}, \Omega \longmapsto^* \clock_{T_2}, \Omega''}\ [\text{G-stepC}]$$

Fig. 2. Timed multistep reduction judgment

multistep reduction relation, which not only requires performing an *instantaneous* computation step but also *advancing the clock*. To convey which of these steps are performed, we annotate the judgment with *proof terms* σ. The resulting multistep reduction judgment $\sigma : \clock_T, \Omega \longmapsto^* \clock_{T'}, \Omega'$ is defined in Fig. 2. These rules define both the judgment and syntax of its proof terms σ. Rule [G-stepC] steps the configuration while maintaining time and rule [G-stepT] explicitly advances the time to some time in the future. Proof terms σ syntactically represent sequences of silent transition steps with monotonically increasing time, and are referred to simply as *sequences*.

The aforementioned dynamics is inherently non-deterministic. At each step, the judgment allows the choice between progressing time or progressing the configuration. Not all such choices are fruitful. For example, choosing to advance time from T to T' for a configuration Ω at T, will rule out any pending silent step $\Omega \underset{T}{\longmapsto} \Omega'$ at T, possibly resulting in a stuck configuration.

3.2 Timed Semantic Session Logical Relation

As emphasized earlier, session types prescribe and classify process behavior on their offering channel. Given a session type A, we are interested in capturing precisely the runtime behavior that a configuration of processes must exhibit in order to comply with the protocol prescribed by A. For our logical relation, we thus define a family of sets $\mathcal{L}[\![A]\!]\ @\ T$, indexed by session type A, that characterizes configurations that are *providers* of protocol A at time T. To account for the provider-client distinction inherent to intuitionism, we must complement this characterization with a family of sets $\mathcal{L}^{\star}[\![A]\!]\ @\ T$, indexed by session type A, that characterizes configurations that a *client may use* as prescribed by type A at time T. These two characterizations account for the dual roles providers and clients assume with regard to timing considerations, (see §2.2.1): whereas providers specify temporal predicates and assert availability at any valid choice of time, clients satisfy temporal predicates and may choose a suitable valid time.

We define the sets $\mathcal{L}[\![A]\!]\ @\ T$ and $\mathcal{L}^{\star}[\![A]\!]\ @\ T$ in §3.2.2. These definitions rely on our timed labelled transition system and multistep reductions introduced in §3.1. For the latter we find it convenient to adopt a more global perspective and work instead with the notion of *computable trajectories*, which we define precisely in §3.2.1. Trajectories (denoted by r, s) are functions from *points in time* to configurations, characterizing the program execution state at each point in time. We are especially interested in the subset of trajectories that can be realized by some sequence σ. Intuitively, σ is a discrete but computable counterpart for the more convenient notion of trajectories. Towards this end we define a relation $\mathcal{R}(r\ ;\ \sigma)$ asserting r is computed by the sequence σ. An element of the relation is termed a *computable trajectory*, or just *trajectory* for short. Section §3.2.1 defines three operations on computable trajectories, a *computable trajectory algebra*, each shown to respect the computability relation and shown to commute and cancel properly.

- An *interleaving* operator $r_1 \otimes r_2$ combines two computable trajectories into one describing the execution of their concurrent composition.

- The *partitioning* operators $r_1 \downarrow^T$ and $r_2 \uparrow^T$ access the before-T and after-T component of the trajectory, *resp.*
- Dually, the *concatenation* operator $r_1 \mathbin{@} r_2$ pieces two trajectories of connected domains together.

3.2.1 Computable Trajectories. In the following sections, unless explicitly stated or deducible from the context, we assume that processes, configurations, and syntactic elements of the temporal logic are all closed.

As discussed, the execution of a program through time can be understood in terms of functions from points in time to configurations, namely *trajectories*.

Definition 3.1 (Temporal intervals). A temporal interval I is either *bounded* or *unbounded*:

- (Bounded) $[\, T_1, T_2\,) \triangleq \{T \mid T_1 \le T < T_2\}$
- (Unbounded) $[\, T, \infty\,) \triangleq \{T' \mid T \le T\}$

Definition 3.2 (Trajectories, lines, segments). A *trajectory* is a function $I \to \mathsf{Conf}$. If I is unbounded, then it is said to be a *line*, otherwise it is said to be a *segment*. The set of trajectories over interval I is denoted by $\mathtt{C}_I$.

Not all trajectories are computational by nature. For example, take the reals as a model, the function that sends all rational numbers to $\mathtt{proc}[a](\,\mathtt{switchL}^{t,p}\,;P\,)$ and irrational numbers to $\mathtt{proc}[a](\,\mathtt{switchR}^{t,p}\,;P\,)$ certainly does not result from our dynamics. We are interested in functions that result from our dynamics. This motivates the following set of definitions.

Definition 3.3 ($\mathtt{c}^{\Omega}_{I}$). $\mathtt{c}^{\Omega}_{I}(-)$ is the constant function sending I to Ω;

Definition 3.4 ($\mathtt{Ext}^{\Omega}_{T}(r)$). Function $\mathtt{Ext}^{\Omega}_{T}(-)$ extends the domain leftward to T of its argument trajectory by sending the additional inputs to Ω. Full definitions are available in the full paper [Yao et al. 2024b].

Definition 3.5 (Computable trajectory). Let $\mathcal{R}(r\,;\sigma)$ be the strongest relation satisfying:

- $\mathcal{R}(\mathtt{c}^{\Omega}_{[\,T,T'\,)}\,;\mathtt{refl}^{\Omega}_{T})$, where T' may be ∞.
- $\mathcal{R}(r\,;\mathtt{stepC}^{\Omega;\Omega'}_{T}(\,\sigma\,))$ if $\mathcal{R}(r\,;\sigma)$.
- $\mathcal{R}(\mathtt{Ext}^{\Omega}_{T}(r)\,;\mathtt{stepT}^{\Omega}_{T;T'}(\,\sigma\,))$ if $\mathcal{R}(r\,;\sigma)$.

An element $\langle r, \sigma\rangle$ of the relation is called a *computable trajectory*, or just *trajectory*. Computable trajectories are denoted by w. The trajectory $\langle r, \sigma\rangle$ is more precisely called a *computable line*, if r is a line, and a *computable segment*, if r is a segment.

Intuitively, $\mathcal{R}(r\,;\sigma)$ takes a discrete description of computation over time, given by σ, and fills in the gaps by (1) assuming that the configuration stays unchanged until the next silent transition step and (2) maintaining the last configuration indefinitely into the future, if the trajectory is a line.

Definition 3.6. Let $w = \langle r, \sigma\rangle$ be a computable trajectory. The domain of w, write $\mathsf{Dom}(w)$, is the domain of r. For all $T \in \mathsf{Dom}(w)$, define $w(T) \triangleq r(T)$.

Definition 3.7 ($S_{[\,T_1,T_2\,)}(\Omega\,;\Omega')$ and $S_{[\,T,\infty\,)}(\Omega)$). Let $S_{[\,T_1,T_2\,)}(\Omega\,;\Omega')$ denote the set of computable trajectories $\langle r, \sigma\rangle$ such that (1) $r \in \mathtt{C}_{[\,T_1,T_2\,)}$, (2) initial configuration of σ is Ω, and (3) if r is a segment then the terminal configuration of σ is Ω', otherwise formally write $\Omega' = \cdot$. Further define $S_{[\,T,\infty\,)}(\Omega) \triangleq S_{[\,T,\infty\,)}(\Omega\,;\cdot)$.

From a computable segment we can recover a sequence (proof in the full paper [Yao et al. 2024b]):

Lemma 3.8. *If* $w \in S_{[\,T_1,T_2\,)}(\Omega_1\,;\Omega_2)$, *then* $\sigma : \textcircled{\scriptsize T}_{T_1}, \Omega_1 \longmapsto^{*} \textcircled{\scriptsize T}_{T_2}, \Omega_2$.

Computable trajectories are functions with computability receipts. Therefore, it is natural to consider and define common operations on functions for computable trajectories. The subtlety of the operations lies in the handling of the receipts, that is the operations on trajectories must be justified by operations on sequences. These operations on the sequences are constructive, therefore proofs carried out with computable trajectories yield effective ways to determine schedules.

In this section, we describe the operations and the necessary properties they satisfy, for the exact construction please refer to the full paper [Yao et al. 2024b].

Definition 3.9 (Equivalence). Trajectories $w_1 \in S_{I_1}(\Omega_1 ; \Omega_1')$ and $w_2 \in S_{I_2}(\Omega_2 ; \Omega_2')$ are said to be *equivalent* on interval I, $w_1 \sim_I w_2$, iff for all $T \in I$, $w_1(T) = w_2(T)$.

In particular, if $I = I_1 = I_2$ and $w_1 \sim_I w_2$, write just $w_1 \sim w_2$.

The equivalence relations on computable trajectories treat them as functions. We remark that $w_1 \sim_I w_2$ is reflexive, symmetric, and transitive by definition.

We will now define a pair of operators that are dual to each other.

Definition 3.10 (Concatenation). The *concatenation* operator $w_1 \mathbin{@} w_2$ concatenates trajectories with connected domains:

$$\cdot \mathbin{@} \cdot : S_{[\,T_1,T_2\,)}(\Omega_1 ; \Omega_2) \to S_{[\,T_2,T_3\,)}(\Omega_2 ; \Omega_3) \to S_{[\,T_1,T_3\,)}(\Omega_1 ; \Omega_3)$$

Concatenation of trajectories coincides with the constituents on the domain each is defined:

Corollary 3.11. *$\forall w_1, w_2$, $w_1 \sim_{\mathsf{Dom}(w_1)} (w_1 \mathbin{@} w_2)$ and $w_2 \sim_{\mathsf{Dom}(w_2)} (w_1 \mathbin{@} w_2)$.*

Proof in the full paper [Yao et al. 2024b]. Additionally, given a trajectory and a point in its domain, it is possible to partition the domain by this point, motivating the following pairs of operators.

Definition 3.12 (Partition). The left and right partition of w by time T s.t. $T \in [\,T_1, T_2\,)$ are:

$$\cdot \downarrow^T : S_{[\,T_1,T_2\,)}(\Omega_1 ; \Omega_2) \to S_{[\,T_1,T\,)}(\Omega ; w(T))$$
$$\cdot \uparrow^T : S_{[\,T_1,T_2\,)}(\Omega_1 ; \Omega_2) \to S_{[\,T,T_2\,)}(w(T) ; \Omega_2)$$

A trajectory coincides with its parts on each domain (proof in the full paper [Yao et al. 2024b]):

Corollary 3.13. *For all w and $T \in \mathsf{Dom}(w)$, $w \downarrow^T \sim_{\mathsf{Dom}(w\downarrow^T)} w$ and $w \uparrow^T \sim_{\mathsf{Dom}(w\uparrow^T)} w$.*

The operators are therefore dual in the following sense:

Lemma 3.14. *For all w and $T \in \mathsf{Dom}(w)$, $w \sim (w \downarrow^T) \mathbin{@} (w \uparrow^T)$*

Proof. Take arbitrary $T' \in \mathsf{Dom}(w)$. If $T' \geq T$, then $((w \downarrow^T) \mathbin{@} (w \uparrow^T))(T') = (w \uparrow^T)(T') = w(T')$. Otherwise, $((w \downarrow^T) \mathbin{@} (w \uparrow^T))(T') = (w \downarrow^T)(T') = w(T')$. □

Partitioning and concatenation both operate on the trajectory of a monolithic configuration. The computation of a *compound* configuration through time consists of multiple configurations computing through time concurrently. This motivates us to define the interleaving operator:

Definition 3.15 (Interleaving). The *interleaving* operator has the following signature:

$$\cdot \otimes \cdot : S_{[\,T,T'\,)}(\Omega_1 ; \Omega_1') \to S_{[\,T,T'\,)}(\Omega_2 ; \Omega_2') \to S_{[\,T,T'\,)}(\Omega_1 \otimes \Omega_1' ; \Omega_2 \otimes \Omega_2')$$

The operator is constructed such that (proof in the full paper [Yao et al. 2024b]):

Corollary 3.16. *For all w_1 and w_2 and T, $w_1 \otimes w_2(T) = (w_1(T)) \otimes (w_2(T))$.*

As a further corollary, partitioning and concatenation distribute over interleaving.

Corollary 3.17. *For all $w_1 \in S_I(\Omega_1 ; \Omega_1')$ and $w_2 \in S_I(\Omega_2 ; \Omega_2')$ and for all $T \in I$,*

- $(w_1 \otimes w_2)\downarrow^T \sim (w_1\downarrow^T) \otimes (w_2\downarrow^T)$ *and* $(w_1 \otimes w_2)\uparrow^T \sim (w_1\uparrow^T) \otimes (w_2\uparrow^T)$.
- $(w_1\downarrow^T \otimes w_2\downarrow^T) @ (w_1\uparrow^T \otimes w_2\uparrow^T) \sim w_1 \otimes w_2$

Proof. Fix arbitrary $T' \in I$ and case split between $T' \geq T$ and $T' < T$, in either case proceed by straightforward computation. □

The interleaving operator is named after the critical setup in its construction: to obtain a singular sequence from two sequences by interleaving instantaneous steps from both sequences while maintaining monotonic ordering in time. In other words, the computational content of the proof constitutes an *algorithm* for scheduling process execution.

Computable trajectories not only facilitate a succinct definition of our logical relation, but also greatly benefit the proof of the fundamental theorem of our logical relation.

3.2.2 Logical Relation. One difficulty presented by session-typed languages is the handling of names. Channel names are runtime values that clients use to distinguish between providers. Although the name of the providing channel is recorded as part of the provider's syntax $\mathtt{proc}[a](P)$, the semantic of the provider does not, and should not, meaningfully depend on the channel name. We introduce *nameless families* to semantically capture this. Nameless families are the actual inhabitants of our logical relation.

Definition 3.18. A *nameless family of configurations* Ω is a family of configurations differing only in the choice of a single (root) process offering channel name. Formally that is for some fixed P and Ω, $\Omega[a] = \mathtt{proc}[a](P) \otimes \Omega$ for all indices a.

The definition can be point-wise extended to computable trajectories. When it is clear from the context, we may speak of nameless families of trajectories and configurations by just "trajectories" and "configurations", *resp.*

Our logical relation necessitates defining an auxiliary sort $\mathcal{A}$ of *urgent types*, inspired by [Bocchi et al. 2019]:

$$\mathcal{A} ::= \mathbf{1} \mid A_1 \otimes A_2 \mid A_1 \multimap A_2 \mid A_1 \oplus A_2 \mid A_1 \,\&\, A_2$$

Urgent types represent a session at a client-instantiated time, right before the communication has occurred. The definition is *not* recursive: the component types (if any) are regular timed session types. We can instantiate a type A at some point in time T, rendering it urgent.

Definition 3.19 (Urgency instantiation $A|_T$). Let $A|_T$ be the urgent type of A instantiated at T:

$$\begin{aligned}
\mathbf{1}^{t.p}|_T &\triangleq \mathbf{1}\\
(A_1 \otimes^{t.p} A_2)|_T &\triangleq (\{T/t\}A_1) \otimes (\{T/t\}A_2)\\
(A_1 \multimap^{t.p} A_2)|_T &\triangleq (\{T/t\}A_1) \multimap (\{T/t\}A_2)\\
(A_1 \oplus^{t.p} A_2)|_T &\triangleq (\{T/t\}A_1) \oplus (\{T/t\}A_2)\\
(A_1 \,\&^{t.p}\, A_2)|_T &\triangleq (\{T/t\}A_1) \,\&\, (\{T/t\}A_2)
\end{aligned}$$

Our logical relation distinguishes itself from tradition in that it is defined for pairs $\langle \mathbf{w}, \Omega\rangle$, called *temporal computability pairs*, where the components are nameless families of computable trajectories and configurations, satisfying $\mathbf{w} \in S_{[T,\infty)}(\Omega)$. Here Ω is the initial configuration and $\mathbf{w}$ is the evidence that it carries to substantiate its semantic property. Intuitively, the logical relation classifies initial configurations by examining their proposed trajectories at various points in time according to the types they advertise.

$$\langle \mathbf{w}, \Omega \rangle \in \mathcal{L}^{\star}[\![A^{t\cdot p}]\!] \ @\ T \quad \text{iff} \quad \forall T'.\{T'/t\}p \wedge (T' \geq T) \implies \mathbf{w}(T') \in \mathcal{V}^{\star}[\![A\,|_{T'}]\!] \ @\ T'$$
$$\langle \mathbf{w}, \Omega \rangle \in \mathcal{L}[\![A^{t\cdot p}]\!] \ @\ T \quad \text{iff} \quad \forall T'.\{T'/t\}p \implies (T' \geq T) \wedge \mathbf{w}(T') \in \mathcal{V}[\![A\,|_{T'}]\!] \ @\ T'$$

$$\Omega \in \mathcal{V}^{(\star)}[\![\mathbf{1}]\!]@\ T \quad \text{iff} \quad \forall a.\Omega[a] \xmapsto[T]{a!\mathtt{cls}} \mathbf{1}.$$

$$\Omega \in \mathcal{V}^{(\star)}[\![A_1 \ \&\ A_2]\!]@\ T \quad \text{iff} \quad \forall a\ \exists \mathbf{w}_1 \exists \Omega_1 \text{ s.t. } \Omega[a] \xmapsto[T]{a?\mathtt{left}} \Omega_1[a] \text{ and } \langle \mathbf{w}_1, \Omega_1 \rangle \in \mathcal{L}^{(\star)}[\![A_1]\!]@\ T, \text{ and}$$
$$\exists \mathbf{w}_2 \exists \Omega_2 \text{ s.t. } \Omega[a] \xmapsto[T]{a?\mathtt{right}} \Omega_2[a] \text{ and } \langle \mathbf{w}_2, \Omega_2 \rangle \in \mathcal{L}^{(\star)}[\![A_2]\!]@\ T.$$

$$\Omega \in \mathcal{V}^{(\star)}[\![A_2 \oplus A_2]\!]@\ T \quad \text{iff} \quad \forall a \text{ either}$$
$$\exists \mathbf{w}_1 \exists \Omega_1 \text{ s.t. } \Omega[a] \xmapsto[T]{a!\mathtt{left}} \Omega_1[a] \text{ and } \langle \mathbf{w}_1, \Omega_1 \rangle \in \mathcal{L}^{(\star)}[\![A_1]\!]@\ T, \text{ or}$$
$$\exists \mathbf{w}_2 \exists \Omega_2 \text{ s.t. } \Omega[a] \xmapsto[T]{a!\mathtt{right}} \Omega_2[a] \text{ and } \langle \mathbf{w}_2, \Omega_2 \rangle \in \mathcal{L}^{(\star)}[\![A_2]\!]@\ T.$$

$$\Omega \in \mathcal{V}^{(\star)}[\![A_1 \otimes A_2]\!]@\ T \quad \text{iff} \quad \forall a \exists c \exists \mathbf{w}_1 \mathbf{w}_2 \exists \Omega_1 \Omega_2 \text{ s.t. } \Omega[a] \xmapsto[T]{a!c} \Omega_1[c] \otimes \Omega_2[a],$$
$$\langle \mathbf{w}_1, \Omega_1 \rangle \in \mathcal{L}^{(\star)}[\![A_1]\!]@\ T \text{ and } \langle \mathbf{w}_2, \Omega_2 \rangle \in \mathcal{L}^{(\star)}[\![A_2]\!]@\ T.$$

$$\Omega \in \mathcal{V}[\![A_1 \multimap A_2]\!] \ @\ T \quad \text{iff} \quad \forall a \forall \mathbf{w}_1 \forall \Omega_1 \text{ s.t. if } \langle \mathbf{w}_1, \Omega_1 \rangle \in \mathcal{L}^{\star}[\![A_1]\!] \ @\ T \text{ then}$$
$$\forall c \exists \mathbf{w}_2 \exists \Omega_2 \text{ s.t. } \Omega[a] \xmapsto[T]{a?c} \Omega_2[a] \text{ and } \langle \mathbf{w}_2, \Omega_1[c] \otimes \Omega_2 \rangle \in \mathcal{L}[\![A_2]\!] \ @\ T.$$

$$\Omega \in \mathcal{V}^{\star}[\![A_1 \multimap A_2]\!] \ @\ T \quad \text{iff} \quad \forall a \forall \mathbf{w}_1 \forall \Omega_1 \text{ s.t. if } \langle \mathbf{w}_1, \Omega_1 \rangle \in \mathcal{L}[\![A_1]\!] \ @\ T \text{ then}$$
$$\forall c \exists \mathbf{w}_2 \exists \Omega_2 \text{ s.t. } \Omega[a] \xmapsto[T]{a?c} \Omega_2[a] \text{ and } \langle \mathbf{w}_2, \Omega_1[c] \otimes \Omega_2 \rangle \in \mathcal{L}^{\star}[\![A_2]\!] \ @\ T.$$

Fig. 3. Timed semantic session logical relation.

We define four (unary) relations:

$\langle \mathbf{w}, \Omega \rangle \in \mathcal{L}^{\star}[\![A]\!] \ @\ T$	Interpreting *latent* configurations to be used with evidence $\mathbf{w}$.
$\Omega \in \mathcal{V}^{\star}[\![\mathcal{A}]\!] \ @\ T$	Interpreting *urgent* configuration to be used.
$\langle \mathbf{w}, \Omega \rangle \in \mathcal{L}[\![A]\!] \ @\ T$	Interpreting *latent* configuration as provider with evidence $\mathbf{w}$.
$\Omega \in \mathcal{V}[\![\mathcal{A}]\!] \ @\ T$	Interpreting *urgent* configuration as provider.

All four relations take the current time T as an input, indicating that meaning of a type is time-dependent. Two *latent relations* classify initial configurations whose advertised services are yet to occur, therefore they require a proposed computable trajectory from Ω to justify their semantic inhabitance. They are akin to expression interpretations in traditional logical relations. On the other hand, the *urgent relations* classify configurations after a time of interaction has been picked and fixed. They are akin to value interpretations in traditional logical relations. The logical relations are defined inductively over A in Fig. 3.

3.2.3 Support for Functional Value Exchange. We briefly sketch modifications required to support communicating functional values. First, actions α are enriched with a complementary pair of actions: $a\,!\,\mathtt{val}(v)$ and $a\,?\,\mathtt{val}(v)$ for sending and receiving value v over a. Let $e \Downarrow v$ be a suitably defined big-step evaluation judgment for functional expressions. The transition rules now include transitions engendering these actions:

[D-! L] $\quad \mathtt{proc}[a](\ \mathtt{consume}^{T}\ b\ ;x\ .\ Q\ ) \xmapsto[T]{b?\mathtt{val}(v)} \mathtt{proc}[a](\ \{v/x\}Q\ )$

[D-? L] $\quad \mathtt{proc}[a](\ \mathtt{supply}^{T}\ b(v)\ ;Q\ ) \xmapsto[T]{b!\mathtt{val}(e)} \mathtt{proc}[a](\ Q\ )$ where $e \Downarrow v$

With premise $\{T/t\}p$:

[D-!] $\quad \mathtt{proc}[a](\ \mathtt{produce}^{t\cdot p}\ v\ ;P\ ) \xmapsto[T]{a!\mathtt{val}(e)} \mathtt{proc}[a](\ P\ )$ where $e \Downarrow v$

[D-?] $\quad \mathtt{proc}[a](\ \mathtt{query}^{t\cdot p}\ ;x\ .\ P\ ) \xmapsto[T]{a?\mathtt{val}(v)} \mathtt{proc}[a](\ \{v/x\}P\ )$

This completes the necessary changes to the transition system. On the logical relation side, we start by defining $!^{t.p}_{\tau}.A\,|_T \triangleq\ !_\tau \cdot \{T/t\}A$ and $?^{t.p}_{\tau}.A\,|_T \triangleq\ ?_\tau \cdot \{T/t\}A$.

The logical relation is then enriched to account for the two additional session types by adding to the $\mathcal{V}^{(\star)}[\![-]\!]@\ T$ cases:

$$\Omega \in \mathcal{V}^{(\star)}[\![\,!_\tau \cdot A]\!]@\ T \triangleq \forall a \exists \mathbf{w}_1 \exists \Omega_1.(\Omega[a] \xmapsto[T]{a!\mathtt{val}(v)} \Omega_1[a]) \wedge (\langle \mathbf{w}_1, \Omega_1\rangle \in \mathcal{L}^{(\star)}[\![A]\!]@\ T) \wedge (v \in [\![\tau]\!])$$

$$\Omega \in \mathcal{V}^{(\star)}[\![\,?_\tau \cdot A]\!]@\ T \triangleq \forall a \exists \mathbf{w}_1 \exists \Omega_1.(\Omega[a] \xmapsto[T]{a?\mathtt{val}(v)} \Omega_1[a]) \wedge \forall (v \in [\![\tau]\!]).\langle \mathbf{w}_1, \Omega_1\rangle \in \mathcal{L}^{(\star)}[\![A]\!]@\ T$$

Here $[\![\tau]\!]$ denote some suitable logical relation defined for the functional language. The exact detail of the definition depends on the language features available in the functional layer. However, for a wide range of choices it is well studied. These definitions here says that the process must be willing to send (receive) a functional value, and that it continues to behave according to A after the message exchange. If the process sends then the value sent must be well-behaving. Otherwise, it may assume that the value it received is well-behaving.

4 Automatic Verification Through Refinement Type System for TILLST

This section explores Mode of Use 1 of our logical relation, once-and-for-all verification using a type system. To facilitate this mode, we first develop a refinement type system for TILLST (§4.1) and then show that well-typed terms inhabit the logical relation (§4.2)

4.1 Refinement Type System for TILLST

Our refinement type system considers the process language introduced in §2.2.1 and assigns TILLST types to them. The resulting typing rules are shown in Fig. 4 and employ the judgment

$$\mathcal{G}\,;\mathcal{F} \mid \Delta \vdash P\ @\ T :: A.$$

The judgment differs from the usual judgment found in ILLST systems by its dependence on the temporal variables in $\mathcal{G}$ and propositions in $\mathcal{F}$. It reads as *"at time T, process term P provides a session of type A, given the typing of channels in Δ and assuming truth of the propositions in $\mathcal{F}$"*. We call attention to the dependencies on the temporal variable context $\mathcal{G}$: the linear channel context Δ, propositional context $\mathcal{F}$, the time of assertions T, and the type A are all scoped under context $\mathcal{G}$.

The time T is the time *at which the judgment is asserted*. The validity and meaning of typing thus depends on T. As usual, we give the typing rules in a *sequent calculus*, where the conclusion denotes the protocol state before the message exchange and the premises the protocol states after the message exchange. Therefore, the time *at* the conclusion must *precede* that of the premises.

Judgmentally, a type $A^{t.p}$ at time T internalizes a family of derivations indexed by time instances $t \geq T$ such that p is true. This is enforced in the right rules by asserting the premises at time variable t along with the premise $p \vdash T \leq t$. Use of the type $A^{t.p}$, on the other hand, is required to occur at some concrete time T' that is accessible from "now" and satisfies the predicate p. This is enforced in the left rules by typing the premises at a fixed future T' ($T \leq T'$) satisfying $p(T)$.

Since right rules *bind* the time of communication t for typing the providing process' continuation, premises of right rules extend $\mathcal{G}$ with the variable t and $\mathcal{F}$ with the proposition p, allowing temporal predicates of future interactions to refer to the times of past interactions. Left rules, on the other hand, update the type of the channel variable interacted with for the continuation, substituting T' for t in A.

TILLST adopts a *global* notion of time, in the sense that time expressions reference points in time that are commonly known and agreed upon. In particular, a closed time expression means the same for all processes. Furthermore, time passes at the same pace for every process. When a

$$[\textsc{Fwd}]\quad \frac{\mathcal{G}\,;\,\mathcal{F} \vdash A \bowtie A' \;@\; T}{\mathcal{G}\,;\,\mathcal{F} \mid x : A \vdash \mathtt{fwd}^{T}\, x \;@\; T :: A'}$$

$$[\textsc{Cut}]\quad \frac{\mathcal{G}\,;\,\mathcal{F} \mid \Delta_1 \vdash P \;@\; T :: A \qquad \mathcal{G}\,;\,\mathcal{F} \mid \Delta_2, x : A' \vdash Q \;@\; T :: C \qquad \mathcal{G}\,;\,\mathcal{F} \vdash A \bowtie A' \;@\; T}{\mathcal{G}\,;\,\mathcal{F} \mid \Delta_1, \Delta_2 \vdash \mathtt{spawn}^{T}\, P\,;\, x\,.\,Q \;@\; T :: C}$$

$$[\mathbf{1}\ \mathrm{R}]\quad \frac{\mathcal{G}, t\,;\,\mathcal{F}, p(t) \vdash T \leq t}{\mathcal{G}\,;\,\mathcal{F} \mid \emptyset \vdash \mathtt{close}^{t.p} \;@\; T :: \mathbf{1}^{t.p}}$$

$$[\mathbf{1}\ \mathrm{L}]\quad \frac{\mathcal{G}\,;\,\mathcal{F} \mid \Delta \vdash Q \;@\; T' :: C \qquad \mathcal{G}\,;\,\mathcal{F} \vdash T \leq T' \qquad \mathcal{G}\,;\,\mathcal{F} \vdash p(T')}{\mathcal{G}\,;\,\mathcal{F} \mid \Delta, x : \mathbf{1}^{t.p} \vdash \mathtt{wait}^{T'}\, x\,;\, Q \;@\; T :: C}$$

$$[\otimes\ \mathrm{R}]\quad \frac{\mathcal{G}, t\,;\,\mathcal{F}, p(t) \mid \Delta_1 \vdash P_1 \;@\; t :: A_1 \qquad \mathcal{G}, t\,;\,\mathcal{F}, p(t) \mid \Delta_2 \vdash P_2 \;@\; t :: A_2 \qquad \mathcal{G}, t\,;\,\mathcal{F}, p(t) \vdash T \leq t}{\mathcal{G}\,;\,\mathcal{F} \mid \Delta_1, \Delta_2 \vdash P_1 \otimes^{t.p} P_2 \;@\; T :: A_1 \otimes^{t.p} A_2}$$

$$[\otimes\ \mathrm{L}]\quad \frac{\mathcal{G}\,;\,\mathcal{F} \mid \Delta, x : \{T'/t\}A_1, y : \{T'/t\}A_2 \vdash Q \;@\; T' :: C \qquad \mathcal{G}\,;\,\mathcal{F} \vdash T \leq T' \qquad \mathcal{G}\,;\,\mathcal{F} \vdash p(T')}{\mathcal{G}\,;\,\mathcal{F} \mid \Delta, x : A_1 \otimes^{t.p} A_2 \vdash \mathtt{split}^{T'}\, x\,;\, y\,.\,Q \;@\; T :: C}$$

$$[\multimap\ \mathrm{R}]\quad \frac{\mathcal{G}, t\,;\,\mathcal{F}, p(t) \mid \Delta, x : A_1 \vdash P_2 \;@\; t :: A_2 \qquad \mathcal{G}, t\,;\,\mathcal{F}, p(t) \vdash T \leq t}{\mathcal{G}\,;\,\mathcal{F} \mid \Delta \vdash \lambda^{t.p}(x : A_1) P_2 \;@\; T :: A_1 \multimap^{t.p} A_2}$$

$$[\multimap\ \mathrm{L}]\quad \frac{\mathcal{G}\,;\,\mathcal{F} \mid \Delta_1 \vdash P \;@\; T' :: \{T'/t\}A_1 \qquad \mathcal{G}\,;\,\mathcal{F} \mid \Delta_2, x : \{T'/t\}A_2 \vdash Q \;@\; T' :: C \qquad \mathcal{G}\,;\,\mathcal{F} \vdash T \leq T' \qquad \mathcal{G}\,;\,\mathcal{F} \vdash p(T')}{\mathcal{G}\,;\,\mathcal{F} \mid \Delta_1, \Delta_2, x : A_1 \multimap^{t.p} A_2 \vdash \mathtt{app}^{T'}\, x\,(P)\,;\, Q \;@\; T :: C}$$

$$[\oplus\ \mathrm{R1}]\quad \frac{\mathcal{G}, t\,;\,\mathcal{F}, p(t) \mid \Delta \vdash P \;@\; t :: A_1 \qquad \mathcal{G}, t\,;\,\mathcal{F}, p(t) \vdash T \leq t}{\mathcal{G}\,;\,\mathcal{F} \mid \Delta \vdash \mathtt{switchL}^{t.p}\,;\, P \;@\; T :: A_1 \oplus^{t.p} A_2}$$

$$[\oplus\ \mathrm{R2}]\quad \frac{\mathcal{G}, t\,;\,\mathcal{F}, p(t) \mid \Delta \vdash P \;@\; t :: A_2 \qquad \mathcal{G}, t\,;\,\mathcal{F}, p(t) \vdash T \leq t}{\mathcal{G}\,;\,\mathcal{F} \mid \Delta \vdash \mathtt{switchR}^{t.p}\,;\, P \;@\; T :: A_1 \oplus^{t.p} A_2}$$

$$[\oplus\ \mathrm{L}]\quad \frac{\mathcal{G}\,;\,\mathcal{F} \mid \Delta, x : \{T'/t\}A_1 \vdash Q_1 \;@\; T' :: C \qquad \mathcal{G}\,;\,\mathcal{F} \mid \Delta, x : \{T'/t\}A_2 \vdash Q_2 \;@\; T' :: C \qquad \mathcal{G}\,;\,\mathcal{F} \vdash T \leq T' \qquad \mathcal{G}\,;\,\mathcal{F} \vdash p(T')}{\mathcal{G}\,;\,\mathcal{F} \mid \Delta, x : A_1 \oplus^{t.p} A_2 \vdash \mathtt{case}^{T'}\, x\,\{Q_1 \mid Q_2\} \;@\; T :: C}$$

$$[\&\ \mathrm{R}]\quad \frac{\mathcal{G}, t\,;\,\mathcal{F}, p(t) \mid \Delta \vdash P_1 \;@\; t :: A_1 \qquad \mathcal{G}, t\,;\,\mathcal{F}, p(t) \mid \Delta \vdash P_2 \;@\; t :: A_2 \qquad \mathcal{G}, t\,;\,\mathcal{F}, p(t) \vdash T \leq t}{\mathcal{G}\,;\,\mathcal{F} \mid \Delta \vdash \mathtt{offer}^{t.p}\,\{P_1 \mid P_2\} \;@\; T :: A_1 \,\&^{t.p}\, A_2}$$

$$[\&\ \mathrm{L1}]\quad \frac{\mathcal{G}\,;\,\mathcal{F} \mid \Delta, x : \{T'/t\}A_1 \vdash Q \;@\; T' :: C \qquad \mathcal{G}\,;\,\mathcal{F} \vdash T \leq T' \qquad \mathcal{G}\,;\,\mathcal{F} \vdash p(T')}{\mathcal{G}\,;\,\mathcal{F} \mid \Delta, x : A_1 \,\&^{t.p}\, A_2 \vdash \mathtt{selL}^{T'}\, x\,;\, Q \;@\; T :: C}$$

$$[\&\ \mathrm{L2}]\quad \frac{\mathcal{G}\,;\,\mathcal{F} \mid \Delta, x : \{T'/t\}A_2 \vdash Q \;@\; T' :: C \qquad \mathcal{G}\,;\,\mathcal{F} \vdash T \leq T' \qquad \mathcal{G}\,;\,\mathcal{F} \vdash p(T')}{\mathcal{G}\,;\,\mathcal{F} \mid \Delta, x : A_1 \,\&^{t.p}\, A_2 \vdash \mathtt{selR}^{T'}\, x\,;\, Q \;@\; T :: C}$$

Fig. 4. Process term typing rules of TILLST.

process advances time to communicate over some channel x in Δ, the same amount of time passes for the remaining channels in Δ. To see this consider the two types:

$$A \triangleq X \multimap^{t_1.t_0 \leq t_1 \leq t_0+15} (X \otimes^{t_2.t_2 = t_1+10} C), \quad B \triangleq X \multimap^{t_1.t_0 \leq t_1 \leq t_0+10} (X \otimes^{t_2.t_2 = t_1+10} C)$$

Both A and B accept a process of some generic type X, work on it, and then return it after 10 units of time, counting from reception of X. The difference is that B needs it within 10 units of time from now, but A can wait a bit longer. Suppose we are programming a client P:

$$\mathcal{G}\,;\,\mathcal{F} \mid x : A, y : B, z : X \vdash P \;@\; t_0 :: D.$$

Process P must send the process $z : X$ to x and y in some order. Here are four possible implementations P_i where $i = \{1, 2, 3, 4\}$:

$$P_1 \triangleq \mathtt{app}^{t_0} x\,(\mathtt{fwd}(z))\,;\mathtt{split}^{t_0+10}\,x\,;z\,.\,\mathtt{app}^{t_0+10} y\,(\mathtt{fwd}(z))\,;\mathtt{split}^{t_0+20}\,y\,;z\,.\cdots$$
$$P_2 \triangleq \mathtt{app}^{t_0+3} y\,(\mathtt{fwd}(z))\,;\mathtt{split}^{t_0+13}\,y\,;z\,.\,\mathtt{app}^{t_0+15} x\,(\mathtt{fwd}(z))\,;\mathtt{split}^{t_0+25}\,x\,;z\,.\cdots$$
$$P_3 \triangleq \mathtt{app}^{t_0+3} x\,(\mathtt{fwd}(z))\,;\mathtt{split}^{t_0+13}\,x\,;z\,.\,\mathtt{app}^{t_0+13} y\,(\mathtt{fwd}(z))\,;\mathtt{split}^{t_0+23}\,y\,;z\,.\cdots$$
$$P_4 \triangleq \mathtt{app}^{t_0+6} y\,(\mathtt{fwd}(z))\,;\mathtt{split}^{t_0+16}\,y\,;z\,.\,\mathtt{app}^{t_0+16} x\,(\mathtt{fwd}(z))\,;\mathtt{split}^{t_0+25}\,x\,;z\,.\cdots$$

P_1 and P_2 will type-check while P_3 and P_4 will not. Process P_1 immediately sends z to x, and sends z *again immediately* to y once it receives z back. It barely makes the deadline imposed by y in the second send. Process P_3, in contrast to P_1, waits 3 units of time in the first send, causing it to miss the deadline in the second send. Process P_2 switches the order between x and y, giving it a bit of slack. It chooses to wait for 3 units for the first send and an additional 2 units for the second send. Process P_4, in contrast to P_2, waits 6 units for the first send, causing the second send to miss the deadline. In all cases, as a process P_i spends time with either participant x or y, time also progresses for the other participant, reducing the window-of-communication.

The above example provides a context for us to discuss an important *asymmetry* between types as antecedents versus succedents. When an antecedent $x : A^{t.p}$ moves from T to a future T', part of the internalized derivations at T, specifically those between T and T', are no longer internalizable at this new time T', as connectives only internalize derivations concerning the future. At the same time the client loses access to these derivations because time progresses equally for both parties. For antecedents, we impose semantic requirements only for the reachable times; for succedents, we require all times under quantification to be reachable and well-behaving.

As a consequence of this asymmetry, the identity rule [Fwd] and the cut-rule [Cut] both carry an extra premise. The definitions are available in Fig. 5.

The premise $A \rtimes A' \mathbin{@} T$ (termed *forward-retyping*) in [Fwd] picks out the part of A that remains reachable at T and rewrites the type accordingly to A'. Its effect is to ensure that a forward does not happen "too late", *i.e.,* that the process providing along x must be available at least for the entire period that the forwarding process promises to be available to its client. Without it the system would be unsound. To see what goes wrong otherwise, consider the following process:

$$x : \mathbf{1}^{t.t_0 \le t \le t_0+5}, y : \mathbf{1}^{t.t_0 \le t \le t_0+5} \vdash \mathtt{wait}^{t_0+2} x\,;\mathtt{fwd}^{t_0+2} y \mathbin{@} t_0 :: \mathbf{1}^{t.t_0 \le t \le t_0+5}$$

The process starts at t_0 and waits until $t_0 + 2$ to close x, then it starts forwarding y. Itself advertises to be available $t_0 \le t \le t_0 + 5$ along its providing channel, which includes $t_0 + 1$. At this time, however, it will not be available as it is waiting to communicate with x. This is a soundness problem pertaining specifically to [Fwd].

A dual problem of *completeness* arises in the [Cut] rule. The premise $A \ltimes A' \mathbin{@} T$ (termed *cut-retyping*) in [Cut] allows the cut as long as the reachable parts of A' at T are covered by A, effectively allowing A to be cut against an A' with a broader quantification. To see why it is useful, let us consider another example. Suppose at time $t_0 + 2$ we have a provider P and client Q satisfying:

$$\Delta_1 \vdash P \mathbin{@} t_0 + 2 :: \mathbf{1}^{t.t_0+2 \le t}, \quad \Delta_2, x : \mathbf{1}^{t.t_0 \le t} \vdash Q \mathbin{@} t_0 + 2 :: C$$

Process Q expects to be able to close x at any time following t_0, and P is a process that can be closed at any time after $t_0 + 2$. However, since the typing is derived at $t_0 + 2$, the earliest possible time that Q can communicate through x is the current time $t_0 + 2$. Therefore, this cut should be permitted, despite the fact that the types provided and used are syntactically different.

$$\frac{\mathcal{G}, t\,;\mathcal{F}, q \vdash p \quad \mathcal{G}, t\,;\mathcal{F}, q \vdash T \leq t}{\mathcal{G}, t\,;\mathcal{F} \vdash \mathbf{1}^{t.p} \rtimes \mathbf{1}^{t.q}\ @\ T} \qquad \frac{\mathcal{G}, t\,;\mathcal{F}, q \vdash p \quad \mathcal{G}, t\,;\mathcal{F}, q \vdash T \leq t \quad \mathcal{G}, t\,;\mathcal{F}, q \vdash A_i \rtimes A_i'\ @\ t}{\mathcal{G}\,;\mathcal{F} \vdash A_1 \,\&^{t.p}\, A_2 \rtimes B_1 \,\&^{t.q}\, B_2\ @\ T}$$

$$\frac{\mathcal{G}, t\,;\mathcal{F}, q \vdash p \quad \mathcal{G}, t\,;\mathcal{F}, q \vdash T \leq t \quad \mathcal{G}, t\,;\mathcal{F}, q \vdash A_i \rtimes A_i'\ @\ t}{\mathcal{G}\,;\mathcal{F} \vdash A_1 \oplus^{t.p} A_2 \rtimes B_1 \oplus^{t.q} B_2\ @\ T} \qquad \frac{\mathcal{G}, t\,;\mathcal{F}, q \vdash p \quad \mathcal{G}, t\,;\mathcal{F}, q \vdash T \leq t \quad \mathcal{G}, t\,;\mathcal{F}, q \vdash A_i \rtimes A_i'\ @\ t}{\mathcal{G}\,;\mathcal{F} \vdash A_1 \otimes^{t.p} A_2 \rtimes B_1 \otimes^{t.q} B_2\ @\ T}$$

$$\frac{\mathcal{G}, t\,;\mathcal{F}, q \vdash p \quad \mathcal{G}, t\,;\mathcal{F}, q \vdash T \leq t \quad \mathcal{G}, t\,;\mathcal{F}, q \vdash A_i \rtimes A_i'\ @\ t}{\mathcal{G}\,;\mathcal{F} \vdash A_1 \multimap^{t.p} A_2 \rtimes B_1 \multimap^{t.q} B_2\ @\ T} \qquad \frac{\mathcal{G}, t\,;\mathcal{F}, T \leq t, q \vdash p}{\mathcal{G}\,;\mathcal{F} \vdash \mathbf{1}^{t.p} \ltimes \mathbf{1}^{t.q}\ @\ T}$$

$$\frac{\mathcal{G}, t\,;\mathcal{F}, T \leq t, q \vdash p \quad \mathcal{G}, t\,;\mathcal{F}, T \leq t, q \vdash A_i \ltimes A_i'\ @\ t}{\mathcal{G}\,;\mathcal{F} \vdash A_1 \,\&^{t.p}\, A_2 \ltimes B_1 \,\&^{t.q}\, B_2\ @\ T} \qquad \frac{\mathcal{G}, t\,;\mathcal{F}, T \leq t, q \vdash p \quad \mathcal{G}, t\,;\mathcal{F}, T \leq t, q \vdash A_i \ltimes A_i'\ @\ t}{\mathcal{G}\,;\mathcal{F} \vdash A_1 \oplus^{t.p} A_2 \ltimes B_1 \oplus^{t.q} B_2\ @\ T}$$

$$\frac{\mathcal{G}, t\,;\mathcal{F}, T \leq t, q \vdash p \quad \mathcal{G}, t\,;\mathcal{F}, T \leq t, q \vdash A_i \ltimes A_i'\ @\ t}{\mathcal{G}\,;\mathcal{F} \vdash A_1 \otimes^{t.p} A_2 \ltimes B_1 \otimes^{t.q} B_2\ @\ T} \qquad \frac{\mathcal{G}, t\,;\mathcal{F}, T \leq t, q \vdash p \quad \mathcal{G}, t\,;\mathcal{F}, T \leq t, q \vdash A_i \ltimes A_i'\ @\ t}{\mathcal{G}\,;\mathcal{F} \vdash A_1 \multimap^{t.p} A_2 \ltimes B_1 \multimap^{t.q} B_2\ @\ T}$$

Fig. 5. Retyping rules of TILLST.

4.2 Fundamental Theorem

To prove that all process terms with valid derivations using the rules in Fig. 4 inhabit our TSSLR, and thus are timely, we prove the fundamental theorem of the logical relation. The theorem is stated for *open terms*, allowing our program to be composed with other objects, as long as they inhabit the logical relation. We first introduce auxiliary definitions to account for open process terms, then state the fundamental theorem. Because process terms contain both free channel variables and temporal variables, both contexts need to be accounted for.

Definition 4.1 ($\delta \in \mathcal{L}^\star[\![\Delta]\!]\ @\ T$). A *sub-forest* for context Δ is a map δ from Δ to temporal computability pairs. We say $\delta \in \mathcal{L}^\star[\![\Delta]\!]\ @\ T$ iff for all $(x : A) \in \Delta$, $\delta_x \in \mathcal{L}^\star[\![A]\!]\ @\ T$

Definition 4.2 ($\Delta \gg P\ @\ T :: A$). A *(runtime) channel substitution* for Δ is a map γ from Δ to channel names. We say $\Delta \gg P\ @\ T :: A$ iff for all $\delta \in \mathcal{L}^\star[\![\Delta]\!]\ @\ T$ and for all substitutions γ, $\exists \mathbf{w}$ s.t. $\langle \mathbf{w}, (\bigotimes_{x \in \Delta} \Omega_x[\gamma(x)]) \otimes \mathtt{proc}[-](\,\hat{\gamma}(P)\,)\rangle \in \mathcal{L}[\![A]\!]\ @\ T$, where Ω_x is initial configuration of δ_x.

Definition 4.3 ($\mathcal{G}\,;\mathcal{F} \mid \Delta \gg P\ @\ T :: A$). Let φ be any assignment of time variables $t \in \mathcal{G}$ such that φ satisfies $\mathcal{F}$. Let $\hat{\varphi}$ be the substitution function induced by the assignment. Then $\hat{\varphi}(\Delta) \gg \hat{\varphi}(P)\ @\ \hat{\varphi}(T) :: \hat{\varphi}(A)$

Theorem 4.4 (FTLR). *If $\mathcal{G}\,;\mathcal{F} \mid \Delta \vdash P\ @\ T :: A$ then $\mathcal{G}\,;\mathcal{F} \mid \Delta \gg P\ @\ T :: A$.*

Proof in the full paper [Yao et al. 2024b]. Additionally, the usual forward and backward closure property is now generalized to account for the passage of time. For the forward direction, given $\langle \mathbf{w}, \Omega\rangle \in \mathcal{L}^\star[\![A]\!]\ @\ T$, the right partition of the computable trajectories at any future time T' inhabits the same type at T'. Dually, for the backwards direction, inhabitance in the type is preserved for any past time as long as there is a way to extend the trajectories to that past time.

Lemma 4.5 (Forward and backward closure). *(Proof in the full paper [Yao et al. 2024b])*

- *If* $\langle \mathbf{w}, \Omega \rangle \in \mathcal{L}^\star[\![A]\!] \mathbin{@} T$ *then* $\forall T'.(T' \geq T) \implies \langle \mathbf{w} \uparrow^{T'}, \mathbf{w}(T') \rangle \in \mathcal{L}^\star[\![A]\!] \mathbin{@} T'$
- *If* $\mathbf{w}_1 \in S_{[T',T)}(\Omega_1 ; \Omega_2)$ *and* $\langle \mathbf{w}_2, \Omega_2 \rangle \in \mathcal{L}[\![A]\!] \mathbin{@} T$, *then* $\langle (\mathbf{w}_1 \mathbin{@} \mathbf{w}_2), \Omega_1 \rangle \in \mathcal{L}[\![A]\!] \mathbin{@} T'$.

Semantically, the retyping relations $A \rtimes B \mathbin{@} T$ and $A \ltimes B \mathbin{@} T$ allow us to translate between $\mathcal{L}^\star[\![-]\!] \mathbin{@} T$ and $\mathcal{L}[\![-]\!] \mathbin{@} T$, captured in the following two lemmas:

Lemma 4.6 (Semantic retyping). *(Proof of its generalization in the full paper [Yao et al. 2024b])*

- *If* $\langle \mathbf{w}, \Omega \rangle \in \mathcal{L}^\star[\![A]\!] \mathbin{@} T$ *and* $A \rtimes B \mathbin{@} T$ *then* $\langle \mathbf{w}, \Omega \rangle \in \mathcal{L}[\![B]\!] \mathbin{@} T$.
- *If* $\langle \mathbf{w}, \Omega \rangle \in \mathcal{L}[\![A]\!] \mathbin{@} T$ *and* $A \ltimes B \mathbin{@} T$ *then* $\langle \mathbf{w}, \Omega \rangle \in \mathcal{L}^\star[\![B]\!] \mathbin{@} T$.

The proofs of Lem. 4.5 and Lem. 4.6 greatly benefit from the abstractions afforded by computable trajectories and are essentially carried out by equational reasoning in terms of the trajectory algebra.

As an immediate result of Thm. 4.4, we can prove the following adequacy theorem for closed terms of type $\mathbf{1}^{t.p}$:

Theorem 4.7 (Adequacy). *If* $P \mathbin{@} \mathtt{init} :: \mathbf{1}^{t.t=\mathtt{init}+\overline{n}}$, *then*
$\exists \sigma.\sigma : \textcircled{\scriptsize t}_{\mathtt{init}}, \mathtt{proc}[a](P) \longmapsto^* \textcircled{\scriptsize t}_{\mathtt{init}+\overline{n}}, \Omega$, *and for some* Ω *s.t.* $\Omega \xmapsto[\mathtt{init}+\overline{n}]{a!\mathtt{cls}} \mathbf{1}$,

Proof. By the fundamental theorem (Thm. 4.4) we have $\emptyset \gg P \mathbin{@} T :: A$ because temporal context is empty. There exists $\mathbf{w}$ s.t. $\langle \mathbf{w}, \Omega \rangle \in \mathcal{L}[\![\mathbf{1}^{t.t=\mathtt{init}+\overline{n}}]\!] \mathbin{@} \mathtt{init}$. where $\Omega = \mathtt{proc}[-](P)$. Let $T \triangleq \mathtt{init} + \overline{n}$. Therefore, $\mathbf{w}(T) \in \mathcal{V}[\![\mathbf{1}]\!] \mathbin{@} T$. That is for any a, $\mathbf{w}(T)[a] \xmapsto[T]{a!\mathtt{cls}} \mathbf{1}$. Obtain the desired sequence by additionally consulting $\mathbf{w}[a] \downarrow^T \in S_{[\mathtt{init},\mathtt{init}+\overline{n})}(\mathtt{proc}[a](P) ; w(T))$. □

Because silent transitions can only occur at a time satisfying their respective predicate as imposed by the process term, existence of a sequence σ suffices to ensure that no process missed its "deadline" during the computation. In particular, Thm. 4.7 entails deadlock-freedom as well as termination for the process term P in question.

Support for Functional Value Exchange. We briefly sketch the necessary changes to support functional value exchange. For the type system, we will introduce another context Γ containing assumptions of form $x : \tau$, asserting x holds a functional value of type τ. The context Γ is structural, and assumptions within may be contracted and weakened at will. This modification will be propagated throughout the system. On the fundamental theorem side, Def. 4.3 will be modified to include substitutions for functional variables. This change propagates throughout the proofs.

5 Whole System Manual Verification: TSSLR In Action

In this section, we will once more revisit the smart home example introduced in §1 and §2.1. We start by reviewing our progress so far: In §2, we distilled TILLST types representing the protocols of the sensor ($A_{\mathtt{BME680}}$) and controller (A_{Hub}) and provided a process implementation (P_{Hub}). With the rules in §4.1, we can easily check (*e.g.,* using our type checker in §6) that our implementation inhabits the proposed type ($P_{\mathsf{Hub}} \mathbin{@} t_0 :: A_{\mathsf{Hub}}$). By the fundamental theorem (Thm. 4.4), proved in §4.2, we conclude that P_{Hub} adheres to the protocol prescribed by A_{Hub}. That is:

$$t_0 ; \cdot \mid \cdot \gg P_{\mathsf{Hub}} \mathbin{@} t_0 :: A_{\mathsf{Hub}}$$

This is where we are right now.

However, clients of the hub device would appreciate an end-to-end *whole system* guarantee: when the controller is connected to the sensors, the entire, heterogeneous system will still behave according to the protocol. Concretely, suppose that we have protocol-adhering sensors connected

on channels a and b, represented by some configuration Ω_a and Ω_b. We would like to ensure that the configuration

$$\Omega_0 \triangleq \Omega_a \otimes \Omega_b \otimes \mathtt{proc}[-](\ \mathtt{spawn}^T P_{\mathsf{Hub}}\ ;\ x\ .\ \mathtt{app}^T x\ (\mathtt{fwd}^T a)\ ;\ \mathtt{app}^T x\ (\mathtt{fwd}^T b)\ ;\ \mathtt{fwd}^T x\ )$$

adheres to the type $!^{\,t_3.T+50\ ms \le t_3}_{\mathtt{bool}}.\ \mathbf{1}^{t_4.t_4=t_3}$. Formally, that is:

$$\exists \mathbf{w}.\langle \mathbf{w}, \Omega_0 \rangle \in \mathcal{L}[\![\ !^{\,t_3.T+50\ ms \le t_3}_{\mathtt{bool}}.\ \mathbf{1}^{t_4.t_4=t_3}\ ]\!]\ @\ T$$

The goal of this section is to provide such a whole-system guarantee.

Our immediate obstacle is that we do not have a representation for these sensors. The sensors are unlikely to be programmed in the process language proposed in §2.2.1. Additionally, we do not have, and will not have, access to the code inside the sensors. On top of this, the origins of the timing requirements are often non-computational (*e.g.,* warming up an internal component), therefore it is likely infeasible to model the sensor using terms in our calculus.

What we are given is an operational manual (*datasheet*) for the sensor. These documents often model the operation of the sensor as a *state machine*, either directly with a figure or indirectly with textual descriptions. In the case where timing is relevant, state machines can be enriched with timing information, resulting in Timed Automata [Alur and Dill 1994]. Fig. 6 shows the state machine that we extracted from the BME680 specification [Bosch 2024].

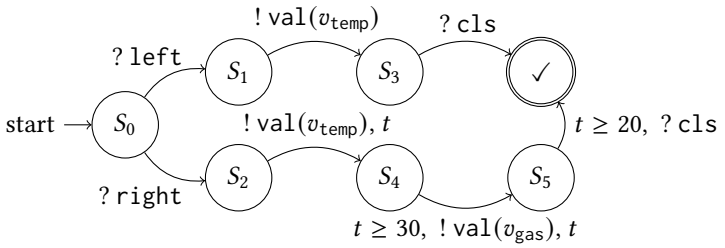

Fig. 6. BME680 Sensor specified using Timed Automaton

We provide a quick introduction on timed automata. Timed automaton come with a sequence of named timers, termed *clocks*. The initial reading of the clock is always zero. As one steps through the states, the clocks constantly tick up. Here, we just have one clock t. Transitions in a timed automaton have three parts: a clock condition, an action, and a set of clocks to "reset". For example, for the transition $S_4 \xrightarrow[t \geq 30, !\mathtt{val}(v_{\mathtt{gas}}), t]{} S_5$, the condition $t \geq 30$ specifies that the transition is only enabled if the clock reads above or equal to 30; the action of this transition is to send a functional value for air quality; finally clock t is reset (becomes zero) after taking this transition. If the condition part of the transition is missing, then the transition is always enabled. If clock resets are omitted, then no clock will be reset. For this transition, because the previous transition from S_3 to S_4 resets the clock t, this effectively means that this transition must wait for 30 *ms*. As one can see, this style of modeling is imperative in the treatment of time. Clocks in this setting can be viewed as shared variables, incremented and accessed periodically and concurrently with the executing process.

We can faithfully represent this automaton in our system by making slight adjustments. First, we enrich our configuration syntax with a new process form $\mathtt{sensor}(\ a;S_i[T]\ )$, representing an instance of the automaton that is currently at state S_i and entered S_i at time T. This induces other necessary, but inessential changes to structural congruence. The definition of nameless configurations also needs to be extended to include this new process form. To represent the transitions, we introduce the new transition rules in Fig. 7. Each transition corresponds to exactly one rule.

$$
\begin{array}{llll}
[S_0 - L] & T \le T_1 & / & \mathtt{sensor}(\, a\,;S_0[T]\,) \xmapsto[T_1]{a?\mathtt{left}} \mathtt{sensor}(\, a\,;S_1[T_1]\,) \\
[S_0 - R] & T \le T_1 & / & \mathtt{sensor}(\, a\,;S_0[T]\,) \xmapsto[T_1]{a?\mathtt{right}} \mathtt{sensor}(\, a\,;S_2[T_1]\,) \\
[S_1] & T \le T_1 \quad v : \tau_{\mathtt{temp}} & / & \mathtt{sensor}(\, a\,;S_1[T]\,) \xmapsto[T_1]{a?\mathtt{val}(v_{\mathtt{temp}})} \mathtt{sensor}(\, a\,;S_3[T_1]\,) \\
[S_3] & T \le T_1 & / & \mathtt{sensor}(\, a\,;S_3[T]\,) \xmapsto[T_1]{a?\mathtt{cls}} \mathbf{1} \\
[S_2] & T \le T_1 \quad v : \tau_{\mathtt{temp}} & / & \mathtt{sensor}(\, a\,;S_2[T]\,) \xmapsto[T_1]{a!\mathtt{val}(v_{\mathtt{temp}})} \mathtt{sensor}(\, a\,;S_4[T_1]\,) \\
[S_4] & T + 30 \le T_1 \quad v : \tau_{\mathtt{gas}} & / & \mathtt{sensor}(\, a\,;S_3[T]\,) \xmapsto[T_1]{a!\mathtt{val}(v_{\mathtt{gas}})} \mathtt{sensor}(\, a\,;S_5[T_1]\,) \\
[S_5] & T + 20 \le T_1 & / & \mathtt{sensor}(\, a\,;S_5[T]\,) \xmapsto[T_1]{a?\mathtt{cls}} \mathbf{1}
\end{array}
$$

Fig. 7. Rules for representing BME680 sensor

The remainder of the argument goes as follows:

(1) Show that the $\mathtt{sensor}(\ -\,;S_0[T]\,)$ can be semantically assigned the type $A_{\mathtt{BME680}}$.
$$\exists \mathbf{w} \text{ s.t. } \langle \mathbf{w}, \mathtt{sensor}(\ -\,;S_0[T]\,)\rangle \in \mathcal{L}[\![A_{\mathtt{BME680}}]\!] \ @\ T$$
(2) Observe that by typing rules (recall that temporal context is empty, both x, y has type $A_{\mathtt{BME680}}$):
$$\ldots \vdash \mathsf{spawn}^T P_{\mathsf{Hub}}\,;z\,.\,\mathsf{app}^T z\,(\mathsf{fwd}^T x)\,;\mathsf{app}^T z\,(\mathsf{fwd}^T y)\,;\mathtt{fwd}^T z \ @\ T :: {!}^{t_3.T+50\ ms\le t_3}_{\mathtt{bool}}.\,\mathbf{1}^{t_4.t_4=t_3}$$
(3) Appeal to the result of Thm. 4.4 with channel substitution $[x \mapsto a, y \mapsto b]$ and subforest $[x \mapsto \mathtt{sensor}(\, a\,;S_0[T]\,), y \mapsto \mathtt{sensor}(\, b\,;S_0[T]\,)]$ and conclude what we want to show.

The challenges of the proof concentrate in the first step. Towards it, we provide the following proof sketch. The idea is that we build up our proof by consecutively analyzing each state, by proving the following sub-goals in order:

(1) $\exists \mathbf{w}$ s.t. $\langle \mathbf{w}, \mathtt{sensor}(\ -\,;S_5[T]\,)\rangle \in \mathcal{L}[\![\mathbf{1}^{t_4.T+20\le t_4}]\!] \ @\ T$
(2) $\exists \mathbf{w}$ s.t. $\langle \mathbf{w}, \mathtt{sensor}(\ -\,;S_4[T]\,)\rangle \in \mathcal{L}[\![{!}^{t_3.T+30\le t_3}_{\tau_{\mathtt{Gas}}}.\,\mathbf{1}^{t_4.t_3+20\le t_4}]\!] \ @\ T$
(3) $\exists \mathbf{w}$ s.t. $\langle \mathbf{w}, \mathtt{sensor}(\ -\,;S_2[T]\,)\rangle \in \mathcal{L}[\![{!}^{t_2.T\le t_2}_{\tau_{\mathtt{Temp}}}.\ {!}^{t_3.t_2+30\le t_3}_{\tau_{\mathtt{Gas}}}.\,\mathbf{1}^{t_4.t_3+20\mathtt{ms}\le t_4}]\!] \ @\ T$
(4) $\exists \mathbf{w}$ s.t. $\langle \mathbf{w}, \mathtt{sensor}(\ -\,;S_3[T]\,)\rangle \in \mathcal{L}[\![\mathbf{1}^{t_3.T\le_3}]\!] \ @\ T$
(5) $\exists \mathbf{w}$ s.t. $\langle \mathbf{w}, \mathtt{sensor}(\ -\,;S_1[T]\,)\rangle \in \mathcal{L}[\![{!}^{t_2.T\le t_2}_{\tau_{\mathtt{Temp}}}.\,\mathbf{1}^{t_3.t_2\le t_3}]\!] \ @\ T$

These goals analyze the automaton in reverse topological order, each building up from before. In each case, the proof mostly constitutes unfolding definitions and making observations. Two exemplary proof cases can be found in the full paper [Yao et al. 2024b].

6 Rust Implementation

We implemented our refinement type system for TILLST as a DSL for Rust, whose syntax can be found in the full paper [Yao et al. 2024b]. Our implementation includes a type checker, but no code generation. We chose Rust for its strong support for systems applications in both language design and tooling, witnessed by several prior session type encodings [Chen et al. 2022; Jespersen et al. 2015]. While those encodings remain within the Rust type system, we opted for a DSL to support temporal predicates. Fig. 8 shows the DSL type of sensor hub from § 1.2.2. Macro `!rtsm{...}` delimits DSL blocks that the Rust compiler passes to our parser and type checker. Types generally take the form of `TyOp < t where p, ...>`, where `TyOp` is the name of the type operator, `t where p` represents predicate $t.p$, and the remaining arguments are continuation

session types. Type operators `ExChoice`, `Lolli`, `Produce` and `Unit`, stands for $\oplus$, $\multimap$, !, and $\mathbf{1}$ *resp.* We apply the typing rules from §4.1 in a syntax-directed traversal of the process and use Rust for the functional layer.

```
!rtsm { type BME680 = ExChoice <t1 where Geq<t1, t0>, TEMP, TEMP_AIR>
        type HUB    = Lolli <t1 where Leq<t0, t1>, BME680,
                        Lolli <t2 where Eq<t2, t1>, BME680,
                        Produce <sort_bool, t3 where Leq<Shift<t1, 50>, t3>,
                        Unit <t4 where Eq<t4, t3>>>>>                    ...}
```

Fig. 8. The sensor hub type in the TILLST Rust DSL

The challenge in implementing TILLST is the temporal judgment $\mathcal{G}\,;\mathcal{F} \vdash p$. In the full paper [Yao et al. 2024b] we establish an encoding of our temporal predicates into FOL. Our implementation uses this to generate queries to an SMT solver. We encode the temporal model via sorts for times and durations with assertions of the axioms. On the provider side, we check the judgment $\mathcal{G}, t\,;\mathcal{F}, p(t) \vdash T \leq t$ to ensure the provider is not too late. On the client side, we must check two judgments. First, we examine if the client communicates at the right time: $\mathcal{G}\,;\mathcal{F} \vdash T \leq T'$. Then, we ensure the communication can go forward in time: $\mathcal{G}\,;\mathcal{F} \vdash p(T')$. We thus encode $\mathcal{G}\,;\mathcal{F} \vdash p$ as the question: *is there an assignment of temporal variables in which $\mathcal{F}$ and not p?* An `unsat` result is then interpreted as validating the judgment, otherwise a type error is generated. For each judgment, the type checker writes out a query in the solver-agnostic SMT-LIB2 format and invokes the solver with a timeout. This yields a sound decision, but one that may be incomplete.

In practice cvc5[Barbosa et al. 2022], our choice of SMT solver, is capable of answering the queries needed to type check the range of examples we have implemented. Examples include the keyless entry protocol on modern automobiles [Wouters et al. 2019] and a radar collision detector for airplane traffic control [Kalibera et al. 2009]. These examples, including the running smart home example, can be found in the full paper [Yao et al. 2024b].

7 Related Work

Logical Relations for Session Types. Prior work on logical relations for session types is relatively young, starting out with unary logical relations for proving termination [DeYoung et al. 2020; Pérez et al. 2012, 2014] and then tackling binary logical relations for proving parametricity [Caires et al. 2013] and noninterference [Balzer et al. 2023; Derakhshan et al. 2021, 2024; van den Heuvel et al. 2024]. Except for the work by van den Heuvel et al. [2024], which targets cyclic process networks based on classical linear logic session types, all the remaining logical relations are developed for intuitionistic linear logic session types, like ours. Our logical relation is most closely related to the unary logical relations for termination; TSSLR asserts not only termination, and thus deadlock freedom, but also *timeliness*. In contrast to any existing logical relations for session types, TSSLR does not require its inhabitants to be syntactically well-typed. As result, our work facilitates *semantic typing* and enables both once-and-for-all verification, given a type system, and per-instance verification of foreign code; both modes are indispensable in our target domain.

Being entirely semantic, our logical relation builds on the foundations laid by Constable et al. [1986]; Martin-Löf [1982]; Timany et al. [2024], brought to scale in the context of the Iris framework [Jung et al. 2018b]. Iris has fueled a multitude of verification efforts, contributing Iris-based program logics targeting per-instance verification of functional program correctness. In this context, we highlight RustBelt by Jung et al. [2018a], which combines both modes of use of logical relations to prove the Rust core language and selected libraries memory safe and race free.

Intuitionistic Metric Temporal Logic (IMTL). Our refinement type system for TILLST is related to work by de Sá et al. [2023] on Intuitionistic Metric Temporal Logic (IMTL), an intuitionistic account of Metric Temporal Logic (MTL) [Koymans 1990; Ouaknine and Worrell 2005]. Metric Temporal Logic (MTL) [Koymans 1990; Ouaknine and Worrell 2005] extends linear temporal logic (LTL) [Pnueli 1977] with temporal intervals. Rather than interpreting propositions over models, as is done in prior work on MTL, de Sá et al. view temporal logic through the lens of the *propositions-as-types paradigm*, focusing on how propositions are proved. A similar endeavor has been undertaken prior by Kojima and Igarashi [2011], albeit for linear temporal logic (LTL) [Pnueli 1977] and a reduced set of temporal modalities. The technical contributions by de Sá et al. comprise a syntactic proof of cut elimination, entailing not only consistency of the logic, but also temporal causality ("future events cannot affect the present") and temporal monotonicity ("a proof can never move backwards in time"). Similarly to TILLST, the authors assume an *instant-based* model of time, witnessed by the fact that cut reductions happen at the judgmental present time. However, IMTL was conceived as a logic, with cut reductions as the primary notion of computation. Our refinement type system for TILLST, in contrast, relies on an actual execution dynamics, which really infuses meaning to an instant-based model of time. Our dynamics is also fundamental to our semantic typing approach, allowing us to show inhabitance of terms with or without a typing derivation.

Temporal Session Types. Our refinement type system for TILLST is also more distantly related to extension of Honda-style session type systems, both binary [Honda 1993; Honda et al. 1998] and multiparty [Honda et al. 2008], to support timing constraints. Adding the notion of a delay, they connect session types to communicating timed automata [Alur and Dill 1994; Bengtsson et al. 1995; Krcál and Yi 2006; Lampka et al. 2009].

In the multiparty session types setting, Bocchi et al. [2014] and Bartoletti et al. [2017] consider temporal guards on communications. These systems assume access only to local clocks and a fixed view of durations as rational numbers. Our type system is defined with respect to a single global clock abstracted over a model of time. Temporal predicates may reference the time of any prior event in the protocol. This matches our domain, where IoT and wireless device hardware maintains synchronized clocks with known bounds on drift, *e.g.,* [Bluetooth SIG 2023]. Work by Neykova et al. [2017] uses a corresponding extension of the Scribble protocol description language [Honda et al. 2011] to build runtime monitoring of protocol adherence. Multiparty reactive sessions (MRS) by Cano et al. [2019] take a global view of time as discretized instants. This synchronous reactive programming model of time has found applications to embedded systems problems via languages such as Esterel [Berry 1999; Berry and Gonthier 1992] and Lustre [Halbwachs et al. 1991]. MRS connects logical instants to the external world via reaction to events. This means constraints cannot be directly specified using physical notions of time. Work by Brun and Dardha [2023] uses the concept of timeouts to model message delivery failure, but similarly leaves correspondence between timeout and physical time intentionally unspecified.

Bocchi et al. [2019] extend temporal guards to asynchronous binary session types. This too gives access only to local views of time. The choice of asynchrony also leads to a subtyping relation that is covariant on output and contravariant on input times. In a synchronous system, the time both participants communicate must coincide. Safely substituting types in our synchronous system requires retyping relations, which distinguish the client and provider roles. For systems applications such as in §2, the distinction of which side may be more permissive is critical to correctness.

Rate-based session types (RBST) [Iraci et al. 2023] introduces a periodic construct to binary session types that specifies parts of the protocol repeat at a fixed interval. Timing in this system is attached only to control flow, with no association to specific communications. This is insufficient to express narrower constraints on specific events, such as the exchange in §2. TILLST is also able

to express dynamic change to the connectivity in protocols, i.e. spawning of new processes, and supports higher-order channels. It does this while still maintaining desirable properties of ILLST, such as deadlock freedom, which RBST lacks.

Verification and Modeling of Embedded, Control, or Hybrid Systems. Generally, when it comes to verification of computational systems, approaches can be broadly divided into external and internal/integrated methods. External methods employed by tools such as UPPAAL [Bengtsson et al. 1995] separate the implementation of the system from the modeling, specification, and proof. The modeling may be approached by timed-automata, and the logic for specification can be a domain specific logic such as (Differential) Dynamic Logic [Harel 1979; Platzer 2008]. Our type theoretic method offers a different, internal and integrated approach, where the implementation and the verification conditions are designed, expressed, and proved in unison in a singular language. Concretely, through logical relations, type checking serves the role of program verification. Our methods enjoy compositionality and support higher-order features (*e.g.,* sending channel names).

While our approach is internal, the technical development supplements and embraces external methods; it is synergistic with external verification methods thanks to the use of a logical relation. As the example in §5 demonstrates, the modeling of the sensor and the proof of the verification conditions as dictated by the logical relation can be approached with any number of external verification methods. Therefore, our approach takes an inclusive and constructive stance when it comes to existing verification works.

8 Conclusions

This paper contributes a compositional framework to enable the *verification* of *timed message-passing* systems, such as IoT applications and real-time systems. The framework consist of a *(a)* language to specify timed protocols, TILLST, rooted in intuitionistic linear logic session types, a *(b)* timed labelled transition system to characterize how programs run, and a *(c)* logical relation, TSSLR, to prove programs compliant with their specifications. To cater to the heterogeneity of its application domain, the paper adopts a *semantic typing* approach, freeing programs to be proved correct from any well-typedness constraint. As a result, the TSSLR can be used in two modes: once-and-for-all verification, given a type system, and per-instance verification of foreign code. The paper illustrates both modes based on the example of an IoT application, using a prototype implementation for the type-based verification.

There exist various avenues to be explored as part of future work. Most immediate is support of recursive behavior in terms of coinductive types, which have been shown to integrate smoothly with a Curry-Howard interpretation of linear session types [Derakhshan and Pfenning 2022; Lindley and Morris 2016], yet must be given a semantic typing interpretation. Another interesting future research direction is to integrate the framework with a cost model to bound the execution time of internal computation. Existing work [Das et al. 2018a,b] in the context of intuitionistic linear session types may serve as a valuable starting point, but again, will have to be endowed with a semantic typing interpretation.

Data-Availability Statement

Code and examples in §6 are available in the accompanied artifact [Yao et al. 2024a].

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. (2211996 and 2211997) and upon work supported by the Air Force Office of Scientific Research under award number FA9550-21-1-0385 (Tristan Nguyen, program manager). Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation or the U.S. Department of Defense.

References

Rajeev Alur and David L. Dill. 1994. A Theory of Timed Automata. *Theoretical Computer Science* 126, 2 (1994), 183–235. https://doi.org/10.1016/0304-3975(94)90010-8

Stephanie Balzer, Farzaneh Derakhshan, Robert Harper, and Yue Yao. 2023. Logical Relations for Session-Typed Concurrency. *CoRR* abs/2309.00192 (2023). https://doi.org/10.48550/ARXIV.2309.00192 arXiv:2309.00192

Haniel Barbosa, Clark W. Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. 2022. cvc5: A Versatile and Industrial-Strength SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 13243)*, Dana Fisman and Grigore Rosu (Eds.). Springer, 415–442. https://doi.org/10.1007/978-3-030-99524-9_24

Massimo Bartoletti, Tiziana Cimoli, and Maurizio Murgia. 2017. Timed Session Types. *Logical Methods in Computer Science* 13, 4 (2017), 1–47. https://doi.org/10.23638/LMCS-13(4:25)2017

Johan Bengtsson, Kim Guldstrand Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. 1995. UPPAAL - a Tool Suite for Automatic Verification of Real-Time Systems. In *DIMACS/SYCON Workshop on Verification and Control of Hybrid Systems (Lecture Notes in Computer Science, Vol. 1066)*, Rajeev Alur, Thomas A. Henzinger, and Eduardo D. Sontag (Eds.). Springer, 232–243. https://doi.org/10.1007/BFB0020949

Nick Benton and Chung-Kil Hur. 2009. Biorthogonality, Step-Indexing and Compiler Correctness. In *14th ACM SIGPLAN International Conference on Functional Programming (ICFP)*. ACM, 97–108. https://doi.org/10.1145/1596550.1596567

Gérard Berry. 1999. *The Constructive Semantics of Pure Esterel.* Technical Report. École des Mines de Paris and INRIA.

Gérard Berry and Georges Gonthier. 1992. The Esterel Synchronous Programming Language: Design, Semantics, Implementation. *Science of Computer Programming* 19, 2 (1992), 87–152. https://doi.org/10.1016/0167-6423(92)90005-V

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 - From Duality to Time-Sensitive Processes. In *28th European Symposium on Programming (ESOP) (Lecture Notes in Computer Science, Vol. 11423)*. Springer, 583–610. https://doi.org/10.1007/978-3-030-17184-1_21

Laura Bocchi, Weizhen Yang, and Nobuko Yoshida. 2014. Timed Multiparty Session Types. In *25th International Conference on Concurrency Theory (CONCUR) (Lecture Notes in Computer Science, Vol. 8704)*. Springer, 419–434. https://doi.org/10.1007/978-3-662-44584-6_29

Bosch. 2024. Gas Sensor BME680. https://www.bosch-sensortec.com/products/environmental-sensors/gas-sensors/bme680/.

Matthew Alan Le Brun and Ornela Dardha. 2023. MAGπ: Types for Failure-Prone Communication. In *32nd European Symposium on Programming (ESOP) (Lecture Notes in Computer Science, Vol. 13990)*. Springer, 363–391. https://doi.org/10.1007/978-3-031-30044-8_14

Luís Caires, Jorge A. Pérez, Frank Pfenning, and Bernardo Toninho. 2013. Behavioral Polymorphism and Parametricity in Session-Based Communication. In *22nd European Symposium on Programming (ESOP) (Lecture Notes in Computer Science, Vol. 7792)*. Springer, 330–349. https://doi.org/10.1007/978-3-642-37036-6_19

Luís Caires and Frank Pfenning. 2010. Session Types as Intuitionistic Linear Propositions. In *21th International Conference onf Concurrency Theory (CONCUR) (Lecture Notes in Computer Science, Vol. 6269)*. Springer, 222–236. https://doi.org/10.1007/978-3-642-15375-4_16

Mauricio Cano, Ilaria Castellani, Cinzia Di Giusto, and Jorge A. Pérez. 2019. *Multiparty Reactive Sessions.* Technical Report 9270. INRIA.

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

Adam Chlipala. 2007. A Certified Type-Preserving Compiler from Lambda Calculus to Assembly Language. In *28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 54–65. https://doi.org/10.1145/1250734.1250742

Robert L. Constable, Stuart F. Allen, Mark Bromley, Rance Cleaveland, J. F. Cremer, Robert Harper, Douglas J. Howe, Todd B. Knoblock, Nax Paul Mendler, Prakash Panangaden, James T. Sasaki, and Scott F. Smith. 1986. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice Hall. http://dl.acm.org/citation.cfm?id=10510

Ankush Das, Jan Hoffmann, and Frank Pfenning. 2018a. Parallel Complexity Analysis with Temporal Session Types. *Proceedings of the ACM on Programming Languages* 2, ICFP (2018), 91:1–91:30. https://doi.org/10.1145/3236786

Ankush Das, Jan Hoffmann, and Frank Pfenning. 2018b. Work Analysis with Resource-Aware Session Types. In *33rd Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. ACM, 305–314. https://doi.org/10.1145/3209108.3209146

Siddharth Sankar Das, Nabajit Deka, Nishant Sinha, Sourav Dhar, Dipanjan Bhattacharjee, and Shantanu Gupta. 2012. Environmental monitoring using sensor data fusion. In *2012 International Conference on Radar, Communication and Computing (ICRCC)*. 83–86. https://doi.org/10.1109/ICRCC.2012.6450552

Luiz de Sá, Bernardo Toninho, and Frank Pfenning. 2023. Intuitionistic Metric Temporal Logic. In *International Symposium on Principles and Practice of Declarative Programming (PPDP)*. ACM, 9:1–9:13. https://doi.org/10.1145/3610612.3610621

Farzaneh Derakhshan, Stephanie Balzer, and Limin Jia. 2021. Session Logical Relations for Noninterference. In *36th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. IEEE Computer Society, 1–14. https://doi.org/10.1109/LICS52264.2021.9470654

Farzaneh Derakhshan, Stephanie Balzer, and Yue Yao. 2024. Regrading Policies for Flexible Information Flow Control in Session-Typed Concurrency. In *38th European Conference on Object-Oriented Programming (ECOOP) (LIPIcs, Vol. 313)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 11:1–11:29. https://doi.org/10.4230/LIPICS.ECOOP.2024.11

Farzaneh Derakhshan and Frank Pfenning. 2022. Circular Proofs as Session-Typed Processes: A Local Validity Condition. *Logical Methods in Computer Science* 18, 2 (2022), 8:1–8:51. https://doi.org/10.46298/LMCS-18(2:8)2022

Henry DeYoung, Frank Pfenning, and Klaas Pruiksma. 2020. Semi-Axiomatic Sequent Calculus. In *5th International Conference on Formal Structures for Computation and Deduction (FSCD) (LIPIcs, Vol. 167)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 29:1–29:22. https://doi.org/10.4230/LIPIcs.FSCD.2020.29

Jean-Yves Girard. 1972. *Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur*. Ph. D. Dissertation. Université Paris VII.

Nicolas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. 1991. The Synchronous Data Flow Programming Language LUSTRE. *Proc. IEEE* 79, 9 (1991), 1305–1320. https://doi.org/10.1109/5.97300

David Harel. 1979. *First-Order Dynamic Logic*. Springer-Verlag, Berlin, Heidelberg.

Kohei Honda. 1993. Types for Dyadic Interaction. In *4th International Conference on Concurrency Theory (CONCUR) (Lecture Notes in Computer Science, Vol. 715)*. Springer, 509–523. https://doi.org/10.1007/3-540-57208-2_35

Kohei Honda, Aybek Mukhamedov, Gary Brown, Tzu-Chun Chen, and Nobuko Yoshida. 2011. Scribbling Interactions with a Formal Foundation. In *7th International Conference on Distributed Computing and Internet Technology (ICDCIT) (Lecture Notes in Computer Science, Vol. 6536)*. Springer, 55–75. https://doi.org/10.1007/978-3-642-19056-8_4

Kohei Honda, Vasco Thudichum Vasconcelos, and Makoto Kubo. 1998. Language Primitives and Type Discipline for Structured Communication-Based Programming. In *7th European Symposium on Programming (ESOP) (Lecture Notes in Computer Science, Vol. 1381)*. Springer, 122–138. https://doi.org/10.1007/BFb0053567

Kohei Honda, Nobuko Yoshida, and Marco Carbone. 2008. Multiparty Asynchronous Session Types. In *35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM, 273–284. https://doi.org/10.1145/1328438.1328472

Atsushi Igarashi and Naoki Kobayashi. 2001. A Generic Type System for the Pi-calculus. In *28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM, 128–141. https://doi.org/10.1145/360204.360215

Grant Iraci, Cheng-En Chuang, Raymond Hu, and Lukasz Ziarek. 2023. Validating IoT Devices with Rate-Based Session Types. *Proceedings of the ACM on Programming Languages* 7, OOPSLA2, Article 278 (2023), 1589–1617 pages. https://doi.org/10.1145/3622854

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

Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2018a. RustBelt: Securing the Foundations of the Rust Programming Language. *Proceedings of the ACM on Programming Languages* 2, POPL (2018), 66:1–66:34. https://doi.org/10.1145/3158154

Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. 2018b. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming* 28 (2018), e20. https://doi.org/10.1017/S0956796818000151

Tomas Kalibera, Jeff Hagelberg, Filip Pizlo, Ales Plsek, Ben Titzer, and Jan Vitek. 2009. CDx: a family of real-time Java benchmarks. In *Proceedings of the 7th International Workshop on Java Technologies for Real-Time and Embedded Systems* (Madrid, Spain) *(JTRES '09)*. Association for Computing Machinery, New York, NY, USA, 41–50. https://doi.org/10.1145/1620405.1620412

J. Klensin. 2001. RFC2821: Simple Mail Transfer Protocol.

Naoki Kobayashi. 1997. A Partially Deadlock-Free Typed Process Calculus. In *12th Annual IEEE Symposium on Logic in Computer Science (LICS)*. IEEE Computer Society, 128–139. https://doi.org/10.1109/LICS.1997.614941

Kensuke Kojima and Atsushi Igarashi. 2011. Constructive linear-time temporal logic: Proof systems and Kripke semantics. *Information and Computation* 209, 12 (2011), 1491–1503. https://doi.org/10.1016/J.IC.2010.09.008

Wen Kokke, Fabrizio Montesi, and Marco Peressotti. 2019. Better Late Than Never: A Fully-Abstract Semantics for Classical Processes. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 24:1–24:29. https://doi.org/10.1145/3290337

Ron Koymans. 1990. Specifying Real-Time Properties with Metric Temporal Logic. *Real-Time Systems* 2, 4 (1990), 255–299. https://doi.org/10.1007/BF01995674

Pavel Krcál and Wang Yi. 2006. Communicating Timed Automata: The More Synchronous, the More Difficult to Verify. In *18th International Conference on Computer Aided Verification (CAV) (Lecture Notes in Computer Science, Vol. 4144)*. Springer, 249–262. https://doi.org/10.1007/11817963_24

Kai Lampka, Simon Perathoner, and Lothar Thiele. 2009. Analytic real-time analysis and timed automata: a hybrid method for analyzing embedded real-time systems. In *9th ACM & IEEE International conference on Embedded software (EMSOFT)*. ACM, 107–116. https://doi.org/10.1145/1629335.1629351

Sam Lindley and J. Garrett Morris. 2015. A Semantics for Propositions as Sessions. In *24th European Symposium on Programming (ESOP) (Lecture Notes in Computer Science, Vol. 9032)*. Springer, 560–584. https://doi.org/10.1007/978-3-662-46669-8_23

Sam Lindley and J. Garrett Morris. 2016. Talking Bananas: Structural Recursion for Session Types. In *21st ACM SIGPLAN International Conference on Functional Programming (ICFP)*. ACM, 434–447. https://doi.org/10.1145/2951913.2951921

Per Martin-Löf. 1982. Constructive Mathematics and Computer Programming. In *Logic, Methodology and Philosophy of Science VI*. Studies in Logic and the Foundations of Mathematics, Vol. 104. Elsevier, 153–175. https://doi.org/10.1016/S0049-237X(09)70189-2

Robin Milner. 1980. *A Calculus of Communicating Systems*. Lecture Notes in Computer Science, Vol. 92. Springer. https://doi.org/10.1007/3-540-10235-3

Robin Milner. 1999. *Communicating and Mobile Systems: the π-calculus*. Cambridge University Press.

Yasuhiko Minamide, Greg Morrisett, and Robert Harper. 1996. Typed Closure Conversion. In *23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM, 271–283. https://doi.org/10.1145/237721.237791

Rumyana Neykova, Laura Bocchi, and Nobuko Yoshida. 2017. Timed runtime monitoring for multiparty conversations. *Formal Aspects of Computing* 29, 5 (2017), 877–910. https://doi.org/10.1007/S00165-017-0420-8

Joël Ouaknine and James Worrell. 2005. On the Decidability of Metric Temporal Logic. In *20th IEEE Symposium on Logic in Computer Science (LICS)*. IEEE Computer Society, 188–197. https://doi.org/10.1109/LICS.2005.33

Daniel Patterson, Noble Mushtak, Andrew Wagner, and Amal Ahmed. 2022. Semantic Soundness for Language Interoperability. In *43rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 609–624. https://doi.org/10.1145/3519939.3523703

Jorge A. Pérez, Luís Caires, Frank Pfenning, and Bernardo Toninho. 2012. Linear Logical Relations for Session-Based Concurrency. In *21st European Symposium on Programming (ESOP) (Lecture Notes in Computer Science, Vol. 7211)*. Springer, 539–558. https://doi.org/10.1007/978-3-642-28869-2_27

Jorge A. Pérez, Luís Caires, Frank Pfenning, and Bernardo Toninho. 2014. Linear Logical Relations and Observational Equivalences for Session-Based Concurrency. *Information and Computation* 239 (2014), 254–302. https://doi.org/10.1016/j.ic.2014.08.001

Andrew M. Pitts and Ian Stark. 1998. Operational Reasoning for Functions with Local State. *Higher Order Operational Techniques in Semantics (HOOTS)* (1998), 227–273.

André Platzer. 2008. Differential Dynamic Logic for Hybrid Systems. *J. Autom. Reason.* 41, 2 (Aug. 2008), 143–189. https://doi.org/10.1007/s10817-008-9103-8

Gordon D. Plotkin. 1973. *Lambda-definability and logical relations*. Technical Report. University of Edinburgh.

Amir Pnueli. 1977. The Temporal Logic of Programs. In *18th Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE Computer Society, 46–57. https://doi.org/10.1109/SFCS.1977.32

Davide Sangiorgi and David Walker. 2001. *The π-calculus: a Theory of Mobile Processes*. Cambridge University Press.

NXP Semiconductors. 2021. *I2C-bus specification and user manual*. Standard. NXP Semiconductors.

SEP. 1999. Stanford Encyclopedia of Philosophy: Temporal Logic. https://plato.stanford.edu/entries/logic-temporal/.

Richard Statman. 1985. Logical Relations and the Typed λ-calculus. *Information and Control* 65, 2/3 (1985), 85–97. https://doi.org/10.1016/S0019-9958(85)80001-2

William W. Tait. 1967. Intensional Interpretations of Functionals of Finite Type I. *The Journal of Symbolic Logic* 32, 2 (1967), 198–212. http://www.jstor.org/stable/2271658

Amin Timany, Robbert Krebbers, Derek Dreyer, and Lars Birkedal. 2024. A Logical Approach to Type Soundness. *Journal of the ACM (JACM)* (2024). To appear.

Bernardo Toninho. 2015. *A Logical Foundation for Session-Based Concurrent Computation*. Ph. D. Dissertation. Carnegie Mellon University and New University of Lisbon.

Bernardo Toninho, Luís Caires, and Frank Pfenning. 2013. Higher-Order Processes, Functions, and Sessions: A Monadic Integration. In *22nd European Symposium on Programming (ESOP) (Lecture Notes in Computer Science, Vol. 7792)*. Springer, 350–369. https://doi.org/10.1007/978-3-642-37036-6_20

Bas van den Heuvel, Farzaneh Derakhshan, and Stephanie Balzer. 2024. Information Flow Control in Cyclic Process Networks. In *38th European Conference on Object-Oriented Programming (ECOOP) (LIPIcs, Vol. 313)*. Schloss Dagstuhl -

Leibniz-Zentrum für Informatik, 40:1–40:30. https://doi.org/10.4230/LIPICS.ECOOP.2024.40

Philip Wadler. 2012. Propositions as Sessions. In *ACM SIGPLAN International Conference on Functional Programming (ICFP)*. ACM, 273–286. https://doi.org/10.1145/2364527.2364568

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

Yue Yao, Grant Iraci, Cheng-En Chuang, Stephanie Balzer, and Lukasz Ziarek. 2024a. *Semantic Logical Relations for Timed Message- Passing Protocols (Artifact)*. https://doi.org/10.5281/zenodo.13937290

Yue Yao, Grant Iraci, Cheng-En Chuang, Stephanie Balzer, and Lukasz Ziarek. 2024b. Semantic Logical Relations for Timed Message-Passing Protocols (Extended Version). arXiv:2411.07215 [cs.PL] https://arxiv.org/abs/2411.07215

Received 2024-07-11; accepted 2024-11-07