Semantics for Tydi

The development of a formal foundation for the Tydi hardware stream specification.

Harold Struik

Delft University of Technology

TU Delft

Semantics for Tydi

The development of a formal foundation for the Tydi hardware stream specification.

by

Harold Struik

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Friday the 30th of January.

Student number:	5651573	
Project duration:	1 January, 2025 – 30 January, 2026	
Thesis committee:	Prof. Peter Hofstee,	TU Delft, IBM, supervisor
	Prof. Soham Chakraborty	TU Delft
	Zaid Al-Ars,	External

This thesis is confidential and cannot be made public until February, 2026.

Cover: CERN computing centre in 2017. Copyright: Robert Hradil, Monika Majer/ProStudio22.ch 2017-04-25

TU Delft

Summary

The Tydi (Typed Dataflow Interface) specification was introduced to bridge a gap in the High Level Synthesis (HLS) domain, enabling engineers to easily define component interfaces, mapping complex, dynamically sized data structures onto hardware streams. While Tydi successfully defines the static bit-level layout, it lacks a formal description for the dynamic procedure of mapping elements in a stream onto hardware lanes. This ambiguity has hindered the development of Tydi related tools and its integration in HLS frameworks. This work introduces a layered formalism for Tydi, centred around a modular operational semantics. It defines a two-tiered type system that separates a flexible user-facing syntax, from a canonical, hardware-oriented representation, strengthening the link between a type's composition and runtime behaviour. The dynamic mapping is implemented by a configurable set of small-step semantics rules. These are derived from a user provided set of formal properties, replacing the specifications monolithic complexity levels with fine-grained control over interface logic. We define key structural determinism and progress metrics for the formalism, enabling the verification of interface implementations for streams of arbitrary type. This foundation enables developers to define verifiably correct, typed hardware stream interfaces as easily as software, while retaining control over critical engineering trade-offs. A simulator of the semantics has been implemented, and provides empirical verification of the formalism and an insightful overview of the parsing logic [48].

Structure of the Thesis

This thesis is organized into seven chapters that progressively develop the formal foundation for the Tydi specification.

Chapter 1: Introduction and Background. This chapter introduces the data streaming paradigm and the specific challenges of designing stream interfaces in hardware. It presents the core concepts of the existing Tydi specification, including the distinction between logical and physical streams and the concept of "streamspace" used to map data over time.

Chapter 2: Analysis: Towards a Formal Foundation. An analysis of the status quo identifies a "specification gap" regarding the runtime behaviour of Tydi interfaces, placing the focus on the loss of hierarchy information between physical streams. The chapter evaluates existing formalisms (e.g. dependent type theory, operational semantics) and proposes a tiered formal framework. This framework separates the interface specification from its implementation, and the static and dynamic domains of the formalism.

Chapter 3: The Formalism. This chapter defines the core mathematical components of the proposed solution. It introduces: 1) The Normalisation Function ($[\![\cdot]\!]$) to reduces the user exposed top-level type to a normalized type in Canonical Form, removing semantically ambiguous type compositions. 2) The Streaming Context (Γ), is a stateful device encapsulating buffers and the parse tree. 3) Interface Properties and Semantics, which use small-step operational rules to define the mapping procedure, interacting with the streaming context. 4) The Implementation Complexity Metric (ICM), a method to quantify the logic cost of specific interface configurations.

Chapter 4: Examples and Empirical Validation. The functionality of the formalism is demonstrated through a full parsing example. The chapter also details the *TinyTydi* simulator, a Python-based tool implemented to provide empirical verification of the normalisation and operational semantics.

Chapter 5: Verification of the Formalism. This chapter provides formal proofs to verify the correctness of the framework. It proves that the normalisation function always produces Canonical Form, that parse trees terminate properly, and that the core ruleset adheres to Structural Determinism and Progress (finite parsing).

Chapter 6: The Union Type: Handling Variant Data. The formalism is extended to include the `Union` type, addressing the complexity of variant data in hardware. It introduces a "deferred parsing commit" mechanism to maintain determinism and synchronicity between tag and data streams.

Chapter 7: Conclusion and Discussion. The final chapter summarizes the research contributions. It discusses the implications for the Tydi specification, such as the potential for automated interface generation and verification, and outlines future work.

Contents

1

Introduction and Background

This chapter will introduce the data streaming paradigm, the challenges it presents for hardware development and how Tydi aims to improves this. The last section presents an analysis of the shortcomings of the specification, laying the foundation for the core contributions of this work.

1.1. The data streaming paradigm

Early work on information systems began to recognize that data was no longer delivered in neat, static batches, but arrived continuously, unstructured, and in volumes too large for manual sorting. Figure 1.1 shows a quaint illustration from a 1992 Xerox paper, introducing Tapestry, a system for managing streams of documents in the advent of ”electronic mail”. In the same year, these researchers at Xerox built upon their experience of Tapestry to develop the notion of continuous queries. Such queries remain active over an append-only stream, only reporting results whenever new data satisfies a condition, effectively treating query evaluation as an ongoing process rather than a discrete event [49]. It introduces a rudimentary model of computation with streams of messages sent across interfaces, and providing semantics for composing and reasoning about such flows. These early systems mark the beginning of the data streaming paradigm, where computation is designed around the realities of unbounded, evolving information flows, rather than static evaluation.

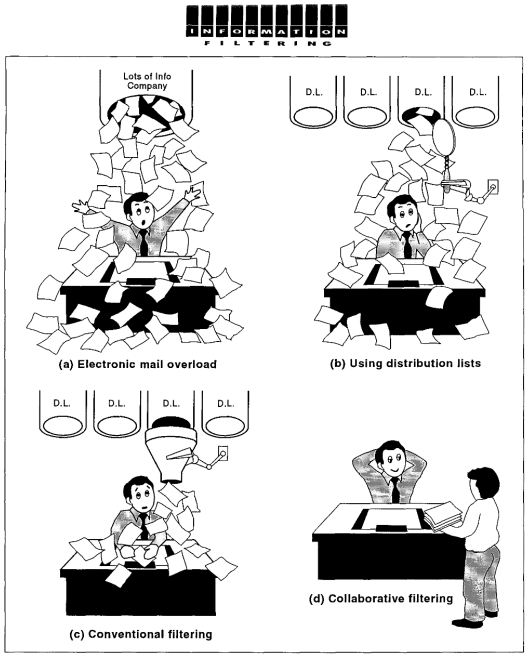


Figure 1.1: From the 1992 paper: A mail operator, overwhelmed by piles of unstructured messages, depicting the practical pressures that motivated automated and continuous filtering rather than periodic, one-shot processing [23].

As detailed in a recent comprehensive survey by Fragkoulis et al. [19], the evolution of the field can be categorised into three distinct generations, each addressing the growing complexity of data velocity and volume.

First Generation (2000–2010): The Era of DSMS.

Although the notion of continuous queries was introduced by Tapestry in 1992, much of the foundational formalisation and system design for stream processing occurred in the late 1990s and early

2000s, where the field diverged from standard database management into Data Stream Management Systems (DSMS). Researchers recognized that the ”one-size-fits-all” approach of traditional relational databases could not meet the low-latency requirements of real-time applications [47]. Systems such as *Aurora*, *Borealis*, and *TelegraphCQ* formalised the requirements for stream processing, establishing that systems must keep data moving and allow for ”windowing”, the ability to perform computations over finite slices of an infinite stream [1, 11]. While these systems successfully introduced sliding-window aggregation and continuous SQL-like queries, they targeted primarily on architectures that relied on increasing machine performance instead of parallelism, and maintained state in abstract, internal structures known as ”synopses” [19].

Second Generation (2011–2019): Scale-Out and Dataflow. The paradigm matured significantly with the advent of the ”Big Data” era and the introduction of the Dataflow model. Influenced by the scalability of MapReduce [17], focus shifted to distributed architectures. This modern approach decoupled the execution engine from the programming model, allowing developers to focus on *what* is being computed rather than *how* the stream is physically managed. The work by Akidau et al. clarified the critical distinction between *event time* (when an event occurred) and *processing time* (when the system observed it), providing a framework for balancing correctness, latency, and cost [3]. These systems (e.g., Apache Flink, Spark Streaming, Google Cloud Dataflow) introduced sophisticated state management, allowing for large, partitioned, and persistent user-defined state [10, 54].

Third Generation (2020–Present): Edge and Hardware Acceleration. We are currently witnessing a shift toward the third generation, characterized by the decentralisation of pipelines to the Edge and the use of specialized hardware. As noted by Zhang et al. and emphasized in recent surveys, the need for ultra-low latency is driving the adoption of hardware accelerators, such as FPGAs and GPUs, directly into the streaming pipeline [55, 19]. Projects like *Fleet* [50] demonstrate the potential of processing streams directly on FPGAs. While the software domain enjoys elegant, high-level frameworks [53, 20], the hardware domain is still lacking in standardized, type-safe interfaces for describing these complex data flows [22, 8, 13]. Recent formal work in the software domain has also addressed the potential of hardware oriented adaptations of stream processing formalisms [28, 31]. This sets the stage for the introduction of Tydi (Typed Dataflow Interface), an open hardware stream specification [40]. It expresses how complex, dynamically sized data structures can be exchanged between components using streaming interfaces, based on an intuitive, hardware-oriented type system.

1.2. Designing stream interfaces in hardware

Before we dive into Tydi, it is important to substantiate *why* working with streams is particularly difficult in the hardware domain. Designing hardware is fundamentally about managing spatial resources and enforcing timing constraints. Unlike software, where memory and dynamic containers can hide layout and temporal ordering, hardware requires explicit decisions about how many wires exist, their widths, where data is stored, and exactly when it is valid. In the domain of dataflow programming, channels that connect components are usually implemented as *streams*: FIFO, point-to-point links where transfers flow in order. Components that consume and produce streams, “streamlets“, therefore need interfaces that describe both the shape of the data and the timing of its elements. A transfer is a set of data elements, sent simultaneously over a single physical stream. Complex, dynamically sized data, (e.g. a chat-message comprised of a single timestamp and a variable length sequence of characters), cannot be exposed as a single fixed-width port, so the data must be serialized. This serialisation requires designers to make various non-trivial design considerations: how to break the structure into smaller chunks of data; *primitives*, how to indicate sequence boundaries, how to drive the validity signals, and how to propagate back-pressure when downstream logic stalls. These tasks are repetitive and error-prone.

This difficulty is illustrated by the Fletcher framework, which predates the Tydi specification and directly motivated its development [37, 38]. Fletcher demonstrates that, even when a high-level software schema (Apache Arrow) precisely describes complex, nested data structures, translating this description into efficient hardware stream interfaces requires non-trivial interface logic. In Fletcher, a rudimentary form of Tydi is used to automatically generate interfaces that decompose complex data types into multiple coordinated streams, handling serialisation, boundary signalling, buffering, and back-pressure explicitly. While this approach proved that such interfaces can be generated automatically, it also illus-

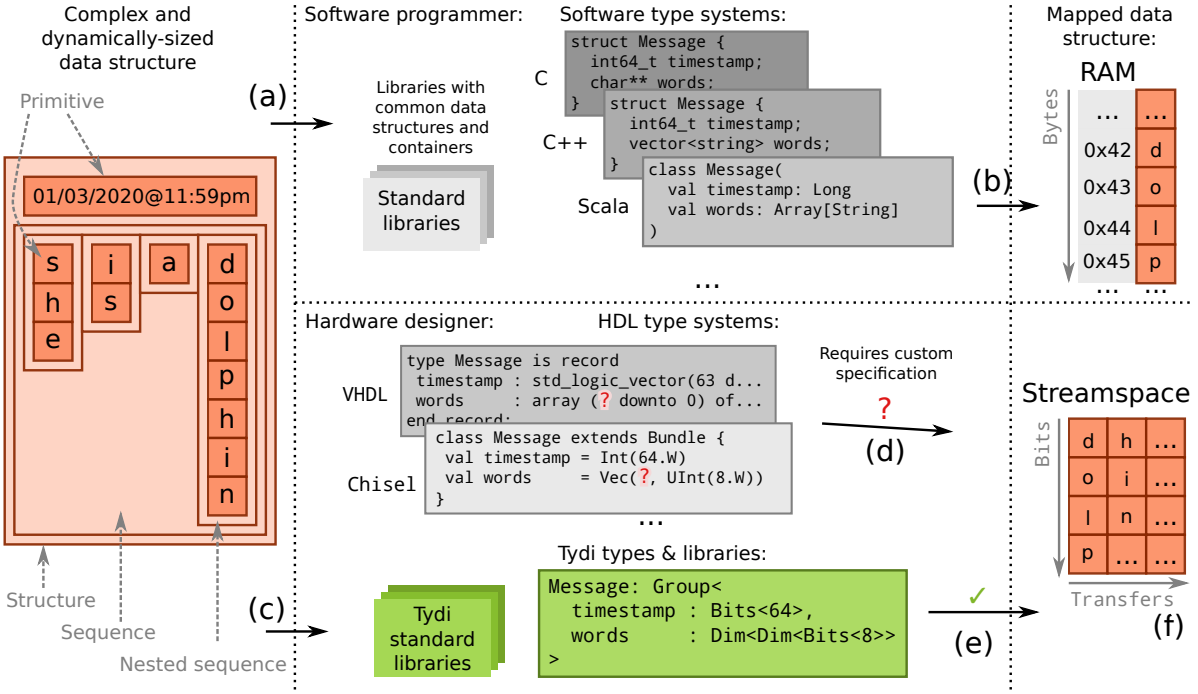


Figure 1.2: *"To implement a data structure, programmers choose some types and containers, helped by language constructs and libraries. (a) Runtime engines and compilers take care of the mapping to RAM. (b) The same for contemporary HDLs (c) is prevented because dynamically sized structures are not inherently supported. When mapping to hardware streams (d) designers customize solutions to transport data structures over multiple stream transfers. Tydi is a specification that clearly and intuitively provides a mapping (e) and predefined containers for common types"* [40].

trates how relatively trivial software data structures, already require complex implementations.

To implement stream interfaces, designers can opt for low-level bit protocols or resort to designing their own protocols, each requiring explicit boundary markers, parsing logic, and buffering. HLS tools handle simple aggregates but typically generate interfaces for primitive types only. Expressing more complex structures would still require manual control logic and custom conventions. As a result, interfaces accumulate implicit assumptions about packing, alignment, ordering, and throughput settings, limiting reuse and making composition difficult. Additional engineering considerations further complicate the implementation of these interfaces. For example, throughput scaling, alignment rules, per-lane strobes, and control-signal complexity; all of these interact with area and timing. Additionally, a wider transfer may reduce cycle count but increases wires; fine-grained strobes ease packing but will require more control logic, etc. These trade-offs may be revisited for every interface in a design. Because of this, hardware stream-processing systems are costly to design and maintain. Interfaces are bespoke, converters proliferate, and correctness relies on matching many low-level conventions. A standardised, type-oriented description of both the spatial layout and temporal behaviour, combined with explicit engineering parameters such as the number of lanes, can greatly reduce this burden. The Tydi specification intends to provide exactly this: a mapping from data types to well-defined stream interfaces that can be implemented consistently, and reused across components.

1.3. The Tydi specification

Due to the elaborate process needed to design an interface for a component that interacts with streams, Tydi was introduced to significantly improve this procedure, as an open, hardware-oriented specification for streaming complex, nested data types. Given a high-level logical type and a set of engineering parameters, Tydi prescribes both the spatial layout of signals on a component interface and the temporal mapping of data onto transfers over that interface. The original work illustrates its value by comparing the software and hardware perspectives of mapping a chat message to either memory, or a hardware stream, shown in figure 1.2. This chat message will be a recurring example throughout this work, comparing implementations and illustrating the impact of mapping procedures.

1.3.1. Analogy: The Universal Lego Machine

To build intuition for the challenges Tydi addresses, consider a Lego assembly machine as shown in figure 1.3. Suppose the machine is designed to build towers. Each tower always begins with a single foundation plate, followed by an arbitrary number of identical tower segments. The instruction booklet describes this structure, but the number of segments is not known in advance. To feed this machine, the interface must expose holes shaped like the required pieces: one hole for the foundation plate and one hole for tower segments. Unlike a standard Lego set where all pieces are available at the start, here the tower may be of any height, making it impossible to pre-allocate storage for ”all” pieces; The machine must start building as soon as the first pieces arrive.

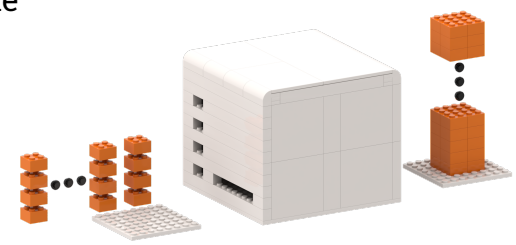


Figure 1.3: The lego tower construction box, operating on streams of tower segments, building towers of arbitrary height.

This illustrates a critical *temporal* constraint: the order matters. The foundation plate must arrive first; only after it is accepted can the machine consume segments. The interface must therefore communicate not only *what* piece is provided, but also *when* it is valid, where the boundaries between towers lie, and when a tower is complete. In hardware, complex, nested, and variably sized data structures cannot be represented as a single fixed-width port. They must be serialized into elements arriving over time, with explicit signals marking structure and validity. The Tydi specification provides the formal ”instruction booklet” to describe these shapes and when they are needed. If every layer of the tower requires 4 pieces, it is possible to expand the interface to 4 holes, however, if the component providing the pieces can only provide 2 pieces at a time, this would be a waste of resources. This illustrates how performance and resource requirements can further impact designing the interface.

For a normal Lego build, the steps of the instruction booklet describe, when which pieces are needed. For the Tydi example, you might need to stay on the same page of the Lego booklet, repeating the same steps until the supply of some types of pieces are used up, and then continuing with the rest of the booklet. This is however where the Lego analogy falls short, since stream interfaces may have much more elaborate conditions, requiring ”skipping through the instruction booklet” in arbitrarily complex ways. Still, it highlights that interacting with streams in hardware is not an obvious procedure, especially when it needs to be defined using finite, statically known resources.

1.3.2. The Core Concepts of Tydi

Similar to how Apache Arrow specifies data layouts in memory, Tydi specifies data layouts ”in-flight” [6]. It defines a two-layer interface model that separates abstract stream descriptions from their concrete hardware realisation. At the top layer, a *logical stream* specifies the structure and behaviour of data exchanged between components, expressed using a small, hardware-oriented type system. At the lower layer, one or more *physical streams* implement this specification as concrete bundles of wires, complete with handshake, data, and control signals. This separation allows designers to reason about complex, dynamically sized data structures at a high level, while delegating bit-level layout and protocol details to a synthesis algorithm. Conceptually, the logical stream defines *what* data is communicated and how it is structured, while the physical stream defines *how* this contract is realised in hardware. Tydi defines “streamspace“ as the 2d plane represented by the physical dimension of parallel lanes or wires, and the temporal dimension of transfers. An example of streamspace mappings is shown in figure 1.4.

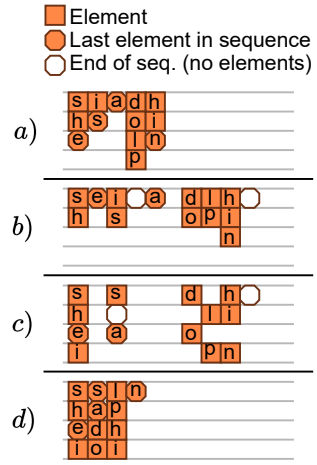


Figure 1.4: Comparing how 2D char seq. $\{\{she\},\{is\},\{a\},\{dolphin\}\}$ can be mapped to streamspace, where $a)$ terminates inner sequences, $b)$ allows delayed termination signals, $c)$ no inner seq. termination + delayed signalling, and $d)$ no inner seq. termination and enforced packing.

Logical streams. Logical streams are defined using the Tydi type system, which distinguishes between data layout and temporal behaviour. Data layout is described using a small set of types: `Null`,

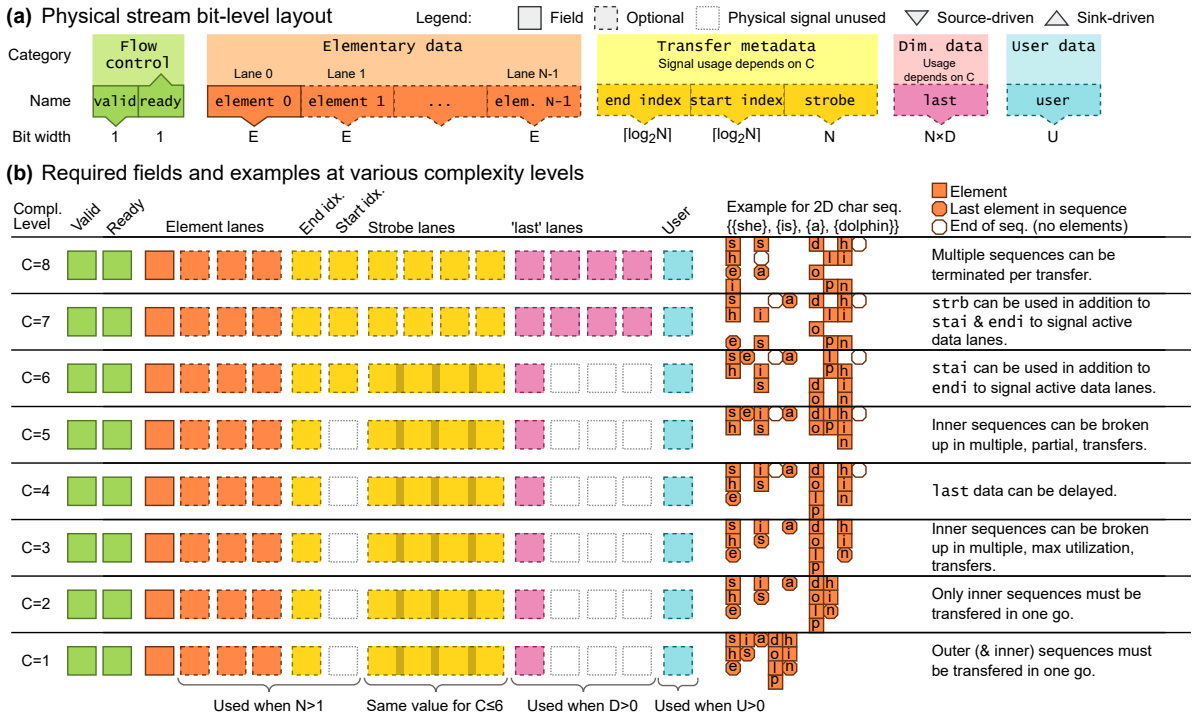


Figure 1.5: This figure shows (a) the layout of lanes for signals and data carrying wires, and (b) how each complexity level impacts the potential transmission of a 2 dimensional char array [15].

representing a singleton value; `Bits`(b), denoting a bit vector of width b; `Group`, a product type containing named fields; and `Union`, a sum type representing a tagged variant. These element types describe how data is structured, but are agnostic to how it is transferred over time. Temporal structure is introduced by the logical `Stream` type of the form:

$$\texttt{Stream}(T_e, t, d, s, c, r, T_u, x)$$

It represents a sequence of elements of type T_e, together with a number of engineering parameters that determine its runtime behaviour. The *throughput* parameter t specifies the minimum number of elements transferred per handshake, and determines the degree of parallelism exposed at the interface. The *dimensionality* d captures the nesting depth of sequences, governing how many independent sequence boundaries must be represented. The *complexity* parameter c encodes ordering and packing guarantees made by the source, trading implementation simplicity against flexibility. Lower complexity levels impose stricter constraints on transfer patterns, while higher levels permit more relaxed scheduling. Synchronicity represents the relation between the dimensionality information of the parent stream and the child stream The user defined data can be used to further customise the contract between source and sink. Figure 1.5 shows how a 2 dimensional char array can be transmitted for various complexity levels. Through recursive composition of `Stream`, `Group`, and `Union` types, logical streams can express arbitrarily complex, nested, and dynamically sized data structures, including inter-stream relations. The chat message example in figure 1.2 is typed using the `Dim` shorthand notation of a specific logical stream node, which adds an additional dimension to its subtype. This gives $\texttt{Group}\langle\texttt{Bits}\langle 64\rangle, \texttt{Dim}\langle\texttt{Dim}\langle\texttt{Bits}\langle 8\rangle\rangle\rangle\rangle$, defining a group of a 64-bit primitive, together with a 2 dimensional sequence of 8 bit primitives, representing the timestamp and chat message respectively.

Physical streams. A physical stream is the concrete hardware realisation of a logical stream, defined:

$$\texttt{PhysicalStream}(E, N, D, C, U)$$

Here E denotes the element payload fields, N the number of parallel element lanes, D the dimensionality, C the complexity level, and U the user-defined transfer fields. A physical stream is implemented as a bundle of signals, all synchronous to a shared clock.

At its core, each physical stream includes a `valid`/`ready` handshake pair, which governs flow control between source and sink. The data payload is carried on the `data` signal, consisting of N lanes, each encoding one element of type E. Sequence boundaries are communicated using the `last` signal, which contains D termination bits per lane, one for each level of nesting. Additional signals such as `stai`, `endi`, and `strb` provide fine-grained control over which lanes are active in a given transfer, enabling partial transfers and relaxed packing when permitted by the selected complexity level. An optional `user` signal carries auxiliary control or metadata alongside the data stream.

Not all signals are required in every configuration. Their presence and interpretation depend systematically on the parameters N, D, C, and U. In particular, increasing the complexity level enables progressively more flexible lane utilisation and sequence termination behaviour. Figure 1.5 provides an overview of which signals are used at each complexity level and how their streamspace representation evolves.

Streamspace. Underlying both logical and physical streams is the notion of *streamspace*. Rather than mapping data structures solely onto space (bits), Tydi models communication in a two-dimensional plane spanned by spatial resources (element bits and lanes) and temporal resources (stream transfers). Complex and dynamically sized data structures are therefore mapped not onto bit-vectors, but across multiple transfers over time. In this view, the logical stream defines the allowed complete streamspace representations of the data, where physical streams occupy specific regions of streamspace, with the relationships between regions on the plane delegated to the logical stream's responsibility. Streamspace provides the conceptual foundation for reasoning about nested sequences, parallelism, and ordering guarantees.

Streamspace Types and the Container Library. To facilitate concise and reusable descriptions of common data structures, Tydi introduces a set of streamspace types shorthand notation, and a standard container library. At the lowest level, shorthand notations such as `DIM`, `REV`, and `NEW` provide a more concise way to interact with the logical stream type.

- `DIM` introduces an additional dimension to its subtype:

$$\texttt{Dim}(T_e, t, c, T_u) \mapsto \texttt{Stream}(T_e, t, 1, Sync, c, Forward, T_u, false)$$

- `REV` creates a physical stream that flows in the opposite direction of its parent, enabling request-response style interfaces:

$$\texttt{REV}(T_e, t, c, T_u) \mapsto \texttt{Stream}(T_e, t, 0, Sync, c, Reverse, T_u, false)$$

- `NEW` introduces an additional physical stream at the same dimensional level, allowing independent streams to coexist without implicit ordering:

$$\texttt{NEW}(T_e, t, c, T_u) \mapsto \texttt{Stream}(T_e, t, 0, Sync, c, Forward, T_u, false)$$

Building on these, the specification defines a library of container types that act as canonical mappings of familiar data structures, such as lists, vectors, structs, and variants, into streamspace. These containers are aliases for well-defined compositions of the underlying types.

From Logical to Physical: Synthesis. The connection between logical and physical streams is established by a synthesis algorithm defined in the specification. This algorithm traverses the logical stream type, extracts element and user fields, and computes the corresponding physical stream parameters. For each logical `Stream` node, the element type is flattened into a linear bit-level representation, the throughput parameter determines the number of lanes N, and the dimensionality and complexity parameters are carried over directly. The result is a canonical `PhysicalStream` description that prescribes the exact signal set and widths required for the interface.

This synthesis process is deterministic: any two tools or designers that start from the same logical stream type and parameters will derive bit-level compatible physical interfaces. By construction, this *should* eliminate the need for bespoke glue logic between independently developed components that adhere to the same Tydi specification. Additionally, when two interfaces are of the same logical stream type, but their properties differ, Tydi can assist their composition by synthesising the required glue logic.

Putting it all together. The core concepts of Tydi form an abstraction stack. Logical streams provide a precise, compositional description of complex data structures and their temporal behaviour. Physical streams realize these descriptions as concrete, parametrized hardware interfaces. The synthesis algorithm bridges the two layers, ensuring that high-level specification is properly lowered to its implementation. This separation of concerns is central to Tydi's goal: enabling reusable, type-safe, and verifiable stream interfaces, while preserving explicit control over performance and implementation complexity.

2

Analysis: Towards a Formal Foundation for Tydi

2.1. Status quo and problem statement

Tydi successfully specifies how complex, dynamically sized data structures can be exchanged, by defining streaming interfaces based on an intuitive, hardware-oriented type system. However, there is a notable omission from the original specification, namely *how* a hardware interface must be implemented in order to adhere to the specification. To a certain extent this is to be expected, as such open hardware specifications are usually only prescriptive regarding the static layout of signals. However, Tydi also describes the temporal layout of data using its notion of streamspace. When two distinct parties develop interfaces in accordance with the specification, their implementations should be compatible for a matching type and complexity level. Any ambiguities in the specification inevitably result in occurrences where this is not the case.

The core problem arises from the lack of a formal description of the runtime mapping procedure. This omission has already hindered the development of Tydi-related tools and hardware utilities. Notably, the original authors have expressed that they attempted a more formal approach during the Fletcher development era, but abandoned it due to time constraints and complexity. The angle bracket notation for the type system is a quirk that originated from this attempt at a formalism.

Through further conversations with the original authors, I have identified that Tydi can be viewed as having three distinct layers: a high-level data structure specification, a low-level physical stream of wires, and a ”mid-level” protocol layer. While the top and bottom layers are well-defined, their connection has been considered as a “gap“ in the specification. More specifically, there are uncertainties regarding the transmission of data in nested streams, and how this is impacted by their inter-stream dependencies. The authors describe the current Tydi specification as a ”factory” for interfaces: it is generic and flexible, but it lacks the prescriptive power to define exactly what a functioning hardware interface *should do* in complex scenarios, such as nested streams with interdependencies.

This indicates that the ”hardware-oriented” type system does not provide intuitive assertions regarding how elements are actually transmitted. For example, when the existence of a `Group` type does not provide strong guarantees regarding how its fields are interleaved or concatenated over physical streams, any formalism interacting with the type must account for the larger structure by also evaluating the fields of the group. Allowing distinct types to correspond to equal streamspace mappings, (and thus technically compatible interfaces), is undesirable for verification.

The flexibility is a double-edged sword: while it enables standards such as JSON to be mapped to Tydi with ease, but it complicates the definition of a formal process for implementation; connecting the top to the bottom. The type of a Tydi stream remains only loosely coupled to its actual runtime behaviour, undercutting the effectiveness of its hardware-oriented nature. Therefore, a core challenge is to formally describe this ”mid-level” runtime procedure such that it algorithmically determines an

interface implementation from a flexible Tydi type and its engineering parameters, providing a provable bridge between high-level data structures and low-level physical streams. This can be seen as the semantic domain of the interface definition: How can we infer how data is transmitted, from the structure of a stream's type?

2.2. Requirements for a formal foundation

Tydi's core ambition is to allow hardware designers to describe complex, nested, and dynamically sized data structures at a high level, while retaining precise control over how these structures are transmitted over streaming interfaces. The absence of a description of the runtime procedure, mapping the typed data to streamspace, is an awkward omission. While the Tydi type determines *what* data is communicated, and using which signals, it does not effectively describe *how* this communication is actually achieved.

In this context, a quote comes to mind, attributed to Mark Manasse in the Types and Programming Languages book by Benjamin C. Pierce: *"The fundamental problem addressed by a type theory is to ensure that programs have meaning. The fundamental problem caused by a type theory is that meaningful programs may not have meanings ascribed to them. The quest for richer type systems results from this tension."* [41] It highlights how the intention to give meaning to interfaces through static structure alone, the Tydi "hardware-oriented type system", inevitably leaves some aspects of the runtime behaviour underspecified. The Tydi type system successfully defines what may be communicated, but falls short of giving explicit meaning to how this communication actually unfolds over time. The resulting ambiguity mirrors Manasse's tension, motivating the need for a richer formal foundation that connects interface types to a precise, incremental runtime semantics. To close this gap, the additional parameters such as synchronicity and complexity are introduced, alongside the type in the Logical Stream specification. How these relate physical streams after synthesis is currently phrased too informally, which has contributed to the existing ambiguities.

From an engineering perspective this has concrete consequences. Two independently developed interfaces that nominally adhere to the same Tydi type may still be behaviourally incompatible, since the specification leaves room for interpretation in the temporal packing of elements, especially when dealing with nested streams. To solve this, I set out to define a formal foundation for Tydi that establishes a precise, algorithmic way to derive the runtime behaviour of an interface from its type and configuration parameters alone. This is a prerequisite for reasoning about compatibility of senders and receivers, and for enabling verification of interface implementations.

At the same time, the formalism must respect the realities of hardware design. All stateful resources implied by this formal foundation, such as buffers, counters, or indices, must be statically bounded and known at design time. Approaches that rely on unbounded queues, memory, or make use of abstract notions of time, are fundamentally incompatible with hardware synthesis. The formal description must therefore remain hardware-aware, ensuring that every construct corresponds to implementable logic with well-defined resource requirements. More specifically, such a formalism must operate *incrementally*. Streaming interfaces process data element-by-element, potentially with arbitrary delays between transfers, and the formalism must mirror this behaviour. The runtime procedure should therefore be defined as an incremental mapping that consumes and emits data stepwise, rather than as a monolithic transformation over entire data instances. When dealing with streams, we cannot cache the 'entire' chat message, since the message has an arbitrary size.

Finally, an important requirement is accessibility. Tydi is explicitly intended as a practical tool for hardware engineers, and shouldn't require an exercise in advanced formal methods. While the formalism must be sufficiently rigorous to support proofs of correctness and determinism, it should avoid imposing heavyweight mathematical machinery on the end user. Ideally, the user-facing model remains intuitive and type-driven, with the formal rigour largely hidden away in the underlying formalism.

To conclude, initially I will focus on identifying the ambiguities and omissions, by constructing a formalism for a subset of the specification, while maintaining the ability to expand this formalism later. The process of determining which portions of the specification to include is an important design consideration in itself, potentially shedding light on how certain ambiguities arose in the first place.

2.3. Evaluation of existing formalisms

A natural starting point for formalising Tydi is the body of work on expressive type systems. In the context of stream interfaces, dependent, session, and refinement types are good candidates for further investigation.

Dependent types allow their definition to depend on their assigned value. In principle, dependent type theory can relate values and types closely enough to encode the detailed invariants governing the hierarchy of nested streams [36]. It can make structural properties (e.g. sequence lengths, alignment relationships, or numeric relations between nested dimensions) part of the type itself, so these properties are checked by the type system rather than expressed as separate interface properties. Recent applications to hardware include the modelling of parametrised DSP circuits in Idris, which captures relationships between data values and bit-level structures like wordlengths or register pruning [43]. Other approaches have utilized dependent types to reason about the physical structure of hardware interfaces and ensure they adhere to standards like AXI [33]. While these systems effectively describe the *static* layout and structural correctness of an interface, they do not provide an intuitive way for a hardware designer to express the temporal unfolding of a stream. As a result, dependent type systems remain a powerful verification tool but are not a good fit for Tydi's goal of remaining an approachable design entry point.

Session types and behavioural type systems excel at verifying communication protocols and ensuring interface compatibility through duality [24]. They have been successfully applied to FPGA-based designs to guarantee communication-safety and deadlock-freedom in reconfigurable parallel kernels [34]. Furthermore, they induce a Labelled Transition System (LTS), mapping naturally to the Finite State Machine (FSM) logic for hardware implementations. However, a fundamental mismatch exists in semantic granularity: session transitions are atomic, whereas Tydi transfers are incremental. While a session type can describe which element comes next in a sequence, it lacks the ability to describe "how much" of that element is transmitted in a specific instance of time. It reasons about transmitting the entire chat message but does not provide sufficient control over the transmission of individual characters across physical lanes. Consequently, session types fail to account for the spatial lane-packing and complexity-level constraints inherent to Tydi's concept of streamspace mapping.

Refinement types offer a middle ground by augmenting base types with logical predicates [21]. Attaching predicates to base types (e.g. Bits(64){t | t ≤ MAX_TS}), is convenient for checking value-level invariants such as valid ranges or non-empty sequences. In a Tydi context, they could formalise critical data invariants, such as ensuring a `Bits(64)` timestamp represents a valid POSIX range. However, refinement types share the same limitation as the aforementioned systems regarding temporal execution: they are essentially static. They can describe the *validity* of a data instance once it is fully assembled but offer no mechanism to govern the incremental, state-dependent logic required to pack partial elements into physical hardware lanes across multiple cycles.

More recently, **specialized type systems** have emerged to bridge the gap between abstract data and hardware timing. *Space-Time Types*, as implemented in Aetherling, use type-directed scheduling to map data-parallel programs onto hardware, with specific throughput requirements [18]. Similarly, *Timeline Types* in Filament provide a modular way to specify and enforce cycle-accurate timing and structural constraints for hardware modules [35]. However, these formalisms are not directly applicable to the Tydi problem due to their fundamental reliance on *static* scheduling. By 'static scheduling' I mean the compile-time, fixed assignment of computation and data transfers to specific cycles (typical of Aetherling), in contrast with the run-time, handshake-driven flow control found in latency-insensitive designs such as Tydi.

Aetherling explicitly targets "statically scheduled, streaming hardware" and operates on homogeneous, fixed-length sequences whose lengths must be known at compile time [18]. It is therefore optimized for designs that intentionally avoid the control-logic overhead of dynamic flow control. Filament likewise focuses on statically timed pipelines and explicitly contrasts its approach with latency-insensitive interfaces (i.e., valid/ready handshaking), which it identifies as inefficient for its target domains [35].

In contrast, Tydi is designed to support dynamically sized data structures and relies on latency-insensitive handshaking to manage dataflow over nested streams. Because it does not bake a cycle schedule into the type system, scheduling decisions emerge at run time from the interface properties and the actual

composition of data instance. Where Aetherling and Filament resolve timing into rigid, cycle-accurate constraints, Tydi deliberately leaves multiple valid physical mappings for the same logical structure. Consequently, these systems are well suited to regular, statically timed data-parallel kernels, but are too specialized to capture the flexible, state-dependent, and irregular mapping required for Tydi.

Since the **software streaming paradigm** is more mature, formal work in this domain could provide effective candidates for a Tydi formalism. Frameworks such as Kahn Process Networks and coalgebraic stream semantics provide elegant mathematical descriptions of streaming computation [27, 45, 16]. Interfaces can be reasoned about as specific category of stream transformers themselves, therefore these formalisms could describe their runtime procedures effectively. However, models from the software domain typically assume arbitrary buffers, not providing sufficient granularity regarding execution timing, which clashes with the finite resources and discrete timing inherent to hardware interfaces. Adapting such models to the hardware domain, which has been expressed as a valuable future work in recent studies [28], is far beyond the scope of this work.

Interface automata and contract-based models offer ways to reason about compatibility through assumptions and guarantees [4, 7]. While useful for higher level protocol reasoning, they struggle to scale to rich, nested data structures without incurring state-space explosion. Furthermore, algorithmically determining an automaton for arbitrary type and interface properties would be a significant challenge, certainly requiring additional formal frameworks. The concepts regarding defining interface contracts through automata are an insightful way to reason about Tydi implementations. I deem it feasible to extend these to a higher level of abstraction, which will require a mapping to the existing Tydi type system.

In essence, we are trying to determine the “meaning” of a Tydi type, where its meaning is concretely the runtime behaviour. This corresponds with the domain of formal semantics of programming languages, describing how the syntax of a system determines its execution. Two main styles of semantics are commonly distinguished: denotational and operational.

Denotational semantics gives each syntactic construct mathematical “meaning“, such as a set of behaviours or execution traces, where the meaning of a larger construct is built systematically from the meanings of its parts. For Tydi this view is useful: a type (plus interface properties) can be read as denoting the set of admissible streamspace traces or transfer sequences, which supports high-level equivalence reasoning and a compact specification of “what” behaviours are allowed. To a certain extent, this is what Tydi currently is, describing the complete streamspace mapping of a given Tydi type in one (ambiguous) step. The denotational style abstracts away the step-by-step mechanism that produces a trace and therefore lacks the temporal granularity required to reason iteratively about element-by-element packing, termination placement, or handshake-driven flow control.

Operational semantics instead describes execution as more granular state transitions. Small-step, rule-based operational semantics are naturally suited to cycle-accurate, hardware-aware descriptions: a configuration can include the current parse configuration, and each rule corresponds to an incremental transfer or controller action. This style has been applied effectively to hardware languages, in the origins of structural operational semantics and recent, cycle-accurate rule-based treatments such as Bluespec/Koika [42, 9]. Moreover, techniques exist to relate stepwise and whole-execution views: the big-stop extension presents a mapping from small-step derivations to big-step judgments, allowing compact high-level proofs to be connected to low-level transition traces[26]. In order to map these methodologies to the Tydi specification, additional structure is required. Since these semantic frameworks naturally integrate with the type theory domain they are strong candidates for integrating in a Tydi formalism.

To summate, the survey of existing formalisms shows that no single approach meets Tydi’s combined needs for usability, hardware awareness, and incremental, cycle-accurate behaviour. Dependent and refinement systems capture rich static invariants but are heavyweight or essentially atemporal; sessions capture protocol definitions but lack the fine-grained spatial/temporal control that Tydi requires; and static, schedule-driven type systems (Aetherling/Filament) are ill-suited to dynamically sized, latency-insensitive streams. The software formalisms are interesting tools to reason about streamlets, but are fundamentally incompatible with the hardware domain, and would require a significant amount of work to effectively describe Tydi. Semantics, in particular the operational style, provide a natural way to

describe Tydi's incremental streamspace mapping procedure, but would still need to be connected to its type system to be used effectively.

2.4. Proposal for a tiered formal framework

By zooming out and evaluating what Tydi currently is, and what it intends to be, I note that the specification can be split in two distinct ways. One axis separates the *user-facing* from the *inner-working*, while the other divides Tydi's *static* from its *dynamic* domain. The user-facing parts of Tydi are intended to be accessible: a hardware-oriented type system, naming conventions, and the layout of data signals for a particular physical stream. The inner-workings are abstracted away; the algorithms synthesising logical streams into bundles of physical streams, the (currently non-existent) description of how streamspace mappings are actually achieved, how Tydi conformant interfaces are implemented, *why* they are compatible, etc. From the other perspective, static concepts are the hardware-oriented type system, the signal layouts and naming conventions, while dynamic concepts include engineering parameters like complexity levels, inter-stream synchronisation, and the implementation of streamspace mappings at runtime. This split allows us to apply requirements only to the relevant domain of the formalism; not encumbering the user-facing with semantic rigour, and not requiring accessibility from the type systems at the inner-working. Figure 2.1 provides a visual representation of these domains.

Formalisms can now be mapped to their most suitable domain, further aiding reasoning about what experts have referred to as the "specification gap". As noted in my conversations with the original authors, Tydi acts as a generic factory of sorts. While it succeeds at the high-level (data structures) and low-level (physical wires), it lacks a prescriptive "middle layer" or protocol layer to define how the dynamic domain is actually implemented. My intuition is that, in the 4 split view, the principle gap exists in the inner-working level, between the static and the dynamic domain.

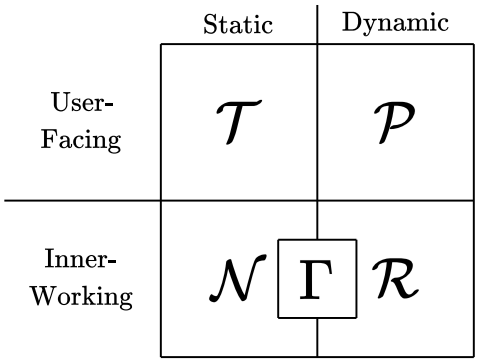


Figure 2.1: The two ways to split the formalism, showing the proposed top-level type $\mathcal{T}$, and its normalised form $\mathcal{N}$, the interface properties $\mathcal{P}$ and its corresponding ruleset $\mathcal{R}$, with the stateful device Γ connecting the static to the dynamic.

This section proposes a tiered formal framework for Tydi, where an interface specification consists independently of (1) the type $\mathcal{T}$, and (2) a set of its dynamic interface properties $\mathcal{P}$. Each can be translated to the inner-working level, where the interface type is mapped to a more hardware-oriented, unambiguous normalised form $\mathcal{N}$, and the dynamic interface properties to its corresponding semantics, implemented by a ruleset $\mathcal{R}$. In order to actually implement an interface, the gap needs to be bridged between the static and the dynamic, which is achieved by a stateful device represented by Γ. Figure 2.1 shows how these two ways to split the formalism, effectively categorise the responsibilities for each aspect of Tydi.

2.4.1. Types, normalisation, and streaming context

The existing synthesis algorithm concatenates all primitive `Bits` fields that correspond to the same physical stream in "Natural Order". This natural ordering is the order of appearance, as written down, or more specifically, pre-order DFS traversal of the type hierarchy. After synthesis, the physical stream node becomes difficult for a formalism to interact with, since the overarching type hierarchy is lost, the inter-stream relationships are hardly evident. The concatenation is still required however, since it ensures that all primitives that correspond to the same physical stream, are transmitted as one. If we can define a procedure that restructures the type while remaining in the type domain, we can derive a hardware-oriented view of the stream while maintaining inter-stream relations. One such method could concatenate the primitive fields "in-place". When extending the chat message example by adding a 32bit emoji field that is transmitted *once* per message, we get: $\texttt{Group}\langle\texttt{Bits}\langle 64\rangle, \texttt{Dim}\langle\texttt{Dim}\langle\texttt{Bits}\langle 8\rangle\rangle\rangle, \texttt{Bits}\langle 32\rangle\rangle$. The synthesis algorithm would create two physical streams, one for the timestamp + emoji, carrying 96bit elements, and another for the individual characters that comprise the message. The hierarchy between these two physical streams

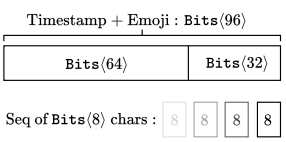


Figure 2.2: A hardware view of the registers holding a normalised timestamp + emoji, and a register for the single character input stream

is now obscured, their respective type not contained in an overarching structure. I propose a normalisation method that maintains the hierarchy, for example by normalising the emoji example to: $\texttt{Group}\langle\texttt{Bits}\langle 96\rangle, \texttt{Stream}\langle\texttt{Bits}\langle 8\rangle, d = 2\rangle\rangle$. This way we can separate the type in which an interface is defined from the type used by physical streams. Figure 2.2 shows the register view of this approach. The user-facing type maintains its flexibility, while the ”inner-working” type contains the hierarchy of physical streams, while satisfying a Canonical Form (CF) that removes semantically ambiguous structures.

The Normalisation Function

The mapping between these domains can be defined by a normalisation function $\|\cdot\| : \mathcal{T} \to \mathcal{N}$, which reduces a flexible top-level type $\mathcal{T}$ to a normalised type $\mathcal{N}$. A Canonical Form specification can enforce properties that remove semantically ambiguous structures in the normalised type. It should disallow directly nested streams or groups, since semantically, $\texttt{Stream}\langle\texttt{Stream}\langle T, d = 1\rangle, d = 1\rangle \equiv \texttt{Stream}\langle T, d = 2\rangle$, similarly any group composition of the form $\texttt{Group}\langle T_1, \ldots, \texttt{Group}\langle T'_1, \ldots, T'_m\rangle, \ldots, T_n\rangle$ is semantically equivalent to $\texttt{Group}\langle T_1, \ldots, T'_1, \ldots, T'_m, \ldots, T_n\rangle$. Singular groups can be reduced: $\texttt{Group}\langle T\rangle \equiv T$. The index of a primitive in a group in the normalised domain does not carry any semantic implications, therefore it should be moved to a consistent location by always placing it at the head.

These procedures guarantee that at most one primitive exists per stream node, and its location in the type is deterministic, while maintaining the hierarchy of the streams. Additionally, the Tydi shorthand notations for the stream node (Dim, New, etc), and the container types (ConcatStruct, PackedVariant, RASElem etc), are all defined by the streamspace types (Bits, Group, Union, etc). The type system that the dynamic formalism interacts with should ideally not include the shorthand and container types. Just like the synthesis function reduces the shorthand notations to bundles of physical streams, my proposal separates the top level from the normalised type, allowing for flexibility, and extensibility at the top level, while retaining the semantic clarity in the normalised domain.

The Streaming Context

As we've seen, the normalised type in canonical form provides a deterministic structure for the dynamic formalism to interact with. However, since we explicitly decoupled the static from the dynamic specification, we have thus far not considered how the type of an interface, actually impacts the runtime behaviour. As shown in diagram 2.1, the dynamic, operational domain interacts with the static, type domain through a streaming context Γ. The streaming context is stateful and deterministically instantiated by the normalised type $\mathcal{N}$. It exposes the fixed set of “dials and switches” by which the dynamic formalism can operate; the dynamic formalism never uses $\mathcal{N}$ directly, it only manipulates Γ.

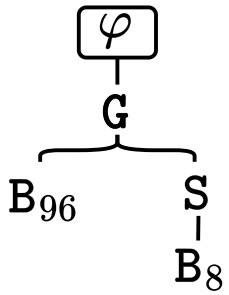


Figure 2.3: The parse tree for the normalised chat message example where φ indicates the root in focus.

Concretely, Γ materializes the explicit buffers, per-stream wire layouts, how the component's internal streams are connected with the tydi interface and a parse tree that maintains the state of the hierarchy of nested streams. All these components should have exact sizes at instantiation, since we cannot define buffers or trees of arbitrary size. The parse tree is constructed based on the normalised type to maintain the hierarchical relations between streams. By terminating nodes in a tree, various behaviours and interleavings can be implemented. For the chat message example, this gives the tree shown in figure 2.3. The root φ of this tree encapsulates the normalised type in a stream with a singular dimension. It carries the termination information of the outer sequence. Like an API of sorts, the streaming context exposes a set of operations to the dynamic formalism, that can use them to implement interface behaviour. These operations can for example inspect the current state of the streaming, send data, update buffers etc. The dynamic formalism would therefore be defined as a set of transitions on this streaming context, $\Gamma \to \Gamma'$.

2.4.2. Interface Properties and Semantics

The dynamic domain of the formalism also has its own user-facing and inner-working split. In the current specification, the parts of Tydi that specify the dynamic properties of a stream are intertwined with the static; The Logical Stream definition contains the stream's type and parameters impacting the static signal layout, alongside the synchronicity and complexity levels impacting the dynamic nature

of the interface. The complexity level is the main dynamic interface property, directly determining the allowed streamspace mappings. A notable feature of Tydi is the use of this parameter to balance the complexity of the implementation logic of the sending and receiving interfaces. However, it is never substantiated *how* the complexity levels actually achieve this balancing.

A larger perspective on defining hardware interfaces, allows us to reason about constraints on inputs of an interface, and guarantees it provides regarding its output. The interface of a component can be considered a streamlet of a specific kind, where its inputs are the way it is connected to the component, and its output is the set of wires connecting to the interface of a receiving component. When reasoning about the implementation complexity of an interface, we cannot only rely on the properties of the output of the interface, we must also consider the way this interface is connected to the internals of the component.

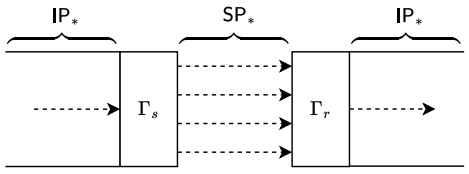


Figure 2.4: The sending and receiving component with their respective interfaces Γ and internal stream properties IP_*. The transmission is governed by the streamspace properties SP_*.

For example, component 1 processes chat messages and we want to construct a sending interface for component 1, incrementally sending a chat message to component 2. If component 1 internally takes more time for certain letters, it cannot always guarantee that the next letter will be available for sending. The implementation of the sending interface will need to take this into account. Crucially, if the output of the sending interface, also does not guarantee that the stream of letters is sent without interruptions, the internal ”slowness” of component 1 has less of an impact on the implementation complexity of its sending interface. Whereas, if the sending interface *does* guarantee a continuous stream of letters, this will require additional logic and buffering, increasing the complexity.

I note a dual: strong input constraints and weak output guarantees decrease implementation complexity, while weak input constraints and strong output guarantees increase implementation complexity. This aligns well with the field of formal interface contracts. Such contracts can be used to reason about compatible interfaces, formalising their assumptions/constraints and their guarantees, providing strong connection to their compositional behaviour in the hardware domain [7, 4]. The classes of properties describing such constraints and guarantees, can be differentiated as shown in Figure 2.4. It shows how the properties describing the internal streams of the component, are separated from those describing the streamspace mapping, the communication between components. When considering the compatibility of interfaces, only the Streamspace Properties are relevant. As such I define interfaces as compatible if the Streamspace Properties of the sender and receiver are identical. Concretely, I distinguish the properties as follows:

- IP_* *Internal-Stream Properties:* predicates over normalised element streams inside a component. Examples include element ordering, presence/absence of empty elements, and per-element termination markers.
- SP_* *Streamspace Properties:* predicates over sequences of transfers. Examples include transfer utilisation, element alignment, and sequence-termination. The original Tydi complexity levels *only* describe properties in this domain.

This allows describing the original complexity levels, but also other combinations of interface properties. The original authors have expressed that the hierarchy of complexity levels is somewhat arbitrary, obscuring the fact that interfaces might have properties that do not fit neatly in a specific level. For instance, if an interface requires a high-complexity feature like lane offset but adheres to low-complexity constraints otherwise, it must currently be described as ”high complexity”. The sets of dynamic interface properties would alleviate this restriction, allowing the designer of an interface to pick the exact interface properties that suit their needs.

Named Property Sets

At the user facing level, I initially proposed that each interface property relates to a specific transition in the semantics, forming a convex hull of rules where each subset would describe a valid interface. This was however needlessly complicated, making individual properties less intuitive, the underlying semantics exceedingly complex, while allowing the description of rather ill-advised interfaces (e.g. an

interface that has no delays on its internal streams, but introduces delays in its streamspace mapping). Therefore, I propose to usage of "Named Property Sets", similar to the complexity levels, but without the inherent hierarchy. Each property set $\mathcal{P}$ maps to a corresponding semantics $\mathcal{P}$ at the inner-working level. This is related to the idea that motivated the original Standard Container Library, providing engineers with a shorthand notation corresponding to higher level structures that one may be more familiar with. The first named property set that should be introduced is a "core", implementing a rudimentary mapping procedure.

Semantics

The property sets are strictly related to sets of transitions, expressed in an operational small-step semantics. Each transition is can be seen as a "rule", governing the allowed behaviour of some interface. The terminology of rule and transition is generally used interchangeably. Rule-based, hardware-oriented approaches such as Bluespec/Koika give a valuable perspective on how one might implement formal hardware descriptions, while remaining explicit about resource bounds needed for synthesis, and providing the methodologies used to verify the formalism [9]. It is my intention to first define a set of default rules required for interacting with the streaming context. These rules will be shared among all rulesets, and can be extended with the transitions that are needed to implement the behaviour specified by the property-set. To make the semantics tractable and focused on the mapping behaviour I introduce three simplifying but deliberate abstractions up front:

1. **Temporal granularity (handshaked transfers).** Time is quantified in handshaked transfers (valid/ready occurrences). I abstract over inter-transfer cycles and sub-cycle timing. This concentrates the semantics on *when* and *which* elements are presented and acknowledged, rather than on low-level timing artefacts that are irrelevant for the purpose of this work.
2. **Deferral of bit-level layout.** The explicit layout of lanes and signals are omitted from the proposed formalism, since these can be inferred from the original specification. Omitting these details reduces clutter in the semantics.
3. **Staged support for `Union`.** The initial formalism excludes the `Union` (variant) from the core development. This choice isolates the canonical mapping behaviour for sequences, groups, and sequences. The semantics and proofs are constructed such that the `Union` can be reintroduced as an extension of the formalism. Its inclusion is presented in chapter 6, and illustrates how the framework is an effective foundation for the entire Tydi specification.

Proving functionality

A formal foundation for Tydi facilitates rigorous verification of the interfaces that it defines. I need to verify the correctness of the mapping of an arbitrary data instance of a given type. A key challenge is ensuring that this holds true, not just for the core ruleset, but also for any potential alterations of its semantics. Therefore, I will base the correctness argument on two fundamental properties that *any* valid ruleset $\mathcal{R}$ must adhere to: *1) Structural Determinism:* For any valid streaming context Γ, exactly one transition rule must be applicable at all times. This can be enforced by defining rules with mutually exclusive and exhaustive premises [2]. *2) Progress (Finite Parsing):* For any finite instance, its mapping must be implemented in finite transitions. I ensure this by requiring that every rule (or finite sequence of rules) advances the context by strictly decreasing a well-founded metric (e.g., by consuming input or terminating a node), thus guaranteeing termination. This method is inspired by Kôika, an operational semantic description for Bluespec System Verilog (BSV), enabling formal reasoning about its execution, maintaining a hardware aware, cycle-accurate perspective [9]. These arguments will only work if the normalisation always maps a top level type to canonical form, and if terminating streams always validly terminate the parsetree. Therefore, these processes will need to be verified beforehand.

2.4.3. Connecting the layers of the formalism

While the original specification hardly mentions the receiving perspective of streamspace, it doesn't really need to since the signal layout is static, and the streamspace mapping is inexact. The receiving interface is mentioned in the specification, regarding the balancing of hardware implementation logic complexity of the sender and receiver. It is however never substantiated how this balancing of logic is actually achieved. Since this work provides sufficient detail to describe the actual mapping procedure, it is important to specify how this connects to the receiving interface, and the balancing of implementation

complexity, as these are valuable aspects of the specification. This section will address both these subjects, finalising in a brief overview of the proposed formalism, how it relates to the requirements identified in the previous chapter and where it deviates from the original specification.

The receiving interface

For the purpose of this work I focus on the sender interface, illustrating how its internal stream and streamspace properties give rise to the semantics describing its runtime mapping procedure. The semantics of the receiving interface can be implemented as the dual of the sending interface, given that the its ruleset is structurally deterministic and injective. More specifically, $\Gamma \xrightarrow{\tau} \Gamma'$ denotes the sender small-step transition that emits a transfer (observation) τ. We require that for every context Γ there is at most one outgoing step: $\left|\{(\tau, \Gamma') \mid \Gamma \xrightarrow{\tau} \Gamma'\}\right| \leq 1$. This reads as, "the number of possible pairs of transfer and successor state (τ, Γ'), that can be reached from a given context Γ is at most one." For injectivity, distinct predecessors cannot reach the same successor under the same observation: $(\Gamma_1 \xrightarrow{\tau} \Gamma') \wedge (\Gamma_2 \xrightarrow{\tau} \Gamma') \Rightarrow \Gamma_1 = \Gamma_2$.

A Perspective on Implementation Complexity

The original specification mentions balancing the required hardware logic to implement a sender-receiver interface set, but does not substantiate how different configurations actually achieve this. My proposed separation of Internal-Stream Properties (IP_*) and Streamspace Properties (SP_*) does more than define compatibility; it directly governs the implementation complexity of the sending and receiving logic. This idea is natural to the field of interface automata [4], where relaxing input assumptions or strengthening output guarantees strictly influence the state space a component must manage. Therefore, the proposed interface properties also enable the design of a quantifiable *Implementation Complexity Metric (ICM)*. I propose defining a mapping from the semantic definition of an interface, to its automata or FSM equivalent, and analysing the state space. Such a metric will allow us to formally reason about the trade-offs between interface implementations of specific types, and their resulting hardware logic cost.

Deviations from the original specification

Besides starting with a subset of the type system, excluding the union, there are fundamental differences. In the proposed formalism, all streams in a type must adhere to the same properties, while Tydi allows interleavings where only some nested physical streams are not synchronised to their parent streams.

Tydi distinguishes complexity level 1 and 2 from the rest, by reasoning about the valid signal not going down. Since the proposed formalism abstracts the temporal resolution of streamspace as "handshaked transfers", it is not expressive enough to reason about signal validity between those transfers. For this formalism, the gaps that are shown between the transfers corresponding to complexity level 2 in figure 1.5, are indistinguishable from the directly adjacent transfers. To represent such a gap, an empty transfer would need to sent, with a handshake taking place.

An overview of the proposal

The proposed tiered formal framework addresses the "specification gap" by distinguishing between the user-facing definition and the inner-working semantics. An interface is specified by a top-level type $\mathcal{T}$ and a set of properties $\mathcal{P}$. The type $\mathcal{T}$ is algorithmically reduced to a hardware-oriented Normalised form $\mathcal{N}$ via a normalisation function $[\![\cdot]\!]$. This normalised type instantiates a stateful Streaming Context Γ, which bridges the static and dynamic domains. The runtime behaviour, the mapping of data to streamspace, is then governed by a ruleset $\mathcal{R}$ derived from the properties $\mathcal{P}$.

3

The Formalism

Building on the findings from the analysis, this chapter defines the formalism. It is structured to mirror the path from design intent to hardware behaviour. The highest level of abstraction encompasses the entire definition of an interface. It is defined as a tuple: InterfaceSpecification $:= (\mathcal{T}, \mathcal{P})$ consisting of the static type of the interface $\mathcal{T}$, and the set of dynamic interface properties $\mathcal{P}$. The *top-level type ($\mathcal{T}$)* expresses the type of the interface. The *normalised type ($\mathcal{N}$)*, is defined to be of Canonical Form (CF) when it adheres to a set of conditions, disallowing semantic ambiguous structures. The normalisation function ($[\![\cdot]\!]$) algorithmically maps $\mathcal{T}$ to $\mathcal{N}$ and enforces the Canonical Form. A stateful *streaming context (Γ)* is defined, instantiated using the normalised type $\mathcal{N}$, and the user specified number of lanes L It encapsulates all runtime state, including input buffers (I), a navigable *parse tree ($Z_{\mathcal{N}}$)* and transfer buffers T_B. The type agnostic, incremental mapping procedure, is defined by the provided interface property set $\mathcal{P}$. A mapping procedure $\mathcal{R}$ is based on its corresponding property set, and defined as a ruleset composed of *small-step operational semantics* ($\rightarrow$) transitions that operate on Γ. Deterministic sequences of such transitions are abstracted by *multi-step transitions* ($\rightarrow^*$), enabling reasoning about the mapping of a complete data instance as a sequences of *cycles*. Finally, it is shown how an interface specification can algorithmically map to an Abstract Finite State Machine, which is used to define an *Implementation Complexity Metric (ICM)*. This provides a method for quantifying the complexity of the logic required to implement a given interface specification in hardware.

3.1. Types, Normalisation and the Streaming Context

3.1.1. Top-Level and Normalised Types

We define the *top-level Type* $\mathcal{T}$ as a grammar, shown in figure 3.1. Here, $\texttt{Bits}(n)$ represents a primitive type with n bits, $\texttt{Group}(\mathcal{T}_1, \ldots, \mathcal{T}_k)$ forms a group of k fields. $\texttt{Dim}(\mathcal{T})$ annotates adding a dimension, defining a sequence, or lifting a sequence to a higher dimension through nesting $\texttt{Dim}(\texttt{Dim}(\mathcal{T}))$. The *normalised type* $\mathcal{N}$ adapts $\mathcal{T}$ by introducing an explicit stream constructor. Here, $\texttt{Stream}(\mathcal{N}, d)$ is a stream of normalised type $\mathcal{N}$ with dimension d.

$$
\begin{aligned}
\mathcal{T} \quad ::= \quad & \texttt{Bits}(n) \qquad (n \in \mathbb{N}) \\
\mid \quad & \texttt{Group}(\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_k) \qquad (k \geq 1) \\
\mid \quad & \texttt{Dim}(\mathcal{T}) \\
\\
\mathcal{N} \quad ::= \quad & \texttt{Bits}(n) \qquad (n \in \mathbb{N}) \\
\mid \quad & \texttt{Group}(\mathcal{N}_1, \mathcal{N}_2, \ldots, \mathcal{N}_k) \qquad (k \geq 1) \\
\mid \quad & \texttt{Stream}(\mathcal{N}, d) \qquad (d \in \mathbb{N}^+)
\end{aligned}
$$

Figure 3.1: The top-level and normalised types

3.1.2. Normalisation Function

We define the normalisation function: $[\![\cdot]\!] : \mathcal{T} \rightarrow \mathcal{N}$ as shown in figure 3.2. It maps a top-level type to a normalised type using two auxiliary functions. $\mathbf{coalesce}$ takes a group of normalised types $(\mathcal{N}_1, \ldots, \mathcal{N}_k)$ and constructs flattened groups by: 1) replacing nested groups by their contents: $\texttt{Group}(\mathcal{N}_1, \texttt{Group}(\mathcal{N}_2, \mathcal{N}_3))$ becomes $\texttt{Group}\ (\mathcal{N}_1, \mathcal{N}_2, \mathcal{N}_3)$, 2) concatenating all $\texttt{Bits}(n_i)$ fields, summing their widths into a single $\texttt{Bits}(n)$ placed at the first index of the group, and 3) if the resulting group contains has only 1 field, it removes the outer group: $\text{coalesce}(\texttt{Group}(\mathcal{N})) \mapsto \mathcal{N}$. The $\mathbf{lift}$ function sums nested stream

dimensions, leaving other types unchanged: $\mathrm{lift}(\texttt{Stream}(\texttt{Stream}(\mathcal{N}', d'), d)) \mapsto \texttt{Stream}(\mathcal{N}', d + d')$

$$\begin{aligned} [\![\texttt{Bits}(n)]\!] &\mapsto \texttt{Bits}(n) \\ [\![\texttt{Group}(\mathcal{T}_1, \ldots, \mathcal{T}_k)]\!] &\mapsto \mathbf{coalesce}\Big(\texttt{Group}([\![\mathcal{T}_1]\!], \ldots, [\![\mathcal{T}_k]\!])\Big) \\ [\![\texttt{Dim}(\mathcal{T})]\!] &\mapsto \mathbf{lift}\Big(\texttt{Stream}([\![\mathcal{T}]\!], 1)\Big) \end{aligned}$$

Figure 3.2: Recursive definition of the normalisation function

3.1.3. The Streaming Context

The context Γ is the stateful device that orchestrates the mapping procedure. It is constructed as $\Gamma(\mathcal{N}, L) \coloneqq (I, Z_{\mathcal{N}}, T_B)$, where $\mathcal{N}$ is the normalised type definition and $L = (l_1, \ldots, l_n)$ is a tuple specifying the number of lanes per normalised physical stream. This constructs the tuple where: I is a tuple of input stream prefixes, holding the next available element i_k for each stream. We define an element as a tuple: $i_k := (t_k, v_k)$ where k is the index of the stream in focus, $t_k \in \{0, \ldots, d_k\}$ is the termination depth, a natural number indicating which dimension of the sequence is terminated by this element, where d_k is the dimension of the respective stream, and v_k is the value representing the data carried by an element, which either has the type of its normalised stream or is empty: $\Gamma \vdash i_k : (t_k, v_k)$ where $0 \leq t_k \leq d_k$ and $v_k \in \{B_k, \varepsilon\}$. Terminating any dimension constitutes the termination of all lower dimensions, e.g. terminating the entire chat message $(t_k = 2)$, also means termination of the last word $(t_k = 1)$. $Z_{\mathcal{N}}$ is a navigable parse tree representing the instance of $\mathcal{N}$ being processed, with the root node represented as φ. Each node can be marked as *terminated*. T_B is a tuple of transfer buffers, one buffer T_{B_k} per physical stream k, each matching the sizes specified in L, and can be marked as terminated. The slots of buffers are distinctly initialized with $\varnothing$. On the context Γ, I define $\mathrm{metaFunctions}(\Gamma)$ and $\Gamma.\texttt{context\_operations}$, where the former leaves the context unchanged, and the latter performs context transformations:

$$\begin{aligned} \mathrm{focus}(\Gamma) &:= \text{returns the type of the node currently in focus,} \\ \mathrm{isLeaf}(\Gamma) &:= \text{returns true iff all direct child nodes of the focus are terminated,} \\ \Gamma.\texttt{next} &:= \text{advances focus to the next non-terminated node (post-order DFS),} \\ \Gamma.\texttt{term} &:= \text{marks the focused node terminated, subsequently invokes } \texttt{next}, \\ \Gamma.\texttt{reset} &:= \text{restores initial parse tree state (all nodes non-terminated),} \\ \Gamma.\texttt{parse} &:= \text{maps } i_k \text{ into its transfer buffer; if full or } t_k == d_k, \text{ invokes } \texttt{send}, \\ \Gamma.\texttt{send} &:= \text{marks the current transfer buffer } T_{B_k} \text{ as terminal,} \\ \Gamma.\texttt{cycle} &:= \text{produces terminated transfers and refreshes the parsed input buffers.} \end{aligned}$$

The post-order, depth first search traversal of the parse tree is crucial, since it ensures that 1) nodes are visited from left to right, and 2) all nodes are visited before any of their parent nodes are. These properties are essential for adhering to the notion of ”natural ordering” from the original spec, while ensuring proper parse tree termination. Natural ordering is equal to pre-order DFS, which would not be ideal for the proposed formalism, since the evaluation of the data carrying nodes would occur after the structural nodes, making termination logic more complex.

3.2. Interface Properties and Semantics

The interface properties $\mathcal{P}$ from the interface specification, describe the assumptions and guarantees regarding the data on a component’s internal streams, and the streamspace representation of this data. This section will introduce a variety of properties that be combined to form a property-set. The complexity levels from the original specification are an example of such sets. ”Named Property-sets” are predetermined sets of properties with specific characteristics. These facilitate reasoning about interface behaviour and improve accessibility by allowing engineers to specify interfaces without manually composing individual properties.

3.2.1. Interface Properties

Internal-stream Properties

Predicates over the component-local element streams (I).

$\mathsf{IP}_{\mathsf{TO}}$ **Internal true order.** Let $E_k = (e_1, e_2, \ldots, e_m)$ be the ordered sequence of elements for a stream k as defined by a complete data instance. Let $C(\Gamma \xrightarrow{*} \Gamma')$ be the sequence of elements i_k consumed from the input prefix I_k during the full mapping $\Gamma \rightarrow^* \Gamma'$. This property asserts that $C(\Gamma \xrightarrow{*} \Gamma') = E_k$. The logical order of the data instance is strictly preserved on the component's input.

$\mathsf{IP}_{\mathsf{Coh}}$ **Input cohesion.** This property guarantees data availability for every input element. For any stream k, when an element $i_k = (t_k, v_k)$ is present in the input prefix I, its value v_k must not be empty. Formally: $\forall k, \forall i_k \in I,\ v_k \neq \varepsilon$. This implies $v_k \in \{B_k\}$. This property simplifies the core ruleset by guaranteeing that any focused `Bits` node has valid data to parse.

$\mathsf{IP}_{\mathsf{ElemTerm}}$ **Data element termination.** This property asserts that a termination signal must be attached to a data-carrying element. An element cannot *only* be a termination signal. F or any input element $i_k = (t_k, v_k)$, the value v_k can only be empty (ε) if no dimension is being terminated ($t_k = 0$). Formally: $v_k = \varepsilon \implies t_k = 0$. This ensures that the last element of any sequence (which must have $t_k > 0$) also carries a valid data payload $v_k = B_k$.

Streamspace Properties

Predicates over the observable wire-level sequences of transfers (τ).

$\mathsf{SP}_{\mathsf{TO}}$ **True order.** This is the streamspace equivalent of $\mathsf{IP}_{\mathsf{TO}}$. It asserts that the relative order of elements consumed from I_k is strictly preserved in their placement onto the output transfers. If element e_a is consumed before e_b, then e_a will be placed in streamspace in a transfer τ_i at lane l_x that is temporally or spatially precedent to e_b's placement in τ_j at l_y (i.e., $i < j$ or $i = j \wedge x < y$).

$\mathsf{SP}_{\mathsf{Coh}}$ **Transfer cohesion.** Elements mapped into a single transfer buffer T_{B_k} must occupy consecutive lanes. If T_{B_k} has L_k lanes $[0 \ldots L_k - 1]$ and the `parse` operation has filled slots $[0 \ldots j]$, the next element will be placed starting at slot $j + 1$. No $\varnothing$ slots (gaps) are permitted between valid elements within an emitted transfer.

$\mathsf{SP}_{\mathsf{Util}}$ **Transfer utilisation.** Every transfer τ emitted to streamspace must contain at least one data element. A transfer where all lanes are $\varnothing$ is forbidden. This is enforced by the core semantics, as the Γ.`send` operation (which marks a buffer for emission) is only invoked by Γ.`parse`, which is triggered by the data-processing rules Bits-Parse and Bits-Term .

$\mathsf{SP}_{\mathsf{Sync}}$ **Inter-stream sync.** This property guarantees transactional atomicity for a full data instance. Let T_n be the sequence of *all* transfers generated by the mapping of data instance n. This property asserts that the full sequence of transfers T is a concatenation $\cdots \circ \mathrm{T}_n \circ \mathrm{T}_{n+1} \circ \ldots$. No transfer from T_{n+1} can be emitted until the M-Full-Map transition for instance n has occurred.

$\mathsf{SP}_{\mathsf{Align}}$ **Element alignment.** When mapping elements to an empty transfer buffer T_{B_k}, the first element is always placed starting at the least-significant lane (lane 0). This disallows arbitrary offsets, simplifying receiver logic. Subsequent elements are packed contiguously per $\mathsf{SP}_{\mathsf{Coh}}$.

$\mathsf{SP}_{\mathsf{ElemTerm}}$ **Data element termination.** The streamspace equivalent of $\mathsf{IP}_{\mathsf{ElemTerm}}$. It guarantees that a sequence's termination signal is contained in the same element as its final data element. If an element $i_k = (t_k, B_k)$ with $t_k = d_k$ is the last element of a sequence, the Bits-Term rule ensures that both B_k and the signal t_k are parsed and emitted in the same transfer.

$\mathsf{SP}_{\mathsf{TransTerm}}$ **Transfer element termination.** This is a weaker version of $\mathsf{SP}_{\mathsf{ElemTerm}}$. It guarantees that a sequence's termination signal is contained in the same transfer as its final data element. This does not disallow the termination signal to be contained in an element without data.

$\mathsf{SP}_{\mathsf{TermInner}}$ **Terminate inner dimensions.** This property enforces that any end of a subsequence of stream k, will terminate the respective transfer buffer T_{B_k}. More specifically, for element $i_k = (t_k, v_k)$ with $d_k > t_k > 0$, the parsing of this element will ensure the corresponding transfer buffer is sent to streamspace on the next cycle.

3.2.2. Default interface semantics

The transitions that are shared across all property-sets interface properties, are the ones responsible for the traversal of the parsetree, the cycle behaviour and the termination of the full data instance. These are captured in single step semantics for the tree traversal, and multi step semantics for the cycle and full-map transitions.

Parse tree traversal

The traversal and termination of the parse tree is achieved by two rules shown in figure 3.3 Struct-Enter and Struct-Term. Struct-Enter traverses the parse tree when the node in focus is of type Stream or Group and has non-terminated children. Struct-Term terminates the node in focus and traverses the parse tree when the node in focus is of type Stream or Group, and its children are terminated.

$$\text{Struct-Enter}\ \frac{\mathsf{focus}(\Gamma) \in \{S, G\} \quad \neg\mathsf{isLeaf}(\Gamma)}{\Gamma \to \Gamma.\texttt{next}} \qquad \text{Struct-Term}\ \frac{\mathsf{focus}(\Gamma) \in \{S, G\} \quad \mathsf{isLeaf}(\Gamma)}{\Gamma \to \Gamma.\texttt{term}}$$

Figure 3.3: The default single-step transitions responsible for traversing the parse tree

Multi-step transitions

Each full traversal of the parse tree constitutes a cycle, updating buffers, and potentially resulting in observable transfers. To enable reasoning about a sequence of transitions I first define a reflexive transitive closure $\to^*$ inductively by M-Step and M-Refl shown in figure 3.4. Here, M-Refl expresses base case of zero steps, and M-Step composes successive small-step transitions. M-Cycle represents the completion of one traversal of the parse tree, from root to root. M-Full-Map captures the complete mapping of an entire data instance, resetting the context to prepare for the next instance. These definitions simultaneously represent the

$$\text{M-Refl}\ \frac{}{\Gamma \to^* \Gamma} \qquad \text{M-Step}\ \frac{\Gamma \to \Gamma' \qquad \Gamma' \to^* \Gamma''}{\Gamma \to^* \Gamma''}$$

Figure 3.4: The definition of $\to^*$

$$\text{M-Cycle}\ \frac{\mathsf{focus}(\Gamma) = \mathsf{focus}(\Gamma') = \varphi \qquad \neg\mathsf{isLeaf}(\Gamma')}{\Gamma \to^* (\Gamma'.\texttt{cycle}).\texttt{next}} \qquad \text{M-Full-Map}\ \frac{\mathsf{focus}(\Gamma) = \mathsf{focus}(\Gamma') = \varphi \qquad \mathsf{isLeaf}(\Gamma')}{\Gamma \to^* (\Gamma'.\texttt{cycle}).\texttt{reset}}$$

Figure 3.5: The multi-step transitions

single and multi-step transitions. Notably, the notion of a cycle is not just the traversal of the tree, but can be translated to the hardware domain, albeit not one to one. The sequence of transitions after M-Cycle is invoked is fully deterministic, ending with the focus back a the root. This is the case since the variables that determine the sequence, the inputs and the transfer buffers, are updated per cycle. Thus we may reason about the mapping of a complete data instance as a sequence of cycles, by M-Full-Map.

3.3. Named Property-Sets

To provide a basic implementation of the formalism, a named property set for the core is introduced, alongside two complexity levels.

3.3.1. The Core

The core is a named property-set corresponding to the basic mapping procedure. It is characterized by the following interface properties:

$$\mathcal{P}_{core} := \{\mathsf{IP}_{\mathsf{TO}},\ \mathsf{IP}_{\mathsf{Coh}},\ \mathsf{IP}_{\mathsf{ElemTerm}},\ \mathsf{SP}_{\mathsf{TO}},\ \mathsf{SP}_{\mathsf{Coh}},\ \mathsf{SP}_{\mathsf{Util}},\ \mathsf{SP}_{\mathsf{Sync}},\ \mathsf{SP}_{\mathsf{Align}},\ \mathsf{SP}_{\mathsf{ElemTerm}}\}$$

The core is implemented as a set of *small-step* transitions $(\to)$ that traverse the parse tree, iterating over input buffers and mapping elements to transfer buffers.

The core ruleset $\mathcal{R}_{core}$ consists of the default semantics together with two rules governing the parsing and termination of Bits nodes shown in figure 3.6. Bits-Parse applies when the focused node is of type Bits and the input element is non-terminal, while Bits-Term applies when the input element terminates the outer dimension.

Struct-Enter
$$\frac{\mathsf{focus}(\Gamma) \in \{S, G\} \quad \neg\mathsf{isLeaf}(\Gamma)}{\Gamma \to \Gamma.\texttt{next}}$$

Struct-Term
$$\frac{\mathsf{focus}(\Gamma) \in \{S, G\} \quad \mathsf{isLeaf}(\Gamma)}{\Gamma \to \Gamma.\texttt{term}}$$

M-Cycle
$$\frac{\mathsf{focus}(\Gamma) = \mathsf{focus}(\Gamma') = \varphi \quad \neg\mathsf{isLeaf}(\Gamma')}{\Gamma \to^* (\Gamma'.\texttt{cycle}).\texttt{next}}$$

M-Full-Map
$$\frac{\mathsf{focus}(\Gamma) = \mathsf{focus}(\Gamma') = \varphi \quad \mathsf{isLeaf}(\Gamma')}{\Gamma \to^* (\Gamma'.\texttt{cycle}).\texttt{reset}}$$

Bits-Parse
$$\frac{\mathsf{focus}(\Gamma) = B_k \quad \Gamma \vdash i_k : (t_k, B_k) \quad t_k < d_k}{\Gamma \to (\Gamma.\texttt{parse}).\texttt{next}}$$

Bits-Term
$$\frac{\mathsf{focus}(\Gamma) = B_k \quad \Gamma \vdash i_k : (t_k, B_k) \quad t_k = d_k}{\Gamma \to (\Gamma.\texttt{parse}).\texttt{term}}$$

Figure 3.6: The default semantics with the core transitions responsible for parsing and terminating Bits nodes.

3.3.2. Complexity Levels

The complexity levels from the original specification can be seen as named property sets. Here I will provide the property-sets for various complexities. Since the original specification does not reason about properties of internal streams, these are interpreted to be their streamspace equivalent. For example, if some complexity level does not include empty element in its streamspace representation, Input Cohesion will also not be violated.

C = 3: For complexity level 3, terminating innermost sequences terminates the transfer. To accomplish this, we alter the core property-set to send on inner dimensions. Instead of removing constraints from the core, we add $\mathsf{SP}_{\mathsf{TermInner}}$, ensuring that inner dimensions also terminate transfers:

$$\mathcal{P}_{C3} := \{\mathsf{IP}_{\mathsf{TO}},\ \mathsf{IP}_{\mathsf{Coh}},\ \mathsf{IP}_{\mathsf{ElemTerm}},\ \mathsf{SP}_{\mathsf{TO}},\ \mathsf{SP}_{\mathsf{Coh}},\ \mathsf{SP}_{\mathsf{Util}},\ \mathsf{SP}_{\mathsf{Sync}},\ \mathsf{SP}_{\mathsf{Align}},\ \mathsf{SP}_{\mathsf{ElemTerm}},\ \mathsf{SP}_{\mathsf{TermInner}}\}$$

Implementing this behaviour requires the premise of Bits-Parse to be updated to $t_k = 0$, maintaining structural determinism when Bits-Send-Inner is added. For $\mathcal{R}_{C3}$, shown in figure 3.7, the resulting behaviour is mapping $a)$ from figure 1.4.

Struct-Enter
$$\frac{\mathsf{focus}(\Gamma) \in \{S, G\} \quad \neg\mathsf{isLeaf}(\Gamma)}{\Gamma \to \Gamma.\texttt{next}}$$

Struct-Term
$$\frac{\mathsf{focus}(\Gamma) \in \{S, G\} \quad \mathsf{isLeaf}(\Gamma)}{\Gamma \to \Gamma.\texttt{term}}$$

M-Cycle
$$\frac{\mathsf{focus}(\Gamma) = \mathsf{focus}(\Gamma') = \varphi \quad \neg\mathsf{isLeaf}(\Gamma')}{\Gamma \to^* (\Gamma'.\texttt{cycle}).\texttt{next}}$$

M-Full-Map
$$\frac{\mathsf{focus}(\Gamma) = \mathsf{focus}(\Gamma') = \varphi \quad \mathsf{isLeaf}(\Gamma')}{\Gamma \to^* (\Gamma'.\texttt{cycle}).\texttt{reset}}$$

Bits-Parse
$$\frac{\mathsf{focus}(\Gamma) = B_k \quad \Gamma \vdash i_k : (t_k, B_k) \quad t_k = 0}{\Gamma \to (\Gamma.\texttt{parse}).\texttt{next}}$$

Bits-Term
$$\frac{\mathsf{focus}(\Gamma) = B_k \quad \Gamma \vdash i_k : (t_k, B_k) \quad t_k = d_k}{\Gamma \to (\Gamma.\texttt{parse}).\texttt{term}}$$

Bits-Send-Inner
$$\frac{\mathsf{focus}(\Gamma) = B_k \quad \Gamma \vdash i_k : (t_k, v_k) \quad 0 < t_k < d_k}{((\Gamma \to \Gamma.\texttt{parse}).\texttt{send}).\texttt{next}}$$

Figure 3.7: The complete ruleset for complexity level $C = 3$.

C = 4 When the signal indicating the end of a (sub)sequence, can be delayed on the internal streams, we violate $\mathsf{IP}_{\mathsf{ElemTerm}}$, which forces this signal to arrive on an empty element, making $i_k : (t_k, \varepsilon)$ where $t_k > 0$ valid input, thereby also violating $\mathsf{IP}_{\mathsf{Coh}}$. Relaxing this constraint requires the inclusion of Bits-Empty-Skip. If the empty element is mapped onto streamspace, $\mathsf{SP}_{\mathsf{ElemTerm}}$ is also violated. Furthermore, since only at $C = 5$ the inner sequences can be split up across multiple transfers, the interface adheres to $\mathsf{SP}_{\mathsf{TermInner}}$ and Bits-Send-Inner needs to be incorporated. To accommodate these changes, I propose the addition of Bits-Empty-Skip and Bits-Empty-Term, which match the type in their premises. Bits-Empty-Skip ensures we do not map empty elements without termination data to streamspace,

Bits-Send-Inner will send empty elements to streamspace if they terminate an inner sequence, and Bits-Empty-Term will terminate the node and send the transfer if the empty element carries the outer dimension termination signal. Notice that any transfer will at least contain termination information, and thus $\mathsf{SP}_{\mathsf{Util}}$ is not violated, since $\varepsilon \neq \varnothing$. This gives the property-set for complexity level 4:

$$\mathcal{P}_{C4} := \{\mathsf{IP}_{\mathsf{TO}},\ \mathsf{SP}_{\mathsf{TO}},\ \mathsf{SP}_{\mathsf{Coh}},\ \mathsf{SP}_{\mathsf{Util}},\ \mathsf{SP}_{\mathsf{Sync}},\ \mathsf{SP}_{\mathsf{Align}},\ \mathsf{SP}_{\mathsf{TermInner}}\}$$

The ruleset $\mathcal{R}_{C4}$ corresponding to complexity level 4, and its property-set $\mathcal{P}_{C4}$, is shown in figure 3.8. Bits-Empty-Term is equal to the core semantics of Bits-Term, except it specifies the type of the input element as ε.

Struct-Enter
$$\frac{\mathsf{focus}(\Gamma) \in \{S, G\} \quad \neg\mathsf{isLeaf}(\Gamma)}{\Gamma \to \Gamma.\texttt{next}}$$

Struct-Term
$$\frac{\mathsf{focus}(\Gamma) \in \{S, G\} \quad \mathsf{isLeaf}(\Gamma)}{\Gamma \to \Gamma.\texttt{term}}$$

M-Cycle
$$\frac{\mathsf{focus}(\Gamma) = \mathsf{focus}(\Gamma') = \varphi \quad \neg\mathsf{isLeaf}(\Gamma')}{\Gamma \to^* (\Gamma'.\texttt{cycle}).\texttt{next}}$$

M-Full-Map
$$\frac{\mathsf{focus}(\Gamma) = \mathsf{focus}(\Gamma') = \varphi \quad \mathsf{isLeaf}(\Gamma')}{\Gamma \to^* (\Gamma'.\texttt{cycle}).\texttt{reset}}$$

Bits-Parse
$$\frac{\mathsf{focus}(\Gamma) = B_k \quad \Gamma \vdash i_k : (t_k, B_k) \quad t_k = 0}{\Gamma \to (\Gamma.\texttt{parse}).\texttt{next}}$$

Bits-Term
$$\frac{\mathsf{focus}(\Gamma) = B_k \quad \Gamma \vdash i_k : (t_k, B_k) \quad t_k = d_k}{\Gamma \to (\Gamma.\texttt{parse}).\texttt{term}}$$

Bits-Send-Inner
$$\frac{\mathsf{focus}(\Gamma) = B_k \quad \Gamma \vdash i_k : (t_k, v_k) \quad 0 < t_k < d_k}{((\Gamma \to \Gamma.\texttt{parse}).\texttt{send}).\texttt{next}}$$

Bits-Empty-Skip
$$\frac{\mathsf{focus}(\Gamma) = B_k \quad \Gamma \vdash i_k : (0, \varepsilon)}{\Gamma \to \Gamma.\texttt{next}}$$

Bits-Empty-Term
$$\frac{\mathsf{focus}(\Gamma) = B_k \quad \Gamma \vdash i_k : (t_k, \varepsilon) \quad t_k = d_k}{\Gamma \to (\Gamma.\texttt{parse}).\texttt{term}}$$

Figure 3.8: The complete ruleset for complexity level $C = 4$.

3.4. The Implementation Complexity Metric

The proposed formalism provides a concrete implementation of an interface, derived from its type and properties. This foundation allows us to quantify the logic cost required to implement a given interface specification, addressing the Tydi specification's goal of "balancing implementation complexity". While the original specification implies this balance through Complexity Levels, it lacks a method to substantiate it. By leveraging the formal semantics defined in this chapter, we can introduce a quantifiable *Implementation Complexity Metric* (ICM).

3.4.1. From Semantics to Automata

The derivation of the ICM relies on mapping the operational semantics to a hardware-equivalent model. This is supported by two key observations regarding the nature of the semantics:

1. Deterministic Intervals: The small-step transitions ($\to$) that occur between invocations of M-Cycle have no temporal equivalence in hardware. When synthesising an interface, these sequences are collapsed to a single execution.
2. State Abstraction: The sequence of multi-step transitions, $\Gamma \to^* \Gamma \ldots$ *do* have temporal equivalence and are fully determined by the input data instance. The state of the hardware interface between cycles is therefore equivalent to the state of the streaming context Γ when the focus is on to the root φ.

We can construct an Abstract Finite State Machine (AFSM) that accurately represents the control logic of the interface. The states of this machine correspond to the distinct configurations of the parse tree $Z_{\mathcal{N}}$ where the focus is on the root.

3.4.2. Defining the Metric

The AFSM is formally defined by a tuple $(S, \mathcal{I}, \delta)$:

- Abstract States (S): The set of reachable equivalence classes of contexts Γ where $\text{focus}(\Gamma) = \varphi$. These states are distinguished solely by the termination status of the nodes in the parse tree $Z_{\mathcal{N}}$.
- Transition Function (δ): The mapping $\delta : (S \times \mathcal{I}) \to S'$, defined by the application of M-Cycle and M-Full-Map, and the set of possible input elements $\mathcal{I}$.

We define the Implementation Complexity Metric (ICM) as the cardinality of the transition function δ, representing the number of distinct transitions between streaming contexts, reflecting the paths the hardware implementation must support: $ICM(\mathcal{N}, \mathcal{P}) = |\delta|$. It captures both the state space required to track the sequence status and the logic required to handle valid inputs, moving between these states.

3.4.3. Case Study: The Chat Message

To demonstrate the metric, we revisit the chat message example: $\mathcal{T}_{msg} := \texttt{Group}\langle\texttt{Bits}\langle 64\rangle, \texttt{Dim}\langle\texttt{Bits}\langle 8\rangle\rangle\rangle$. This is normalised to $\mathcal{N} := \texttt{Group}\langle\texttt{Bits}\langle 64\rangle, \texttt{Stream}\langle\texttt{Bits}\langle 8\rangle, d = 1\rangle\rangle$. We compare the ICM for property sets.

Case A: Core Property Set

First, we analyse the interface under $\mathcal{P}_{core}$, tracing the parsing of a message instance $msg := (\text{'}12 : 30\text{'}, \{\text{'}h\text{'}, \text{'}e\text{'}, \text{'}y\text{'}\})$.

- State (1) (Init): All nodes unterminated. Parsing ('h') transitions to State (2), as the timestamp $\texttt{Bits}\langle 64\rangle$ is terminated (it is a singleton), but the character stream is not.
- State (2) (Timestamp Terminated): The $\texttt{Bits}\langle 64\rangle$ node is skipped. Parsing ('e') results in a self-loop State (2) $\to$ (2). Parsing ('y'), which is the last element, transitions to State (3).
- State (3) (All Terminated): The parse tree is fully terminated. The M-Full-Map rule applies, resetting the tree and transitioning back to State (1).

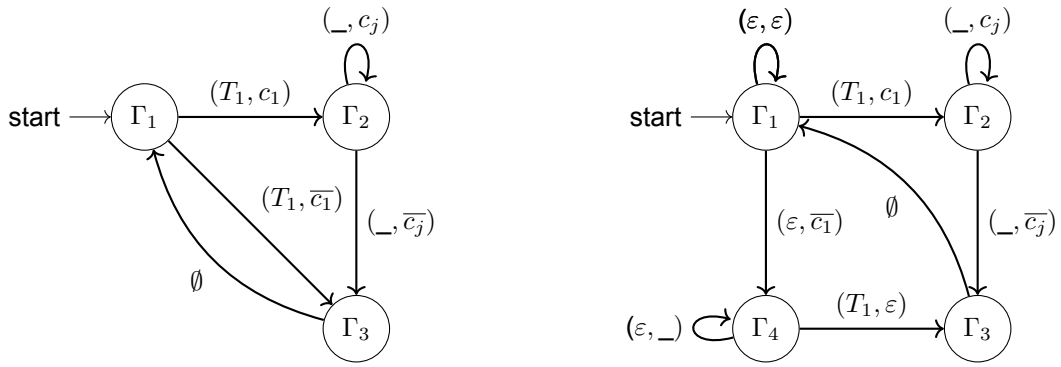
An additional transition exists, for a data instance where the message consists of a single character, which would terminate both the timestamp, and the character sequence in the first cycle, which corresponds to the transition from (1) $\to$ (3). This configuration yields 3 reachable states and 5 transitions shown in figure 3.9a, and gives $ICM = |\delta| = 5$.

Case B: Relaxed Cohesion

Next, we modify the specification by removing Input Cohesion ($\mathcal{P}_{core} \setminus \text{IP}_{\text{Coh}}$). This requires the inclusion of the Bits-Empty-Skip rule, allowing empty elements (ϵ) on internal streams, while still adhering to IP_{Coh} This relaxation introduces complexity:

1. Empty Cycles: If no data is available, the system self-loops at the initial state ($(1) \to (1)$).
2. Desynchronized Termination: Since input availability is not guaranteed, it is possible for the character stream to terminate *before* the timestamp is received. This necessitates a new state, State (4), where the stream is terminated but the timestamp is not.

This results in an expanded state machine with transitions for partial data availability (e.g., $(1) \to (4)$, $(4) \to (4)$ and $(4) \to (3)$). The resulting AFSM has 4 states and 8 distinct transitions, giving $ICM = 8$. Figure 3.9 compares the FSMs for the two interfaces. To reiterate, the transitions between states in the FSM, strictly correspond to the usage of M-Cycle and M-Full-Map, moving between distinct streaming contexts Γ, governed by the input elements. This comparison illustrates how relaxing input constraints strictly increases the implementation complexity of the interface logic, effectively quantifying the engineering trade-off between flexibility and resource cost.



(a) Core Property Set

(b) Relaxed Input Cohesion

Figure 3.9: Abstract finite state machines for $\mathcal{N} := \texttt{Group}\langle\texttt{Bits}\langle 64\rangle, \texttt{Stream}\langle\texttt{Bits}\langle 8\rangle, d = 1\rangle\rangle$. States correspond to distinct streaming contexts Γ with focus at the root. Each transition is labelled by a pair (T_i, c_j) denoting the ith timestamp element and the jth character element in a cycle; _ denotes an element is not considered since its stream is terminated, an overline indicates that the element terminates its stream, and ϵ denotes the empty element.

4

Examples and Empirical Validation

4.1. Full chat message parsing example

To illustrate the functionality of the formalism, I describe the experience from a hardware designer's perspective: from defining the top level type, to the actual transmission of data. Using the core properties + $\mathsf{SP}_{\mathsf{TermInner}}$, parsing the chat message example plus the single 32-bit emoji, typed as:

$$\mathcal{T}_{msg} := \texttt{Group}\langle\texttt{Bits}\langle 64\rangle, \texttt{Dim}\langle\texttt{Dim}\langle\texttt{Bits}\langle 8\rangle\rangle\rangle, \texttt{Bits}\langle 32\rangle\rangle$$

which is normalised to:

$$[\![\mathcal{T}_{msg}]\!] \mapsto \texttt{Group}\langle\texttt{Bits}\langle 96\rangle, \texttt{Stream}\langle\texttt{Bits}\langle 8\rangle, d = 2\rangle\rangle$$

The user specifies the number of lanes for the outer physical stream, and for the physical stream carrying the characters: $L := (1, 4)$. Now the streaming context can be constructed: $\Gamma(\mathcal{N}, L) \coloneqq (I, Z_{\mathcal{N}}, T_B)$ The input buffers I_1 and I_2 are initialized corresponding to the size of their elements (t_1, v_1) and (t_2, v_2), just like the transfer buffers T_{B_1} and T_{B_2}, where T_{B_1} has 1 lane: $\boxed{\varnothing}$ and T_{B_2} has 4: $\boxed{\varnothing}\boxed{\varnothing}\boxed{\varnothing}\boxed{\varnothing}$. A marked transfer buffer is visualized as having a thicker outline: $\boxed{\varnothing}$. The parse tree $Z_{\mathcal{N}}$ is constructed as shown in figure 2.3, with the focus on the root φ. Its inline representation is: $\boxed{\varphi}G(B_{96}, S(B_8))$. The element containing the timestamp + emoji corresponding to the first message is represented as $(1, \texttt{msg}_1)$, where $\texttt{msg}_1$ is the binary concatenation of the timestamp and the emoji. Since this is a singular element flowing on the outer stream, it always the 'last' element of a sequence of 1: $t_1 == d_1 == 1$.

Parsing example

Table 4.1 shows the first and last parsing iterations for the text message. I strongly recommend also viewing the more elaborate representation that can be found in Appendix B.

1) The initial state with empty buffers, the focus on the root and its children are not terminated, thus M-Cycle is invoked. This updates the input buffers, and shifts the focus to B_{96} according to post-order DFS traversal. 2) Since the focus is B_{96}, Bits-Term can be invoked, moving the element from I_1 to T_{B_1} and marking it: $\boxed{(1,\texttt{msg}_1)}$, terminating the node B_{96}, and moving to the next node, which is the leaf node B_8 corresponding to the char stream. 3) The input element $(0, \mathsf{s})$ has $t_2 = 0$ thus Bits-Parse maps it to the transferbuffer, and next is invoked, moving to S. 4) S has a non terminated child node B_8 thus we apply Struct-Enter moving the focus to G. 5) G has a non terminated child node S thus we apply Struct-Enter moving the focus to φ. 6) φ has a non terminated child node G thus we apply M-Cycle, sending the terminated transferbuffer T_{B_1} to streamspace, updating both inputs I_1 and I_2 and moving to the next non terminated node B_8. In iterations 7 to 10, $(0, \mathsf{h})$ is parsed and mapped to the T_{B_2}. 11) the input $(1, \mathsf{e})$ has $0 < t_k < d_k$, thus by inclusion of the $\mathsf{SP}_{\mathsf{TermInner}}$ property, the Bits-Send-Inner rule is invoked, parsing the element, marking T_{B_2} as terminal, and advancing the focus.

This results in the transfer $\boxed{(0,\mathsf{s})}\boxed{(0,\mathsf{h})}\boxed{(1,\mathsf{e})}\boxed{\varnothing}$ being sent at the next cycle (step 15). This sequence of parsing and sending repeats until the final element of the message is reached. 60) M-Cycle is invoked,

loading the final input element $(2, \mathsf{n})$ into I_2. The focus moves to B_8. 61) The focus is B_8, the input element has $t_k = d_k = 2$, thus Bits-Term is invoked, mapping $(2, \mathsf{n})$ to the transfer buffer, marking T_{B_2} as terminal, terminating the B_8 node, and advancing the focus to S. 62) The focus is now on S. Its only child, B_8, is terminated ($\mathsf{isLeaf}(\Gamma)$ is true), so Struct-Term is invoked, terminating S and advancing to G. 63) The focus is on G. All its children (B_{96} and S) are now terminated. Struct-Term is applied, terminating G and advancing the focus to the root, φ. 64) The focus is at φ. All its children (node G) are terminated, so $\mathsf{isLeaf}(\Gamma')$ is true. This triggers the M-Full-Map rule. The final transfer buffer T_{B_2} is sent to streamspace, the input buffer is updated (for the new message), and the parse tree is reset.

ID	$\mathbf{I_1}$	$\mathbf{I_2}$	$\mathbf{Z}_\mathcal{N}$	$\mathbf{T_{B_1}}$	$\mathbf{T_{B_2}}$	Streamspace output	Justification
1	∅	∅	$\boxed{\varphi}\, G\,(\,B_{96}, S\,(\,B_8\,))$	∅	∅ ∅ ∅ ∅	-	init
2	$(1, \mathtt{msg}_1)$	$(0, \mathsf{s})$	$\varphi\; G\,(\boxed{B_{96}}, S\,(\,B_8\,))$	∅	∅ ∅ ∅ ∅	-	M-Cycle
3	∅	$(0, \mathsf{s})$	$\varphi\; G\,(\cancel{B_{96}}, S\,(\boxed{B_8}))$	$(1, \mathtt{msg}_1)$	∅ ∅ ∅ ∅	-	Bits-Term
4	∅	∅	$\varphi\; G\,(\cancel{B_{96}}, \boxed{S\,(\,B_8\,)})$	$(1, \mathtt{msg}_1)$	(0,s) ∅ ∅ ∅	-	Bits-Parse
5	∅	∅	$\varphi\, \boxed{G\,(\cancel{B_{96}}, S\,(\,B_8\,))}$	$(1, \mathtt{msg}_1)$	(0,s) ∅ ∅ ∅	-	Struct-Enter
6	∅	∅	$\boxed{\varphi}\, G\,(\cancel{B_{96}}, S\,(\,B_8\,))$	$(1, \mathtt{msg}_1)$	(0,s) ∅ ∅ ∅	-	Struct-Enter
7	$(1, \mathtt{msg}_2)$	$(0, \mathsf{h})$	$\varphi\; G\,(\cancel{B_{96}}, S\,(\boxed{B_8}))$	∅	(0,s) ∅ ∅ ∅	$(1, \mathtt{msg}_1)$	M-Cycle
8	$(1, \mathtt{msg}_2)$	∅	$\varphi\; G\,(\cancel{B_{96}}, \boxed{S\,(\,B_8\,)})$	∅	(0,s) (0,h) ∅ ∅	-	Bits-Parse
9	$(1, \mathtt{msg}_2)$	∅	$\varphi\, \boxed{G\,(\cancel{B_{96}}, S\,(\,B_8\,))}$	∅	(0,s) (0,h) ∅ ∅	-	Struct-Enter
10	$(1, \mathtt{msg}_2)$	∅	$\boxed{\varphi}\, G\,(\cancel{B_{96}}, S\,(\,B_8\,))$	∅	(0,s) (0,h) ∅ ∅	-	Struct-Enter
11	$(1, \mathtt{msg}_2)$	$(1, \mathsf{e})$	$\varphi\; G\,(\cancel{B_{96}}, S\,(\boxed{B_8}))$	∅	(0,s) (0,h) ∅ ∅	-	M-Cycle
12	$(1, \mathtt{msg}_2)$	∅	$\varphi\; G\,(\cancel{B_{96}}, \boxed{S\,(\,B_8\,)})$	∅	(0,s) (0,h) (1,e) ∅	-	Bits-Send-Inner
13	$(1, \mathtt{msg}_2)$	∅	$\varphi\, \boxed{G\,(\cancel{B_{96}}, S\,(\,B_8\,))}$	∅	(0,s) (0,h) (1,e) ∅	-	Stuct-Enter
14	$(1, \mathtt{msg}_2)$	∅	$\boxed{\varphi}\, G\,(\cancel{B_{96}}, S\,(\,B_8\,))$	∅	(0,s) (0,h) (1,e) ∅	-	Stuct-Enter
15	$(1, \mathtt{msg}_2)$	$(0, i)$	$\varphi\; G\,(\cancel{B_{96}}, S\,(\boxed{B_8}))$	∅	∅ ∅ ∅ ∅	(0,s) (0,h) (1,e) ∅	M-Cycle
...	...	...	...	...	...	...	...
60	$(1, \mathtt{msg}_2)$	$(2, n)$	$\varphi\; G\,(\cancel{B_{96}}, S\,(\boxed{B_8}))$	∅	(0,h) (0,i) ∅ ∅	-	M-Cycle
61	$(1, \mathtt{msg}_2)$	∅	$\varphi\; G\,(\cancel{B_{96}}, \boxed{S\,(\cancel{B_8})})$	∅	(0,h) (0,i) (2,n) ∅	-	Bits-Term
62	$(1, \mathtt{msg}_2)$	∅	$\varphi\, \boxed{G\,(\cancel{B_{96}}, S\,(\cancel{B_8}))}$	∅	(0,h) (0,i) (2,n) ∅	-	Struct-Term
63	$(1, \mathtt{msg}_2)$	∅	$\boxed{\varphi}\, G\,(\cancel{B_{96}}, S\,(\cancel{B_8}))$	∅	(0,h) (0,i) (2,n) ∅	-	Struct-Term
64	$(1, \mathtt{msg}_2)$	(t_2, v_2)	$\varphi\; G\,(\boxed{B_{96}}, S\,(\,B_8\,))$	∅	∅ ∅ ∅ ∅	(0,h) (0,i) (2,n) ∅	M-Full-Map

Table 4.1: The first and last cycles of parsing the chat message example.

4.2. Empirical Validation with the TinyTydi Simulator

The TinyTydi simulator provides an executable version of the small-step operational semantics, realised in Python [48]. At startup the simulator normalises a random or user-supplied top-level type and instantiates the streaming context $\Gamma(\mathcal{N}, L)$, with a a parse tree $Z_\mathcal{N}$, a fixed set of per-stream transfer buffers and a tuple of input prefixes I. The Python implementation centres on the `StreamContext` abstraction: `_build_tree()` constructs the parsetree Γ and assigns leaf indices; `cycle(inputs)` injects `Element` values into the context; `parse()` maps an input `Element` into its leaf buffer and, when an element is terminal or a buffer is full, calls `_buffer_to_transfer()` to produce a `Transfer`; and `term()`, `next()` and `upnext()` implement the tree traversal an dnod termination. Notably, the traversal used in the simulator is not yet updated to use post-order DFS travesal, instead relying on an earlier manual implementation of a similar traversal, still using the now defunct `.upnext` operation.

After a type has been normalised, the simulator generates a random data instance of this normalised type, which consists of a set of elements and termination signals for each internal stream. The top-level `apply_semantics(sc)` routine directly encodes the rule hierarchy used in the thesis (Root, Stream, Group, NBits cases): it inspects the current focus, performs the rule-specific actions (terminate, parse, send, advance focus) and returns any generated `Transfer` objects. The simulator effectively operates by invoking applying `apply_semantics(sc)` until it halts, at which point the output transfers are compared with the original initialized data instance, ensuring no order constraints were violated. During each application of the semantics, the state of the streaming context is printed, alongside the streamspace representation, an example of which can be found in Appendix 4.1. The simulator can continuously generate random top-level types, normalise them, generate a data instance, and apply the semantics. This procedure provides a solid empirical foundation for the stability of the formalism, operating on much more complex type structures than would be feasible for a human operator.

5

Verification of the Formalism

This chapter will present a set of arguments, verifying that the proposed formalism implements interfaces without fault. I use the core as an example, subsequently detailing how this can be applied for arbitrary $\mathcal{R}$. First, I will show that the normalisation function always gives canonical form. This ensures that the guarantees provided by a normalised type in CF can be used by subsequent arguments. It is shown that the parse tree properly terminates for an arbitrary finite data instance. The subsequent argument shows that structural determinism holds for the core. It illustrates how matching each context state exhaustively and uniquely in a ruleset, ensures that the parsing logic never stalls. Finally, by showing that a ruleset strictly decreases a metric for progress, I show that on finite input, any ruleset will eventually terminate. Furthermore, it is shown that this termination will always happen properly, terminating the parsetree without resulting in disallowed tree states. Together, these arguments provide a foundation for the effectiveness of the proposed formalism.

5.1. Correctness of Normalisation

Canonical Form Definition

The set of Canonical Form normalised types $\mathrm{CF} \subseteq \mathcal{N}$ is defined:

$\texttt{Bits}\langle n \rangle \in \mathrm{CF}$	$\iff$	$n \in \mathbb{N}^+$.
$\texttt{Stream}\langle \mathcal{N}', d \rangle \in \mathrm{CF}$	$\iff$	$d \in \mathbb{N}^+$, $\mathcal{N}' \in \mathrm{CF}$, and $\mathcal{N}'$ is not of type $\texttt{Stream}\langle\cdot\rangle$.
$\texttt{Group}\langle \mathcal{N}_1, \ldots, \mathcal{N}_k \rangle \in \mathrm{CF}$	$\iff$	$k > 1$, for every $i \in \{1, \ldots, k\}$ we have $\mathcal{N}_i \in \mathrm{CF}$ and $\mathcal{N}_i \neq \texttt{Group}\langle\cdot\rangle$. At most one $\mathcal{N}_i$ may be of the form $\texttt{Bits}\langle n \rangle$, and if such a $\texttt{Bits}\langle n \rangle$ field exists it appears at $\mathcal{N}_1$.

Proof of Proper Normalisation

Theorem 1. *For any top-level type $\mathcal{T}$, its normalisation $[\![\mathcal{T}]\!]$ is in Canonical Form (CF).*

Proof. We proceed by structural induction on the definition of $\mathcal{T}$.

Base Case: $\mathcal{T} = \texttt{Bits}(n)$ By definition, $[\![\texttt{Bits}(n)]\!] = \texttt{Bits}(n)$. This satisfies CF condition (1).

Inductive Step 1: $\mathcal{T} = \texttt{Group}(\mathcal{T}_1, \ldots, \mathcal{T}_k)$ Assume the inductive hypothesis (IH) that for all $i \in \{1, \ldots, k\}$, $[\![\mathcal{T}_i]\!] = \mathcal{N}_i, \mathcal{N}_i \in CF$. We analyse $\mathcal{N} = \mathrm{coalesce}(\texttt{Group}(\mathcal{N}_1, \ldots, \mathcal{N}_k))$. The $\mathrm{coalesce}$ function executes three steps:

1. **Flattening:** Any $\mathcal{N}_i$ of the form $\texttt{Group}(\mathcal{N}'_1, \ldots, \mathcal{N}'_m)$ is replaced by its children. By the IH, $\mathcal{N}_i$ is in CF, so its children $(\mathcal{N}'_j)$ must satisfy the $\texttt{Group}$ conditions. Specifically, they are in CF and are not $\texttt{Group}$ types. Thus, the flattened list $L = (N_1, \ldots, N_p)$ consists only of $\texttt{Bits}$ and $\texttt{Stream}$ types, all in CF.

2. **Concatenation:** All $\texttt{Bits}(n_j)$ types in L are removed, summed into $\texttt{Bits}(n_{sum})$, and this single `Bits` type is placed at the head of the list. The list now contains at most one `Bits` type (at the head) and all original `Stream` types.
3. **Group Removal:** The function inspects the resulting $\texttt{Group}(N_1'', \ldots, N_q'')$.
   - **Case A:** The group contains a single field (i.e., $q = 1$). This field must be $\texttt{Bits}(n_{sum})$ or a single $\texttt{Stream}(\ldots)$. The function returns this single field $\mathcal{N} = N_1''$. By the IH (and construction), this field is in CF.
   - **Case B:** The group contains multiple fields (i.e., $q > 1$). The result is $\mathcal{N} = \texttt{Group}(N_1'', \ldots, N_q'')$. This type satisfies condition (3) of CF because:
     - All children N_j'' are in CF (from step 1).
     - No child is a `Group` (from step 1).
     - At most one child is `Bits` (from step 2).
     - If a `Bits` child exists, it is N_1'' (from step 2).

In all cases, the conditions hold, thus $\mathcal{N} \in CF$

Inductive Step 2: $\mathcal{T} = \texttt{Dim}(\mathcal{T}')$ Assume the (IH) that $[\![\mathcal{T}']\!] = \mathcal{N}', \mathcal{N}' \in CF$. We analyse $\mathcal{N} = [\![\texttt{Dim}(\mathcal{T}')]\!] = \text{lift}(\texttt{Stream}(\mathcal{N}', 1))$, and proceed by case analysis on the type of $\mathcal{N}'$.

1. $\mathcal{N}'$ *is not of type* `Stream`: By the IH, $\mathcal{N}'$ must be of type $\texttt{Bits}(n)$ or $\texttt{Group}(\cdot)$. By the definition of lift, $\mathcal{N}'$ is unchanged. This result satisfies condition (2) of CF, since $\mathcal{N}' \in CF$ and is not a `Stream`.
2. $\mathcal{N}'$ *is of type* `Stream`: By the IH, $\mathcal{N}'$ must be $\texttt{Stream}(\mathcal{N}'', d')$ where $\mathcal{N}'' \in CF$ thus $\mathcal{N}''$ is not of type `Stream`. By the definition of lift, $\mathcal{N} = \text{lift}(\texttt{Stream}(\mathcal{N}'', d'), 1) \mapsto \texttt{Stream}(\mathcal{N}'', d' + 1)$. This result satisfies condition (2) of CF since $\mathcal{N}'' \in CF$ and is not of type `Stream`.

In all cases, $\mathcal{N} \in CF$ holds. By the principle of structural induction, the property holds for all $\mathcal{T}$. □

5.2. Well-formed tree termination

Theorem 2 (Well-formed tree termination)**.** *For any context* Γ *with focus node* F*, if* Struct-Term *is applicable, then all nodes in the subtree rooted at* F *must be in a terminated state.*

Proof. The proof is by contradiction.

1. **Assumption:** Assume Struct-Term is applicable to F, but there exists at least one non-terminated node N in the subtree of F.
2. **Premise:** The applicability of Struct-Term on F requires $\text{isLeaf}(\Gamma)$ to be true. By definition, this means all direct children of F are in a terminated state.
3. **Implication:** Since N is in the subtree of F and its direct children are terminated, N cannot be F or a direct child of F. Thus, N must be a proper descendant of some direct child C_k.
4. **Ancestors:** Let $P_0, P_1, \ldots, P_m$ be the unique path of ancestors from $P_0 = N$ up to $P_m = C_k$.
5. **Inductive Argument:** We prove by induction on i that P_i must be non-terminated.
   - **Base Case ($i = 0$):** $P_0 = N$ is non-terminated by our initial assumption (1).
   - **Inductive Step:** Assume P_i (for $i < m$) is non-terminated. Its parent is P_{i+1}. For P_{i+1} to be marked as terminated, the Struct-Term rule must have been applied to it at some point (it must be a structural node, S or G, to have children). However, the premise for Struct-Term on P_{i+1} requires all of its children, including P_i, to be terminated. Since P_i is non-terminated, the premise isLeaf for P_{i+1} is false. Therefore, Struct-Term could not have been applied to P_{i+1}, and P_{i+1} must also be non-terminated.
6. **Contradiction:** By induction, $P_m = C_k$ must be non-terminated.

7. **Conclusion:** This leads to a contradiction. Step (2) states that C_k (a direct child of F) *is* terminated. Step (6) proves that C_k *must be* non-terminated. Our initial assumption (1) must be false. Therefore, no non-terminated node N can exist in the subtree of F when Struct-Term is applicable.

□

5.3. Proving the Core

This section will provide the structural determinism and progress proofs for the core. These proofs provide a rather verbose view of how structural determinism and progress of a ruleset are determined, functioning as a template for proofs of other rulesets.

5.3.1. Structural Determinism

Theorem 3 (Structural Determinism)**.** *For any valid streaming context* Γ *exactly one rule from the core ruleset is applicable.*

Proof. The proof proceeds by a case analysis on the type of the focus, $\mathsf{focus}(\Gamma)$.

Case 1: $\mathsf{focus}(\Gamma) \in \{\varphi, S, G\}$ **(Structural nodes)** The rules that can apply are Struct-Enter, Struct-Term, Cycle and Full-Map. Their premises are only differentiated by $\mathsf{isLeaf}(\Gamma)$. For all these rules, either $\mathsf{isLeaf}(\Gamma)$ or $\neg\mathsf{isLeaf}(\Gamma)$ must be true, thus they are mutually exclusive and exhaustive regarding the `Stream`, `Group` and φ nodes.

Case 2: $\mathsf{focus}(\Gamma) = B_k$ **(Primitive node)** The rules that can apply are Bits-Parse and Bits-Term. The applicability is differentiated by the input element $i_k : (t_k, v_k)$. By the Internal-Stream Property $\mathsf{IP}_{\mathsf{Coh}}$, $v_k \in \{B_k\}$, so $v_k \neq \varepsilon$. By $\mathsf{IP}_{\mathsf{ElemTerm}}$, $t_k = 0$ is only possible if $v_k \neq \varepsilon$. The applicability is therefore differentiated only on the value of t_k relative to the stream dimension d_k.

- The premise for Bits-Parse is $t_k < d_k$.
- The premise for Bits-Term is $t_k = d_k$. (Note: $t_k > d_k$ is ruled out by the context definition).

These two conditions, $t_k < d_k$ and $t_k = d_k$, are mutually exclusive and exhaustive for any $t_k \in \{0, \ldots, d_k\}$. Therefore, exactly one of these two rules is applicable.

Conclusion Since the case analysis covers all possible node types in the parse tree and the conditions within each case are mutually exclusive and exhaustive, exactly one rule is applicable for any valid context Γ. □

5.3.2. Progress and Termination

Lemma 1 (Finite Instance)**.** *For any normalised type* $\mathcal{N}$ *and corresponding set of input streams* I*, every well-formed data instance* $(\mathcal{N}, I)$ *is finite. In particular, the parse tree* $Z_{\mathcal{N}}$ *induced by* $\mathcal{N}$ *has finitely many nodes, and each complete input stream* $I_k \in I$ *has finite elements.*

Theorem 4 (Progress)**.** *For any well formed data instance* $(\mathcal{N}, I)$*, every sequence of small-step transitions* $\Gamma_0 \to \Gamma_1 \to \ldots$ *is finite, terminating when* Full-Map *is applicable.*

Proof. To prove termination, we define a metric $M(\Gamma)$ that maps each context to a value in a well-founded set.

1. **Well-founded Set:** We use $(\mathbb{N} \times \mathbb{N}, <_{\text{lex}})$, the set of pairs of natural numbers ordered lexicographically: $(n, s) <_{\text{lex}} (n', s')$ iff $n < n'$ or $(n = n'$ and $s < s')$ This relation is well-founded.
2. **Metric Definition:** Define $M(\Gamma) \coloneqq (N, S)$, where:
   - N the number of non-terminated nodes in the parse tree $Z_{\mathcal{N}}$;
   - S the sum of the remaining lengths of all input streams in I.

   By Lemma 1, both N and S are finite, hence $M(\Gamma) \in \mathbb{N} \times \mathbb{N}$.

3. **Transition Analysis:** We show that every rule application either strictly decreases M or, after finitely many steps, leads to a rule that does:
   - **Struct-Term, Bits-Term:** Each application marks one node as terminated, so $N' = N - 1$. Since $(N-1, S') <_{\text{lex}} (N, S)$ for any S', the metric strictly decreases.
   - **Bits-Parse:** Consumes one element from an input stream, so $S' = S - 1$ while $N' = N$. Thus $(N, S-1) <_{\text{lex}} (N, S)$ again, the metric strictly decreases.
   - **Struct-Enter:** Changes the focus but not N or S, so $M(\Gamma') = M(\Gamma)$. However, since $Z_{\mathcal{N}}$ is a finite, acyclic tree (by Lemma 1), repeated Struct-Enter steps can only traverse finitely many nodes before reaching a leaf or a fully-terminated structure, where one of the above rules applies and decreases M.

Conclusion. Every transition either strictly decreases $M(\Gamma)$ or occurs in a finite sequence that culminates in a strict decrease. Since $<_{\text{lex}}$ is well-founded, no infinite sequence of transitions exists. Therefore, all derivations must terminate in a state where no rule is applicable. This terminal state occurs when all inputs are consumed ($S = 0$) and all nodes are terminated ($N = 1$, for the root φ). By the proof of Determinism, this state satisfies $\mathsf{focus}(\Gamma) = \varphi$ and $\mathsf{isLeaf}(\Gamma)$, the premises for applying Full-Map. □

5.4. Connecting the results

It is not obvious why the proofs above together ensure that $\mathcal{R}_{core}$ always properly parses a data instance of arbitrary type $\mathcal{N}$. Firstly, proper normalisation ensures that the normalisation function always provides types in Canonical Form, with its corresponding guarantees. Then, the well-formed tree termination shows that for any interface specification, regardless $\mathcal{T}$ or $\mathcal{P}$, and regardless of data instance, the tree will always terminate from leaf to root, never leaving nodes inaccessible. This also enables the reduction required for the ICM equivalence classes, of contexts where the root is in focus. The subsequent two proofs are related to the core specifically, and really operate more like “checks“, ensuring that $\mathcal{R}_{core}$ is structurally deterministic, and the progress property holds. This is achieved by 1) ensuring the transitions are exhaustive and unique w.r.t. the possible context states, and secondly 2) applying the transitions will always decrease the size of the input, and/or decrease the number of non-terminated nodes of the parsetree. Since the ruleset is structurally deterministic, the parsing will never halt, and since the tree is guaranteed to terminate properly (on finite input), showing that the number of active node decreases is sufficient.

5.4.1. Proving other property-sets

The first two proofs are valid for any interface specification, but the latter, structural determinism and progress, are specific to the core. These proofs are rather verbose, and for other property-sets it is possible (and desirable) to show they implement structural determinism and ensure progress in much shorter proofs, using the above as templates.

Analysing Complexity level 4

Lets look at the property-set for complexity level 4, and determine if structural determinism and progress hold.

Structural Determinism Proof Outline. For the case of structural determinism we use case analysis on $\mathsf{focus}(\Gamma)$. The structural nodes are parsed by the default semantics, which is verified for the core, and since all other semantics only considers the case $\mathsf{focus}(\Gamma) = B_k$, these results can be reused, and we only have to verify structural determinism for the case where the focus is on bits. Since $\mathsf{IP}_{\mathsf{Coh}} \notin \mathcal{P}_{C4}$, we know that for any input stream I_k the set of input elements is: $\{B_k, \varepsilon\}$. For $\Gamma \vdash i_k : (t_k, B_k)$, Bits-Parse, Bits-Term and Bits-Send-Inner are applicable, depending on the value of t_k Bits-Parse accounts for the case $t_k = 0$, Bits-Term accounts for the case $t_k = d_k$, and Bits-Send-Inner accounts for the case $0 < t_k < d_k$, which are mutually exclusive, and cover all possible values of (t_k, B_k). For $\Gamma \vdash i_k : (t_k, \varepsilon)$, Bits-Empty-Skip, Bits-Send-Inner and Bits-Empty-Term are applicable, depending on the value of t_k. Bits-Empty-Skip accounts for the case $t_k = 0$, Bits-Empty-Term accounts for the case $t_k = d_k$, and Bits-Send-Inner accounts for the case $0 < t_k < d_k$, which are mutually exclusive, and cover all possible

values of (t_k, ε). Therefore, for any valid context, exactly one rule is applicable, and $\mathcal{R}_{C4}$ is structurally deterministic. □

Progress Proof Outline. In order to prove that progress holds for $\mathcal{R}_{C4}$, we use the metric $M(\Gamma) \coloneqq (N, S)$ over the well-founded set $(\mathbb{N} \times \mathbb{N}, <_{lex})$, where N is the count of non-terminated nodes and S is the total size of input prefixes. Since every transition (or finite sequence of non-decreasing transitions) strictly decreases $M(\Gamma)$, the system must reach a terminal state (Full-Map) in a finite number of steps for any finite input. Using the results from the core, we know that Struct-Term and Bits-Term strictly decrease N, Bits-Parse strictly decreases S while N is constant. Bits-Empty-Term strictly decreases N, and Bits-Send-Inner decreases S while N stays constant. The odd one out is Bits-Empty-Skip, which does not decrease S or N. However, by lemma 1, finite instance enforces that an input stream I_k has finite number of ε entries. Therefore, the repeated invocation of Bits-Empty-Skip and M-Cycle, ensures that eventually the input element will no longer be empty, $\Gamma \vdash i_k : (t_k, B_k)$, and a metric decreasing rule will apply. Struct-Enter leads in a finite number of steps to a leaf node where another (metric-decreasing) rule applies. Since every transition (or finite sequence of non-decreasing transitions) strictly decreases $M(\Gamma)$, the system must reach a terminal state (Full-Map) in a finite number of steps for any finite input. □

6

The Union Type: Handling Variant Data

This chapter will discuss the the inclusion of the Union type and the specific challenges it presents. I first intend to define a *what* is considered semantically ambiguous when working with the Union, defining the Canonical Form and the required normalisation function. Then, the semantics will need to be expanded to include transitions for the Union, utilising the structure enforced by the normalisation function, ensuring deterministic streamspace mappings of the elements corresponding to sequences in Unions.

6.1. Design considerations for variant types in hardware

The Union is fundamentally different from the other types in the Tydi type system since it carries inherit unknowns. It is defined as a variant type: Union$\langle T_1, T_2, \ldots, T_n \rangle$ which is instantiated with a tag t, indicating which field is active: $I(T \in \{T_1, T_2, \ldots, T_n\}, t \in (1, \ldots, n))$. Since the tag determines which field is active at runtime, we cannot determine its type statically. The complexity it introduces lives on two levels, how to normalise a Union such that its canonical form gives useful structure for the semantics to interact with, and how to implement this semantics without invalidating the properties that enable the verification of the parsing. This section will illustrate the subtle design considerations regarding normalisation and semantics, leading to the eventual solution presented in the subsequent section.

Design considerations for integrating the Union type

Since the normalisation and the resulting canonical form have strong implications for the semantics, this is the first procedure that needs to be defined. Restating the goal of normalisation: ensuring the size of the elements flowing over the physical stream is made explicit, while also enforcing unambiguous structures for the semantics to interact with. Let's start small with a Union that only contains primitives: $\texttt{Union}\langle \texttt{Bits}_1\langle\cdot\rangle, \texttt{Bits}_2\langle\cdot\rangle, \ldots, \texttt{Bits}_n\langle\cdot\rangle\rangle$ Since only one of the fields is active, we determine the size of the element on the physical stream by taking the largest possible primitive in the union. The tag field can be made explicit by encasing the union in a group, together with a primitive representing the tag. The normalisation would then be $[\![\texttt{Union}\langle\texttt{Bits}\langle 2\rangle, \texttt{Bits}\langle 3\rangle\rangle]\!] \mapsto \texttt{Group}\langle\texttt{Bits}\langle 1\rangle, \texttt{Union}\langle\texttt{Bits}\langle 3\rangle\rangle\rangle$. The tag is of size 1, since it can either select $\texttt{Bits}\langle 2\rangle$ or $\texttt{Bits}\langle 3\rangle$. Even for unions that only contain primitives, this normalisation does not to adhere to the requirements I proposed for canonical form of Groups. Since the tag field and the data inside the Union correspond to the same physical stream, the proposed normalisation is not sufficient. Consequently, it requires an additional step, either changing the way the Union node is parsed by the semantics, making the Union node a data-carrying node like the Bits node, possibly normalising it as: $[\![\texttt{Union}\langle\cdot\rangle]\!] \mapsto \texttt{Union}_{tag}\langle\cdot\rangle$, or by defining method that deterministically combines the primitives in a union, together with its tag data. The latter solution seems more suitable for the integration with the Tydi formalism, since it adheres to the structure that the semantics expects. Furthermore, concatenating the tag field with the data field makes the size of

the element lanes explicit, in contrast with the alternative, which obscures the width of the lanes of a PhysicalStream, by separating the tag field. This would result in the above example to be normalised as such: $[\![\texttt{Union}\langle\texttt{Bits}\langle 2\rangle, \texttt{Bits}\langle 3\rangle\rangle]\!] \mapsto \texttt{Bits}\langle 4\rangle$. The downside of this approach is that the Union type may be completely obscured after normalisation, leaving the logic in the receiver to decode the data field. It is, however, important to restate that the aim of the normalisation is to enforce semantic unambiguity in the normalised type, which may not necessarily overlap with a 'human intuitive' type. In fact, the above example matches the way synthesis would generate the physical streams, with element lanes of width 4, since the specification explicitly states that it does not specify how the bits should be interpreted: such interpretation is explicitly out of scope.

Expanding the normalisation

Now, within the Tydi spec it is entirely possible to have nested stream structures within a Union, which already occurs when mapping a nullable string to Tydi: string is a sequence, and if it can be null, the Union needs to be utilized since it is now a variant type containing the singleton, or the sequence of characters. If we imagine the parsetree for a type that contains unions with nested streams, a problem for its semantics becomes apparent. For example, if we extend the chat message by allowing it to be encoded in 8- or 16-bit characters: $\texttt{Group}\langle\texttt{Bits}\langle 64\rangle, \texttt{Union}\langle\texttt{Dim}\langle\texttt{Bits}\langle 8\rangle\rangle, \texttt{Dim}\langle\texttt{Bits}\langle 16\rangle\rangle\rangle\rangle$ When initialising the streaming context, what should the parse tree look like? The previously proposed style of normalisation would proceed as follows:

$$[\![\texttt{Group}\langle\texttt{Bits}\langle 64\rangle, \texttt{Union}\langle\texttt{Dim}\langle\texttt{Bits}\langle 8\rangle\rangle, \texttt{Dim}\langle\texttt{Bits}\langle 16\rangle\rangle\rangle\rangle]\!]$$

1. Each `Dim` type is normalised into an explicit `Stream` node:

$$\texttt{Group}\langle\texttt{Bits}\langle 64\rangle, \texttt{Union}\langle\texttt{Stream}\langle\texttt{Bits}\langle 8\rangle, d=1\rangle, \texttt{Stream}\langle\texttt{Bits}\langle 16\rangle, d=1\rangle\rangle\rangle$$

2. Since the `Union` contains two alternatives, a tag field of width 1 is introduced to signal the active field. In accordance with earlier discussion, the tag is made explicit by enclosing the union in a `Group` together with the tag primitive:

$$\texttt{Group}\langle\texttt{Bits}\langle 64\rangle, \texttt{Group}\langle\texttt{Bits}\langle 1\rangle, \texttt{Union}\langle\texttt{Stream}\langle\texttt{Bits}\langle 8\rangle, d=1\rangle, \texttt{Stream}\langle\texttt{Bits}\langle 16\rangle, d=1\rangle\rangle\rangle\rangle$$

3. The outer `Group` is then normalised by eliminating the nested `Group`. Since the $\texttt{Bits}\langle 64\rangle$ and $\texttt{Bits}\langle 1\rangle$ primitives correspond to the same physical stream, they are merged into a single primitive of width 65:

$$\texttt{Group}\langle\texttt{Bits}\langle 65\rangle, \texttt{Union}\langle\texttt{Stream}\langle\texttt{Bits}\langle 8\rangle, d=1\rangle, \texttt{Stream}\langle\texttt{Bits}\langle 16\rangle, d=1\rangle\rangle\rangle$$

The resulting normalised form of the type would correspond to the parse tree shown in figure 6.1. According to the existing formalism, the streaming context should be initialised with all tree nodes non-terminated. How would the current style of semantics interact with this tree?

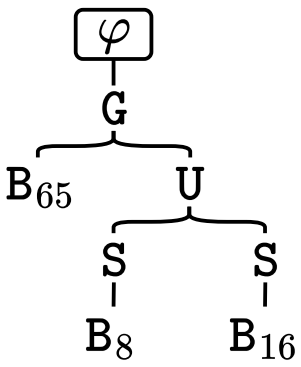


Figure 6.1: The parse tree for the Union example, with 2 encodings of chat messages

Let's assume the property-set of the interface is $\mathcal{P}'_{core}$, a hypothetical version of the core that supports the Union. First, the $\texttt{Bits}\langle 65\rangle$ node is visited, which contains the 64-bit timestamp plus the 1-bit tag information for the Union. This highlights the importance of the normalisation procedure of groups, moving the primitive to the first index of the group, since it needs to be certain that the tag data is parsed before evaluating the Union node. Furthermore, since input cohesion is guaranteed by the core, the 65-bit element is available on the internal streams, and can be parsed. Say the value of the tag is '1', indicating the second field of the Union is active for this data instance, containing the 16-bit character stream. However, post order DFS traversal would visit the node corresponding to the 8-bit character stream first, since it is the next non-terminal node. Furthermore, if input cohesion is not guaranteed, the tag field may not be available, and the semantics continues traversing into the subtrees of union1 and union2, which would violate TrueOrder, since it may be possible to transmit elements in a different order than they existed in the data instance. This highlights that the parsing of the union tag field, needs to have consequences for the semantics, only enabling the subtree corresponding to the selected field of the union.

During traversal subtree activation

Based on the proposed union normalisation a question arises: How can the semantics differentiate between a regular $\texttt{B}_n$ node, and one that contains tag information from the Union?

The following is proposed: for any bits field, when it is parsed, also enable all union nodes that are in its level of the hierarchy. For $\texttt{S}\langle\texttt{B.}\rangle$, if the $\texttt{B.}$ node is parsed, the semantics that parses it will also activate the unions at its current level (regardless of their existence), so in this example none exist and parsing would happen as usual. Similarly, for $\texttt{G}\langle\texttt{B.},\ \texttt{S}_1\langle\cdot\rangle,\ \texttt{S}_2\langle\cdot\rangle\rangle$, it would achieve nothing, and less obviously, for

$$\mathcal{N} := \texttt{G}\langle\texttt{S}_1\langle\texttt{G}_1\langle\texttt{B}_1,\ \texttt{U}_1\langle\cdot\rangle\rangle\rangle,\ \texttt{S}_2\langle\texttt{G}_2\langle\texttt{B}_2,\ \texttt{U}_2\langle\cdot\rangle\rangle\rangle\rangle,$$

parsing $\texttt{B}_1$ would only activate $\texttt{U}_1$, not $\texttt{U}_2$, even though $\texttt{U}_2$ may *appear* on the same level as $\texttt{B}_1$ (removed as many steps from the root), it is in a separate subtree, so not in the same hierarchy, therefore not activated.

If we have

$$\mathcal{N} := \texttt{G}\langle\texttt{B.},\ \texttt{S}\langle\cdot\rangle,\ \texttt{U}_1\langle\cdot\rangle,\ \texttt{U}_2\langle\cdot\rangle\rangle,$$

the union fields, and their subtrees, would be initialized as terminated, however, when Bits-Parse parses the $\texttt{B.}$ field, it activates the Union nodes. Subsequently, when traversing the parse tree, first the $\texttt{S}\langle\cdot\rangle$ subtree is visited, then we visit $\texttt{U}_1$ (not its subtree since *only* the union node is activated), and the semantics for parsing the union node reads the tag from $\texttt{B.}$ and activates its corresponding subtree. Working through an example: 1) visit $\texttt{B.}$, it activates the two union nodes, subsequently visiting the stream subtree, 2) we visit the first union node $\texttt{U}_1\langle\cdot\rangle$, it reads the tag field and activates its corresponding subtree, 3) the subtree is now active so `.next` will traverse to the bottom of the subtree, according to post-order DFS, 4) we parse some primitive in a stream at the bottom of the union subtree (ensured to exist by CF for the union), the element is terminal, the bits node is terminated and post-order DFS iterates up the subtree, the stream node is terminated etc., traversing up until we hit the union node again, 5) now the union node is again an effective leaf node, but it is impossible for the semantics differentiate between state (2) and the current state. The proposed solution would evaluate the tag again, and activate the subtree.

This leaves us with a few fundamental considerations to make. What is semantically unambiguous when considering the Union? Consider the proposed normalisation: making the tag field explicit by enclosing the union in a group and adding a primitive corresponding to its tag. When normalising the structure further, this tag field primitive will be concatenated with other, data carrying primitives, that correspond to the same physical stream, creating the $\texttt{B}_{65}$ field for the chat message example. This seems to introduce ambiguity again, since it is now no longer obvious whether $\texttt{B}_{65}$ is just a 65-bit wide data element, or if it contains tag information. In the problem statement I articulated a similar problem: “When the existence of a `G` type does not provide strong guarantees regarding how its fields are interleaved / concatenated over physical streams, any formalism interacting with the type must account for the larger structure by also evaluating the fields of the group.’ The proposed Union normalisation seems to cause ambiguity regarding the semantics of the Bits field; its existence no longer provides strong guarantees regarding how data is transmitted, requiring the formalism to account for the larger structure by also evaluating nodes of the parse tree around the Bits field, to potentially activate any Unions. The parsing of the tag field should therefore not be done by the semantics for the primitive, but for the union, since its existence requires the tag field to exist in the parent stream.

Maintaining structural determinism

Say we implement normalisation as proposed initially: the union is encapsulated by a group, with a primitive corresponding to its tag field, and the primitives in the fields of the union are subsequently incorporated according to natural ordering. When initialising the streaming context, in the parse tree the subtrees of the union are terminated, but the union nodes are not. During evaluation, the bits field is parsed as normal by the semantics of the interface, using Bits-Parse or Bits-Term When the union node is in focus, and it is a leaf node (its subtrees are terminated), the semantics evaluate the index of the input buffer that corresponds to the tag field. (this requires the index of the tag to be deterministic, which is ensured by the natural ordering of the normalisation) If the data is valid, meaning the primitive carrying the tag was parsed during the current tree traversal, the semantics activates the subtree corresponding to the selected tag.

A problem remains: the Union's subtree corresponding to the tag needs to be activated, before evaluating the Union node, otherwise the same behaviour occurs where the Union node can be visited twice during the same traversal, while being unable to differentiate between the two.

There were various attempts to create a distinction between primitive nodes with, and without tag data, like having $\texttt{B}_{\cdot}$ and $\texttt{B}_{\cdot}^{tag}$, or utilising a value dependent type theory, where the semantics can adapt based on the value of its instance. All these ideas are fundamentally flawed: they violate structural determinism from the perspective of the cycle.

If we want to incorporate the Union with the existing formalism, the exact route taken while iterating over the parse tree, *cannot* depend on a value that is evaluated halfway through the traversal. No matter the formalism or type-theoretic approach, if we want to maintain the property that the small-step semantics can be reduced to the multi-step semantics, all the transitions between each invocation of M-Cycle need to be fully deterministic. Therefore, when dealing with dependent sum type such as the union, the *only* valid solution is to defer the evaluation of its subtree, postponing the parsing of its corresponding physical stream to the next tree traversal / cycle. This can be implemented by using the tag to mark the subtree that should be activated in the subsequent cycle, similar to how the `.send` context operation marks the transfer buffer that should be transmitted. For the chat message example this would mean that, if the tag field selects the 12-bit primitive, it would be transmitted in the same cycle, and no subtree would be activated, otherwise, if one of the two character streams is selected by the tag field, only in the subsequent cycle would the first character of the message be parsed.

To reiterate this idea: at the beginning of each cycle, an *itinerary* is created; the post-order DFS traversal of the tree, with all the terminated nodes removed. In essence this already happens for the existing formalism excluding the Union, but this is not mentioned since without the union, the itinerary could not change based on the value of an element. When a subtree of a Union is activated, this is not reflected in the itinerary of the current tree traversal, since it was created at the start of the cycle, before any elements were evaluated. Therefore invoking `.next` will continue as if the subtree was still terminated. For the chat message + 12-bit example, the initial itinerary would be:

Cycle	Itinerary	Judgement / Argumentation
1	$[\texttt{B}_{78},\ \texttt{U},\ \texttt{G},\ \texttt{Root}]$	$\texttt{B}_{78}$ is parsed and the tag selects $\texttt{S}_1$, carrying the 8-bit characters.
2	$[\texttt{B}_{8},\ \texttt{S}_1,\ \texttt{U},\ \texttt{G},\ \texttt{Root}]$	$\texttt{B}_{78}$ is not visited since it was terminated during the first traversal. The first 8-bit character is parsed.

Maintaining tag-data synchronicity

In the previous example, everything went according to plan, since for

$$\mathcal{N} := \texttt{G}\langle \texttt{B}_{78},\ \texttt{U}\langle \texttt{S}_1\langle \texttt{B}_8, d=1\rangle,\ \texttt{S}_2\langle \texttt{B}_{16}, d=1\rangle\rangle\rangle$$

it is ensured that $\texttt{B}_{78}$ always exists *once* per instance. This is the case, since the root represents the outer sequence; a stream with $d = 0$, ensuring that every element $i_k : (v_k, t_k)$ on its physical stream is parsed as if it were the last of the sequence. What if this type is encapsulated in another stream, in which case the bits node $\texttt{B}_{78}$ containing the tag field, is not necessarily terminated on the first iteration? This gives:

$$\mathcal{N} := \texttt{S}_0\langle \texttt{G}\langle \texttt{B}_{78},\ \texttt{U}\langle \texttt{S}_1\langle \texttt{B}_8, d=1\rangle,\ \texttt{S}_2\langle \texttt{B}_{16}, d=1\rangle\rangle\rangle,\ d=1\rangle.$$

This could lead to the following problem:

Cycle	Itinerary	Judgement / Argumentation
1	$\texttt{B}_{78} \to \cdots$	$\texttt{B}_{78}$ is parsed, tag selects $\texttt{S}_1$. Node not terminated due to $\texttt{S}_0$. Subtree for $\texttt{S}_1$ is marked.
2	$\texttt{B}_{78} \to \texttt{B}_8 \to \cdots$	$\texttt{B}_{78}$ parsed again, its tag selects $\texttt{S}_2$. $\texttt{B}_8$ parsed non-terminal. Ambiguity arises whether to activate $\texttt{S}_2$.
3a	$\texttt{B}_{78} \to \texttt{B}_8 \to \texttt{B}_{16} \to \cdots$	Simultaneous activations lead to illegal interleavings of S_1 and S_2 and loss of tag information.
4a	$\cdots$	Union subtrees terminate, but tag information from cycle 3a is lost.

The above example illustrates a new problem: the information for activating $\mathtt{S}_1$ a second time, contained in the third element of $\mathtt{B}_{78}$, was lost due to the fact that its physical stream can transmit data before the union's subtree has terminated. One way to solve this is by enforcing back-pressure in the specification; disallowing the receiver to accept transfers from $\mathtt{B}_{78}$ until the physical streams corresponding to the union have finished. This is not desirable, since it disregards the notion of implementing the receiver as the dual of the sender interface. Additionally, the intention is to implement a semantics that *always* produces a valid, unambiguous streamspace mapping, regardless of the order in which the receiving interface accepts the transfers.

Sending the tag over the parent stream, while ensuring that the streamspace mappings of the physical streams corresponding to the parent stream and the union do not contain illegal interleavings, is rather complicated. In the example without the outer stream, the root ensured that $\mathtt{B}_{78}$ was sent once. We could attempt to replicate this, by explicitly synthesising the union, its tag plus the maximum of its fields, as a separate physical stream of $d = 0$. This would normalise the expanded chat message example

$$\mathcal{T} := \mathtt{D}\langle \mathtt{G}\langle \mathtt{B}_{64},\ \mathtt{U}\langle \mathtt{B}_{12},\ \mathtt{D}\langle \mathtt{B}_{8}\rangle,\ \mathtt{D}\langle \mathtt{B}_{16}\rangle\rangle\rangle$$

to

$$\mathcal{N} := \mathtt{S}_0\langle \mathtt{G}_0\langle \mathtt{B}_{64},\ \mathtt{S}_1\langle \mathtt{G}_1\langle \mathtt{B}_{14},\ \mathtt{U}\langle \mathtt{S}_2\langle \mathtt{B}_{8}, d=1\rangle,\ \mathtt{S}_3\langle \mathtt{B}_{16}, d=1\rangle\rangle\rangle,\ d=0\rangle\rangle,\ d=1\rangle.$$

The stream $\mathtt{S}_1$ with $d = 0$ functions as the `NEW` shorthand that the original specification proposes, where a separate physical stream is created at the same dimensionality as the parent stream. It ensures that the tag information is always sent together with the physical stream corresponding to the union's subtree. Let's check how this would behave:

Cycle	Itinerary	Judgement / Argumentation
1	$\mathtt{B}_{64} \rightarrow \mathtt{B}_{14} \rightarrow \cdots$	$\mathtt{B}_{64}$ non-terminal. $\mathtt{B}_{14}$ terminal, tag selects $\mathtt{S}_2$.
2	$\mathtt{B}_{64} \rightarrow \mathtt{B}_{8} \rightarrow \cdots$	First 8-bit element parsed, non-terminal.
3	$\mathtt{B}_{64} \rightarrow \mathtt{B}_{8} \rightarrow \cdots$	$\mathtt{B}_{64}$ terminal. $\mathtt{B}_{8}$ non-terminal.
4	$\mathtt{B}_{8} \rightarrow \cdots$	$\mathtt{B}_{8}$ terminal. Union terminates. Full-map reached.

The solution avoids activating multiple subtrees of the union, and the loss of tag data, since $\mathtt{B}_{14}$ is transmitted once, and the data corresponding to the selected subtree is also transmitted once. However, a different problem arises, since the `G` type enforces that all its fields are transmitted together, and we've seen the $\mathtt{B}_{64}$ being transmitted three times, while the second field of $\mathtt{G}_0$, namely $\mathtt{S}_1$, has only been terminated once. Full-Map can only be applicable after the complete data instance has been parsed, which is not the case This imposes a strict synchronisation requirement for the separate physical stream that we've created for the Union data. Additionally, if the union, and thereby its tag information, is part of a larger sequence, the union must fully terminate before any data corresponding to the larger sequence can be sent. Making the example even more complicated:

$$\mathcal{N} := \mathtt{S}_0\langle \mathtt{G}_0\langle \mathtt{B}_{64},\ \mathtt{S}_1\langle \mathtt{G}_1\langle \mathtt{B}_{14},\ \mathtt{U}\langle \mathtt{S}_2\langle \mathtt{B}_{8}, d=1\rangle,\ \mathtt{S}_3\langle \mathtt{B}_{16}, d=1\rangle\rangle\rangle,\ d=0\rangle,\ \mathtt{S}_4\langle\cdot\rangle\rangle,\ d=1\rangle,$$

Even though the outer sequence $\mathtt{S}_0$ has dimension 1, exactly one instance of $\mathtt{B}_{64}$ and $\mathtt{S}_4\langle\cdot\rangle$ may be sent over their respective physical streams before $\mathtt{S}_1$ has finished. The inner stream, enclosing the union as a sequence of $d = 0$, cannot enforce this synchronisation for its parent.

Deferred Parsing Commit

From the various attempts at normalisation and integration in the semantics, a few important observations are made: 1) The size of the physical stream elements should be explicit in the normalised type, including the tag for a union, 2) the union must affect the parse tree, initialising its subtrees as terminated, activating the required subtree based on tag evaluation, 3) to maintain structural determinism, the itinerary of the current tree traversal cannot depend on an evaluation within the current traversal, and 4) the state of the union should enforce strict tag-data synchronicity between its parent stream and the streams in its subtrees, avoiding the loss of tag data.

The core idea is to make the parsing transitions *non-committal* within a cycle: parsing a parent-stream primitive marks the corresponding input buffer as *ready to parse*, but the effect of that parse on the streaming context is deferred until the next time M-Cycle is invoked. Concretely, this means that

Γ.parse does not consume an input element during the tree traversal; it just marks the element as 'to parse'. This mirrors the existing .send context operation, and better reflects the hardware reality since traversing the parse tree has no real temporal equivalence. To integrate this behaviour with unions I add a small piece of internal state to each Union node: a *stored tag record* containing

$$\texttt{stored\_tag} := (\texttt{tag\_value},\ \texttt{term\_flag})$$

where term_flag contains the termination data t_k from the input element i_k, for the parent stream k that the union corresponds to. It indicates whether the parent stream was terminated in the cycle that supplied the tag. The invariant is:

> The union may only *refresh* its stored_tag when it has no active subtrees; whenever the union has active subtrees those subtrees must terminate before the union considers a new tag.

This design results in three behaviours, requiring two new context operations:

1. **Read-and-store (deferred activation).** When the stored termination data indicates the parent stream is active, and the Union currently has no active subtrees, the Union reads the tag *and the termination value* from the parent stream and stores them. The semantics does not enter the selected subtree during the same cycle; instead it invokes .activate to mark the subtree for activation in the subsequent cycle.
2. **Undo / unparse when busy.** If the parent primitive is parsed while the Union still has active subtrees, the semantics must invoke .unparse on the parent. This effectively implements back pressure on the internal streams of the component.
3. **Terminal-tag handling.** If the stored_tag indicates the parent stream is terminated and the Union has no active subtrees, the Union node is terminated, erasing the stored record and ensuring the union and its parent stream remain synchronized.

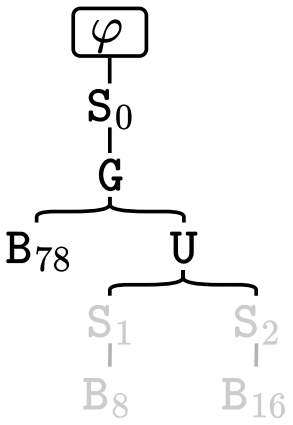


Figure 6.2: The tree representing a sequence of variable encoded chat messages, initialized with terminated union subtrees.

Let's consider the sequence of chat messages, with 2 encodings and the 12-bit 'neither' field. Its parsetree is shown in figure 6.2. The transitions in table 6.1 show that the parsing is effectively undone on the parent stream, and that the itinerary can be determined at the cycle, ensuring deterministic parsing and maintaining tag-data synchronicity.

6.2. Formalism extension for the Union

Based on the analysis, this section presents the canonical form and normalisation for the Tydi type system including unions Subsequently, the required extensions to the streaming context and semantics are introduced, after which an argument is given to show that the resulting formal framework adheres to structural determinism and progress.

6.2.1. Normalisation and Canonical Form

Firstly, the top-level type $\mathcal{T}$ and normalised type $\mathcal{N}$ must be expanded to include the union type: $\texttt{Union}\langle\mathcal{T}_1,\ldots,\mathcal{T}_n\rangle$, which is normalised to $\texttt{Union}\langle\mathcal{N}_1,\ldots,\mathcal{N}_n\rangle$. We do not add the null type, instead adding 1 to the tag, making unions inherently nulla and up to user specification, thereby also allowing singular unions to exist. For a Union to adhere to Canonical Form, it must contain no primitives, be enclosed in a group carrying a single primitive at the head, all its fields must be normalised types adhering to canonical form, these cannot include unions since they should be enclosed in groups, they can include groups, which can contain additional unions, and they can include streams. Different from groups, singular unions are allowed, since this defines a nullable type. To achieve normalisation, I propose a "*hoist*" function for the Union, similar to the "*coalesce*" function for the Group. It is responsible for enforcing Canonical Form, integrating in the normalisation function without invalidating the normalisation of other types.

Cycle	Itinerary	Judgement / Argumentation
1	[$\mathtt{B}_{78}$, U, G, Root]	$\mathtt{B}_{78}$ parsed, U has no active subtrees ⇒ read tag+term from $\mathtt{B}_{78}$ which contains tag selecting $\mathtt{S}_1$ and is non-terminal; invoke `.activate` to mark $\mathtt{S}_1$ for the next cycle.
2	[$\mathtt{B}_{78}$, $\mathtt{B}_8$, $\mathtt{S}_1$, U, G, Root]	$\mathtt{B}_{78}$ parsed. Because $\mathtt{S}_1$ is active we begin parsing $\mathtt{B}_8$ which is non-terminal. On visiting U we see it has an active subtree ⇒ undo parsing of $\mathtt{B}_{78}$ on the parent stream by invoking `.unparse`.
3	[$\mathtt{B}_{78}$, $\mathtt{B}_8$, $\mathtt{S}_1$, U, G, Root]	$\mathtt{B}_{78}$ parsed, $\mathtt{B}_8$ reaches terminal; $\mathtt{S}_1$ terminates. U now has no active subtrees ⇒ invoke `.activate` to refresh the tag+term from the (most recent) $\mathtt{B}_{78}$ transient parse and schedule activation of the other subtree ($\mathtt{S}_2$) for the next cycle.
4	[$\mathtt{B}_{78}$, $\mathtt{B}_{16}$, $\mathtt{S}_2$, U, G, Root]	$\mathtt{B}_{78}$ parsed, $\mathtt{B}_{16}$ parsed; U has an active subtree (now $\mathtt{S}_2$), so invoke `.unparse` on the parent stream $\mathtt{B}_{78}$.
5	[$\mathtt{B}_{78}$, $\mathtt{B}_{16}$, $\mathtt{S}_2$, U, G, Root]	$\mathtt{B}_{78}$ parsed, $\mathtt{B}_{16}$ parses and is terminal; $\mathtt{S}_2$ terminates. U has no active subtrees and the stored tag is non-terminal; invoke `.activate` which selects the $\mathtt{B}_{12}$ primitive (non-terminal), this has no effect since it has no subtree.
6	[$\mathtt{B}_{78}$, U, G, Root]	$\mathtt{B}_{78}$ parsed and is terminal. U has no active subtrees and the stored tag is non-terminal; invoke `.activate` which selects $\mathtt{S}_1$ and is terminal, thereby marking $\mathtt{S}_1$ for the next cycle.
7	[$\mathtt{B}_8$, $\mathtt{S}_1$, U, G, Root]	$\mathtt{B}_8$ parsed and is terminal; $\mathtt{S}_1$ terminates. U now has no active subtrees and the stored tag is terminal; terminate the Union node. G then has no children and terminates, $\mathtt{S}_0$ has no children and terminates. Root is leaf node.

Table 6.1: The iteration when using the deferred parsing commit method for the extended chat message example: $\mathcal{N} := \mathtt{S}_0\big(\mathtt{G}\langle \mathtt{B}_{78},\ \mathtt{U}\langle \mathtt{S}_1(\mathtt{B}_8, d=1),\ \mathtt{S}_2(\mathtt{B}_{16}, d=1)\rangle\big), d=1\big)$

Union Hoisting Integrating the Union into the normalisation function is defined as:

$$[\![\mathtt{Union}(\mathcal{T}_1, \ldots, \mathcal{T}_k)]\!] \mapsto \mathbf{hoist}\Big(\mathtt{Union}([\![\mathcal{T}_1]\!], \ldots, [\![\mathcal{T}_k]\!])\Big)$$

The hoist function takes a `Union` with already-normalised fields. This guarantees that before application of the hoist function, the fields are either 1) Primitives, 2) normalised streams, or 3) Groups with either a) one primitive at the head, and at least one Union or Stream, or b) Groups containing multiple Unions and/or Streams. The hoist function operates as follows:

1. Calculating the tag size based on the number of fields $b_{tag} = \lceil \log_2(k+1) \rceil$ (where tag 0 is "null"), subsequently calculate the maximum data width b_{prim} from any child $\mathcal{N}_i$ that is a `Bits` or a primitive in a `Group` type.
2. Creating a new primitive $\mathtt{B}_{tag} = \mathtt{Bits}\langle b_{tag} + b_{prim} \rangle$, and removing all primitives from the Union and from Groups in the Union.
3. If this removes all fields (i.e., the original `Union` only contained primitives), the function returns only $\mathtt{B}_{tag}$.
4. Otherwise, it returns $\mathtt{Group}\big(\mathtt{B}_{tag}, \mathtt{Union}(\mathcal{N}''_1, \ldots, \mathcal{N}''_p)\big)$, where $(\mathcal{N}''_1, \ldots, \mathcal{N}''_p)$ is the sequence of remaining non-primitive fields.

6.2.2. Default Semantics proposal

This section extends $\mathcal{R}_{core}$ with the small-step semantics supporting the Union as an example implementation. No existing core transition is modified; instead, I expand the context operations and inference rules, with the goal of preserving structural determinism and progress, enabling the proofs to remain applicable.

Additional context operations

The initialisation of the streaming context Γ adds the ability for the union nodes to maintain a register for recording the tag and termination data. Additionally, Γ is extended with the following context operations:

- $\Gamma.\texttt{unparse}$: reverts any effect that $\Gamma.\texttt{parse}$, $\Gamma.\texttt{send}$ or $\Gamma.\texttt{term}$ may have had for the physical stream corresponding to the Union node currently in focus.
- $\Gamma.\texttt{activate}$: reads the tag and termination data from the parent stream's transient parse, stores the pair $(tag, term)$ in the internal state of Union node U, and marks the subtree corresponding to the stored tag as active for the *subsequent* cycle.

Union ruleset expansion

Union-Busy

$$\frac{\mathsf{focus}(\Gamma) = U \quad \neg\mathsf{isLeaf}(\Gamma)}{\Gamma \to (\Gamma.\texttt{unparse}).\texttt{next}}$$

Union-Terminate

$$\frac{\mathsf{focus}(\Gamma) = U \quad \mathsf{isLeaf}(\Gamma) \quad \mathsf{stored\_tag} = (\_, d_k)}{\Gamma \to \Gamma.\texttt{term}}$$

Union-Activate

$$\frac{\mathsf{focus}(\Gamma) = U \quad \mathsf{isLeaf}(\Gamma) \quad \mathsf{stored\_tag} = (\_, t_k) \quad t_k < d_k}{\Gamma \to (\Gamma.\texttt{activate}).\texttt{next}}$$

Figure 6.3: Additional small-step transitions for Union semantics.

Union-Busy undoes any effect on the parent stream when it has an active subtree and thus is not a leaf node. Union-Terminate terminates itself when it is a leaf node, and its stored termination data indicates that the parent stream has been terminated. Union-Activate reads the tag and termination data from its parent stream, if the union node has no active subtrees and its stored termination data indicates that the parent stream is not terminated. Together, these rules integrate Unions into the semantics corresponding to $\mathcal{P}_{core}$ while adhering to structural determinism and progress.

Compatibility with structural determinism and progress arguments

Structural determinism holds by the fact that: 1) the added rules are distinct from the $\mathcal{R}_{core}$ differentiated by the node in focus being the Union, and 2), the added rules are mutually exclusive since when the union is not a leaf node Union-Busy applies, and when it is a leaf node, either Union-Terminate or Union-Activate is applicable, based on the stored termination signal t_k being d_k for Union-Terminate and $t_k < d_k$ for Union-Activate.

For the progress property, it is slightly less obvious since the addition of Union-Busy can undo the effect of Bits-Parse and Bits-Term, which invalidates the result of the transition analysis 5.3.2. A similar solution as used for proving $\mathcal{P}_{C4}$ can be applied here. By lemma 1, the Union substructure must be finite, therefore a finite number of invocations of M-Cycle will result in a termination of the Union subtree, which violates the premise of Union-Busy and enables either Union-Terminate or Union-Activate, which will strictly decrease either N or S. Therefore, a finite number of transitions will decrease the metric, ensuring the result of our argument is valid and progress is maintained.

7

Conclusion and Discussion

This chapter will present the conclusion of this work, including 1) a brief overview of how the scope developed throughout the period of research, 2) a summary of the contributions, connecting to 3) the impact on the existing Tydi specification, and 4) the future work that remains. Subsequently, 5) the related work and positioning of this research is illustrated, concluding this chapter with 6) a discussion and 7) acknowledgements.

7.1. Research overview

During the period of research, the scope and deliverables were quite ill-specified for a long time, and the direction has changed various times. To provide some additional context, this section first presents an overview of the research period, subsequently outlining the contributions that it resulted in.

7.1.1. Evolution of the Research Scope

The research began with a question that had existed for a while: *Can we prove that compatible Tydi interfaces maintain correct communication in practice?* Initial investigation relied on anecdotal evidence of ambiguities regarding the mapping of nested streams, described as a “missing middle“ between the high-level logical stream and the physical implementation. The focus slowly shifted from “finding out what’s wrong“, which proved difficult without a formal basis to begin with, to constructing a formal foundation for the specification itself.

This resulted in a period of time where I attempted to capture the essence of Tydi; what it intended to effectuate through the way it was defined. Consequently, the scope was narrowed to “TinyTydi,“ a subset of the specification excluding Union types, and focusing on a basic, low complexity mapping. A lot of time was spent on determining which formal framework would be most suitable for describing the Tydi specification. This phase, characterized by the “evolution of the whiteboard“, showed that the proposed normalisation function closely mirrored the hardware synthesis algorithm, and how the semantics could be simplified by ensuring unambiguous structures of the normalised type. I initially introduced a high detail semantics, even reflecting concepts such as maintaining individual buffer pointers in the dynamic formalism, eventually moving to a more abstracted approach using the context-operations. Before a hiatus to work at CERN, a Python-based simulator provided empirical verification, highlighting edge cases and motivating the need for properties like structural determinism.

Upon returning, the objective evolved from defining a stand-alone formalism to integrating it back into the existing Tydi specification. This effort produced the “two-way split“ perspective, distinguishing between static/dynamic and user-facing/inner-working domains. In the final stages of research, I started building upon the concepts that I had proposed instead of tweaking the foundations. I established the Implementation Complexity Metric and Named Property-Sets, driven by an attempt to articulate the fundamental concepts for a broader audience. This led to a formalism that harmonized Tydi’s seemingly incompatible intentions; providing an accessible interface specification for complex hardware streams, while maintaining formal rigour that enabled provably correct hardware implementations.

7.1.2. Summary of Contributions

The contributions of this work can be separated in two sections: An analysis investigating the Tydi specification, what it is and what it intends to be, identifying problems and ambiguities that were speculated to exist and 2) the proposal for a formal foundation for the Tydi specification, capturing the original intent of the specification while providing provably correct transmission / streamspace mapping of arbitrary Tydi instances, ensuring that matching interfaces are rigorously specified and verifiably compatible.

First, through a thorough analysis of the specification and its usage, I identified the conceptual “gap“ in the specification, where it lacked the prescriptive rigour to define how runtime behaviour for complex, nested streams actually produced the streamspace mappings. After synthesis, the hierarchy of physical streams was lost, and how this mapped to the complexity levels remained undefined.

To resolve this, I introduced a tiered formal framework centred on a modular operational semantics. This framework contributes the following:

- **The Two-Way Split:** A separation between the *user-facing* definition and the *inner-working* formalism, and between the *static* type domain and the *dynamic* runtime domain. This allows the specification to remain accessible and flexible for the engineer while utilising more rigorous formalisms to define the inner-workings.
- **Normalisation and Canonical Form:** A procedure to map flexible top-level types to a hardware-oriented Normalised form. This removes structural ambiguities and makes the hierarchy of nested physical streams and their element sizes explicit.
- **The Streaming Context, Interface Properties and Semantics:** The introduction of a stateful abstraction (Γ) bridges the gap between static types and dynamic behaviour. The formalisation of Internal-Stream and Streamspace Properties replace monolithic complexity levels together with other dynamic stream parameters, and map to modular sets of small step operational semantics. These iteratively determine the streamspace mapping for arbitrarily typed instances.
- **The Implementation Complexity Metric (ICM):** A new metric, utilising the detailed specification of the interface implementation, allowing engineers to more accurately quantify the logic cost of specific interface configurations.

A less tangible contribution was expressed by engineers that work with the specification; the formalism shows that what Tydi attempts to achieve, is actually formally feasible. Before, the specification offloaded the rather complex and fault-sensitive work of implementing an interface exactly as it should be implemented according to the spec, to the engineer. When using the formalism, after providing the type and interface properties, the engineer is presented with the normalised inputs streams, and the internal stream properties that these should adhere to, leaving all the complexity of building transfers and managing lane validity/signalling to the specification.

7.2. Related Work and Positioning

Dataflow and stream-processing models A large body of work studies streams from a high-level, software-oriented perspective, providing semantic foundations that inform hardware-oriented formalisms. Type- and trace-based models expose precise composition and ordering properties for streams, which are useful when mapping logical stream specifications to implementations [16, 5, 30, 28]. Other systems and languages (e.g., AquaLang, StreamQRE and related flow dataflow formalisms) show how type and effect systems can enforce progress, ordering, and determinacy properties for transformations over streams [46, 31]. These high-level frameworks provide fundamental concepts for reasoning about the relation between Tydi’s types and its observable stream behaviour.

Formal semantics for hardware design There is an active thread of work utilising operational semantics for hardware aware formalisms. For example, Koika, a core language for Bluespec, gives a deterministic operational semantics for rule-based hardware, together with mechanized artifacts and a verified compiler [9]. Their ORAAT methodology for structural determinism inspired the construction of my modular ruleset approach. Recent efforts also develop mechanized operational semantics for dataflow circuits and structurally-defined small-step semantics for pipelined processors; these works show how transitions can closely mirror real hardware stages and support machine-checked proofs of properties such as determinism and progress [29, 12]. This line of research validates my choice

of small-step operational semantics as a practical and verifiable method for expressing cycle-aware streaming behaviour.

Hardware interface and conformance formalisms Several alternative approaches target interface compatibility and correctness. Tripakis *et al.* formalise a conformance relation between abstract dataflow models and cycle-accurate FSM hardware to preserve performance properties such as throughput and latency [52]. Interface automata and contract-based interface theories capture temporal assumptions and guarantees, permitting compatibility and refinement checks independent of an operational model [4, 7]. These provide the fundamental concepts required for my reasoning about interface compatibility and implementation complexity. Aetherling represents a third, important approach: it encodes space- and time-aware interface properties in a type system (the L^{st} space-time IR with `SSeq` and `TSeq` types) and uses type directed scheduling to produce statically scheduled streaming circuits. This enables a well-typed composition to corresponds to correct-by-construction, statically scheduled hardware implementations [18].

7.2.1. The Tydi Ecosystem

The Tydi framework provides a typed, data-centric approach to defining hardware stream interfaces. Peltenburg *et al.* introduced the Tydi specification as an open standard for expressing complex and variable-length data structures over hardware streams, enabling higher-level, software-like abstractions for streaming dataflow interface designs [40]. Building upon this foundation, Reukers *et al.* developed an intermediate representation (IR) for Tydi that codifies its type system and stream composition rules, allowing structured interfaces to be instantiated, connected, and verified independently of computation logic [44]. Tian *et al.* further extended this ecosystem with *Tydi-lang*, a domain-specific language that integrates Tydi-spec into a hardware description workflow and compiles to the Tydi IR, demonstrating efficient translation from high-level stream-oriented descriptions to synthesizable VHDL [51]. The IR and language together provide a reusable foundation for design tools that reason about streamlet interface contracts. Recent extensions of the Tydi ecosystem emphasize composability and integration with hardware design languages. Cromjongh *et al.* integrate Tydi into Chisel, enabling the generation of Tydi-compliant streaming accelerators with significantly reduced interface boilerplate and improved component reuse [15]. Meloni *et al.* present *Tywaves*, a type-aware waveform viewer that preserves Tydi's hierarchical type information during simulation and debugging [32]. Additionally, the open-source visualisation tool *tydi-stream-vis* provides interactive visualisations of Tydi data structures and stream transfers, further aiding education and design exploration [14]. Tydi's broader motivation aligns with recent work on raising abstraction in FPGA-based dataflow design. Peltenburg and colleagues identify the abstraction gap between high-level big data frameworks and hardware accelerators, arguing that typed streaming specifications like Tydi can help bridge this divide [25]. Similarly, their JSON-to-Arrow FPGA accelerator design using Tydi streams demonstrates the practical benefits of structured, variable-length data handling for high-throughput streaming hardware [39]. Together, these works establish Tydi as a unifying framework for typed hardware streams, bridging high-level data semantics with low-level synthesis, verification, and debugging workflows.

7.3. Future Work

The formalism provides a strong motivation for Tydi, how it can be implemented on both the user facing level, and the formal inner-workings. This success became apparent when the foundation started to empower the addition of new concepts, for example enabling the union integration. Still, there remains a lot of potential for adding parts of the specification that have yet to be incorporated. I will first address the implications for the Tydi specification, illustrating the various paths forwards, subsequently outlining the practical work that will support these efforts.

7.3.1. Implications for the Tydi Specification

Mapping the Tydi specification in its current form to the proposed formalism is an interesting challenge. Arguably, the Logical Stream is the main point of interaction of Tydi, and connecting it to the formalism is not trivial. The top level type is restricted to the Bits, Group, Union and Dim types, but in the Tydi specification, the Dim type represents a Logical Stream shorthand. Therefore, to map these concepts, the top level type should be normalised, subsequently wrapped in an additional logical stream, with its parameters determined by the streamspace properties.

For the complexity levels, the streamspace properties are its obvious parallel. Synchronicity, can be implemented by treating the stream node like the union node; Since canonical form ensures exactly 1 stream node per physical stream, we can implement a rule that when a child stream is visited, it checks the dimension of its parent stream, potentially applying back-pressure by unparsing like the Union node would. This can be expressed by a streamspace property, even providing additional detail regarding when to apply back-pressure, for which dimensionality of the parent and child stream. The throughput parameter in the logical stream is replaced with an explicit lane definition, provided to the user after normalisation has been applied, and the physical streams are concrete.

A fundamental difference is the freedom to express different stream properties per physical stream. In the formalism, the interface properties are determined for the entire stream, each physical stream in the structure reflecting the same decisions regarding complexity and synchronicity, whereas in the original specification, the user is free to express a nested stream that does not adhere to synchronicity, while its parent stream encapsulating stream does. The decision to apply the interface properties globally has been made to ease the formalisation, while retaining the ability to expand the formalism to more complex type constructions. For example, extending the type system by differentiating between synchronised and asynchronous streams, can implement the same behaviour as the group synchronisation described above. This would mean that the inclusion of certain types could be incompatible with certain interface properties, creating a dependency between the static, user-facing domain $\mathcal{T}$, and the dynamic user-facing domain $\mathcal{P}$. Therefore, in order to maintain compatibility with the formalism, if the user wants to violate the synchronisation properties for certain nested streams, these are violated for all streams. This may seem like a big constraint on the Tydi specification, but it is important to note that this only applies to alleviating constraints on the streamspace mapping. For example, if a classic Tydi interface adheres to complexity level 7, it can still create streamspace mappings that would adhere to complexity level 2, it just isn't *guaranteed*. Similarly, in the formalism, if the user wants to remove synchronicity constraints for some nested stream, these are also removed for the parent stream, but the interface can still produce this streamspace mapping if the user supplies the parent stream elements in a "low-complexity" fashion on the internal streams.

As it stands, this will need a more thorough investigation, determining how much freedom we want to give regarding changing complexities halfway through the type structure. In this effort we should also include the shorthand types that have been excluded from the formalism thus far such as the reverse stream.

The signal layout can be specified for each named property-set, similar to the way that the current definition exists for the complexity levels. This means that by creating the named property-sets and the corresponding semantics for each complexity level, the existing signal layout can be mapped to the formalism as is. The container types are defined using the existing Tydi types, and can therefore be mapped naturally mapped to the formalism, since the original Tydi types have a 1 to 1 correspondence with the top level type definition.

Overall, the formalism proves to be a flexible foundation that, though motivating which portions of the original specification to incorporate and which to exclude, will further refine the underlying Tydi specification, removing ambiguities or ill-defined concepts from the original. Since the interface properties are more flexible than the original complexity levels, expanding the formalism can eventually make a formal Tydi specification more expressive than its current form, while also providing provably correct interface implementations and quantifiable logic requirements through the implementation complexity metric.

7.3.2. Practical considerations

Regarding the formalism, various additions can be made. Most of these revolve around expanding the named property-sets, providing each with the correct semantics and an implementation complexity metric. For each property-set, a sufficiently accurate ICM algorithm needs to be defined, such that an interface designer can easily compare the implications of choosing certain properties. In its current state, the ICM is an interesting proposal, but leaves room for improvement since it is not as intuitive as the complexity levels. It can be argued that the complexity levels did not substantiate *how* they actually represented the implementation complexity, but this does not take away from the fact that the ICM in its current state is not an easy metric to use for balancing sending and receiving interface logic. Ideally, the

formalism would provide an algorithmic way for determining the ICM, for each interface specification. Since the interface specification consists of both the type, and interface properties, the a user could also consider different ways to *type* their data to potentially lower the required logic complexity.

Besides this, I note that it would be a great step forward to update the simulator to use the most recent semantics, integrating the union. This will allow for an empirical verification of the union together with various property-sets. Additionally, I have started on a mechanisation of the formalism in Lean4. Fully implementing this would be a significant selling point for critical engineering applications.

Lastly, to better reflect the hardware realities, the simulator can be adapted to an FPGA implementation. Here we could define arbitrary JSON interfaces, synthesising randomized top level types and data instances and parsing them by the generated interfaces, providing empirical evidence for the effectiveness of the formalism. Similar work has already started for the chisel integration of Tydi, providing an illustrative verification of Tydi's use case.

7.4. Discussion

A fundamental consideration has to be made regarding how Tydi presents itself. If the formal rigour provided in this work is incorporated into the specification, it would be very prescriptive regarding implementation. This is necessary to formally verify that no illegal interleavings will exist in the streamspace mapping of compatible Tydi interfaces, but is it not too *restrictive* for a specification? Potentially, the dynamic formalism can just be used for the generator and compositor tooling that implement Tydi, leaving the static specification as a separate entity, giving engineers the freedom to implement their own hardware that adheres to the specification. It remains to be seen how results of this work should be reflected in an update to Tydi. Needless to say, being able to generate compatible component interfaces, by only supplying the type of the stream and the desired interface properties, is a very attractive proposition.

Regarding the implementation complexity metric, it does not account for distinct streaming context states w.r.t. the buffers, only distinguishing states by the tree termination. If we implemented the ICM to also incorporate resource elements required for implementing an interface it would better reflect hardware cost. When considering the "who's responsible for the buffer" example, the ICM would better reflect making design trade-offs. Since currently certain interface properties will not impact the ICM, while in practice there will be a concrete difference in logic required.

Acknowledgements

Many acknowledgments are in order, for the thesis period, and the years leading up to the creation of this work. I feel obliged to mention their names here, making their invaluable contributions that accumulated over the years concrete.

For supervision and Tydi specification expertise
Peter Hofstee, Zaid Al-Ars, Casper Cromjongh, Yongding Tian, Matthijs Brobbel, Jeroen van Straten

For support regarding formalisms, mathematics and the lego editor
Wan Fokkink, Herman Geuvers, Benedikt Ahrens, the r/math community, Jakob van Lenten

For overall support
Jaap & Lena Struik, Marie-Lise Mackloet, Dennis van Gilst, Rex Fleur

For Excellence in Education
Michael Stevens, Jeroen van Nieuwaal, Wessel Oele, Cor Kraaikamp, Grant Sanderson, Mirjam Blaauwboer, Stefan Hugtenburg, Lena Struik

References

[1] Daniel J. Abadi et al. “Aurora: a new model and architecture for data stream management”. In: *The VLDB Journal* 12.2 (Aug. 2003), pp. 120–139. issn: 0949-877X. doi: 10.1007/s00778-003-0095-z. url: https://doi.org/10.1007/s00778-003-0095-z.

[2] Luca Aceto et al. “Rule formats for determinism and idempotence”. In: *Sci. Comput. Program.* 77.7–8 (July 2012), pp. 889–907. issn: 0167-6423. doi: 10.1016/j.scico.2010.04.002. url: https://doi.org/10.1016/j.scico.2010.04.002.

[3] Tyler Akidau et al. “The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing”. In: *Proc. VLDB Endow.* 8.12 (Aug. 2015), pp. 1792–1803. issn: 2150-8097. doi: 10.14778/2824032.2824076. url: https://doi-org.tudelft.idm.oclc.org/10.14778/2824032.2824076.

[4] Luca de Alfaro and Thomas A. Henzinger. “Interface automata”. In: *Proceedings of the 8th European Software Engineering Conference Held Jointly with 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ESEC/FSE-9. Vienna, Austria: Association for Computing Machinery, 2001, pp. 109–120. isbn: 1581133901. doi: 10.1145/503209.503226. url: https://doi.org/10.1145/503209.503226.

[5] Rajeev Alur et al. “Interfaces for Stream Processing Systems”. In: *Principles of Modeling: Essays Dedicated to Edward A. Lee on the Occasion of His 60th Birthday*. Ed. by Marten Lohstroh, Patricia Derler, and Marjan Sirjani. Cham: Springer International Publishing, 2018, pp. 38–60. isbn: 978-3-319-95246-8. doi: 10.1007/978-3-319-95246-8_3. url: https://doi.org/10.1007/978-3-319-95246-8_3.

[6] Apache Software Foundation. *Apache Arrow: A Cross-Language Development Platform for In-Memory Analytics*. https://arrow.apache.org/. Version 15.0.0. 2025.

[7] Ezio Bartocci et al. “Information-flow interfaces”. In: *Formal Methods in System Design* 66.1 (May 2025), pp. 3–48. issn: 1572-8102. doi: 10.1007/s10703-024-00447-0. url: https://doi.org/10.1007/s10703-024-00447-0.

[8] Suhail Basalama and Jason Cong. “Stream-HLS: Towards Automatic Dataflow Acceleration”. In: *Proceedings of the 2025 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. FPGA ’25. Monterey, CA, USA: Association for Computing Machinery, 2025, pp. 103–114. isbn: 9798400713965. doi: 10.1145/3706628.3708878. url: https://doi.org/10.1145/3706628.3708878.

[9] Thomas Bourgeat et al. “The essence of Bluespec: a core language for rule-based hardware design”. In: *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2020. London, UK: Association for Computing Machinery, 2020, pp. 243–257. isbn: 9781450376136. doi: 10.1145/3385412.3385965. url: https://doi.org/10.1145/3385412.3385965.

[10] Paris Carbone et al. “State management in Apache Flink®: consistent stateful distributed stream processing”. In: *Proc. VLDB Endow.* 10.12 (Aug. 2017), pp. 1718–1729. issn: 2150-8097. doi: 10.14778/3137765.3137777. url: https://doi-org.tudelft.idm.oclc.org/10.14778/3137765.3137777.

[11] Sirish Chandrasekaran et al. “TelegraphCQ: continuous dataflow processing”. In: *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*. SIGMOD ’03. San Diego, California: Association for Computing Machinery, 2003, p. 668. isbn: 158113634X. doi: 10.1145/872757.872857. url: https://doi-org.tudelft.idm.oclc.org/10.1145/872757.872857.

[12] Robert J. Colvin and Roger C. Su. “Structural Operational Semantics for Functional and Security Verification of Pipelined Processors”. In: *Computer Aided Verification*. Ed. by Ruzica Piskac and Zvonimir Rakamarić. Cham: Springer Nature Switzerland, 2025, pp. 363–388. isbn: 978-3-031-98668-0.

[13] Jason Cong et al. “FPGA HLS Today: Successes, Challenges, and Opportunities”. In: *ACM Trans. Reconfigurable Technol. Syst.* 15.4 (Aug. 2022). issn: 1936-7406. doi: 10.1145/3530775. url: https://doi.org/10.1145/3530775.

[14] Casper Cromjongh. *abs-tudelft/tydi-stream-vis*. original-date: 2025-04-24T09:49:44Z. Sept. 2025. url: https://github.com/abs-tudelft/tydi-stream-vis (visited on 11/12/2025).

[15] Casper Cromjongh et al. “Hardware-Accelerator Design by Composition: Dataflow Component Interfaces With Tydi-Chisel”. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 32.12 (Dec. 2024), pp. 2281–2292. issn: 1557-9999. doi: 10.1109/TVLSI.2024.3461330.

[16] Joseph W. Cutler et al. “Stream Types”. In: *Proc. ACM Program. Lang.* 8.PLDI (June 2024). doi: 10.1145/3656434. url: https://doi.org/10.1145/3656434.

[17] Jeffrey Dean and Sanjay Ghemawat. “MapReduce: simplified data processing on large clusters”. In: *Commun. ACM* 51.1 (Jan. 2008), pp. 107–113. issn: 0001-0782. doi: 10.1145/1327452.1327492. url: https://doi-org.tudelft.idm.oclc.org/10.1145/1327452.1327492.

[18] David Durst et al. “Type-directed scheduling of streaming accelerators”. In: *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2020. London, UK: Association for Computing Machinery, 2020, pp. 408–422. isbn: 9781450376136. doi: 10.1145/3385412.3385983. url: https://doi.org/10.1145/3385412.3385983.

[19] Marios Fragkoulis et al. “A survey on the evolution of stream processing systems”. In: *The VLDB Journal* 33.2 (Mar. 2024), pp. 507–541. issn: 0949-877X. doi: 10.1007/s00778-023-00819-8. url: https://doi.org/10.1007/s00778-023-00819-8.

[20] Marios Fragkoulis et al. “A survey on the evolution of stream processing systems”. In: *The VLDB Journal* 33.2 (Mar. 2024), pp. 507–541. issn: 0949-877X. doi: 10.1007/s00778-023-00819-8. url: https://doi.org/10.1007/s00778-023-00819-8.

[21] Tim Freeman and Frank Pfenning. “Refinement types for ML”. In: *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*. PLDI ’91. Toronto, Ontario, Canada: Association for Computing Machinery, 1991, pp. 268–277. isbn: 0897914287. doi: 10.1145/113445.113468. url: https://doi.org/10.1145/113445.113468.

[22] Dan R. Ghica. “Function interface models for hardware compilation”. In: *Ninth ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMPCODE2011)*. July 2011, pp. 131–142. doi: 10.1109/MEMCOD.2011.5970519.

[23] David Goldberg et al. “Using collaborative filtering to weave an information tapestry”. In: *Commun. ACM* 35.12 (Dec. 1992), pp. 61–70. issn: 0001-0782. doi: 10.1145/138859.138867. url: https://doi-org.tudelft.idm.oclc.org/10.1145/138859.138867.

[24] Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. “Language primitives and type discipline for structured communication-based programming”. In: *Programming Languages and Systems*. Ed. by Chris Hankin. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 122–138. isbn: 978-3-540-69722-0.

[25] Joost Hoozemans et al. “FPGA Acceleration for Big Data Analytics: Challenges and Opportunities”. In: *IEEE Circuits and Systems Magazine* 21.2 (Secondquarter 2021), pp. 30–47. issn: 1558-0830. doi: 10.1109/MCAS.2021.3071608.

[26] David M Kahn, Jan Hoffmann, and Runming Li. *Big-Stop Semantics: Small-Step Semantics in a Big-Step Judgment*. 2025. arXiv: 2508.15157 [cs.PL]. url: https://arxiv.org/abs/2508.15157.

[27] Gilles Kahn. “The Semantics of a Simple Language for Parallel Programming”. In: *IFIP Congress*. 1974. url: https://api.semanticscholar.org/CorpusID:18030506.

[28] Shadaj Laddad et al. “Flo: A Semantic Foundation for Progressive Stream Processing”. In: *Proc. ACM Program. Lang.* 9.POPL (Jan. 2025). doi: 10.1145/3704845. url: https://doi.org/10.1145/3704845.

[29] Tony Law, Delphine Demange, and Sandrine Blazy. “A Mechanized Semantics for Dataflow Circuits”. In: *Proc. ACM Program. Lang.* 9.OOPSLA1 (Apr. 2025). doi: 10.1145/3720432. url: https://doi.org/10.1145/3720432.

[30] Konstantinos Mamouras et al. “Data-trace types for distributed stream processing systems”. In: *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2019. Phoenix, AZ, USA: Association for Computing Machinery, 2019, pp. 670–685. isbn: 9781450367127. doi: 10.1145/3314221.3314580. url: https://doi.org/10.1145/3314221.3314580.

[31] Konstantinos Mamouras et al. “StreamQRE: modular specification and efficient evaluation of quantitative queries over streaming data”. In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2017. Barcelona, Spain: Association for Computing Machinery, 2017, pp. 693–708. isbn: 9781450349888. doi: 10.1145/3062341.3062369. url: https://doi.org/10.1145/3062341.3062369.

[32] Raffaele Meloni, H. Peter Hofstee, and Zaid Al-Ars. “Tywaves: A Typed Waveform Viewer for Chisel”. In: *2024 IEEE Nordic Circuits and Systems Conference (NorCAS)*. Oct. 2024, pp. 1–6. doi: 10.1109/NorCAS64408.2024.10752465.

[33] Jan de Muijnck-Hughes and Wim Vanderbauwhede. “A Typing Discipline for Hardware Interfaces”. In: *33rd European Conference on Object-Oriented Programming (ECOOP 2019)*. Ed. by Alastair F. Donaldson. Vol. 134. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019, 6:1–6:27. isbn: 978-3-95977-111-5. doi: 10.4230/LIPIcs.ECOOP.2019.6. url: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ECOOP.2019.6.

[34] Nicholas Ng et al. “Session types: towards safe and fast reconfigurable programming”. In: *SIGARCH Comput. Archit. News* 40.5 (Mar. 2012), pp. 22–27. issn: 0163-5964. doi: 10.1145/2460216.2460221. url: https://doi.org/10.1145/2460216.2460221.

[35] Rachit Nigam, Pedro Henrique Azevedo de Amorim, and Adrian Sampson. “Modular Hardware Design with Timeline Types”. In: *Proc. ACM Program. Lang.* 7.PLDI (June 2023). doi: 10.1145/3591234. url: https://doi.org/10.1145/3591234.

[36] Bengt Nordström, Kent Petersson, and Jan M. Smith. *Programming in Martin-Lo¨f’s type theory: an introduction*. USA: Clarendon Press, 1990. isbn: 0198538146.

[37] Johan Peltenburg et al. “Fletcher: A Framework to Efficiently Integrate FPGA Accelerators with Apache Arrow”. In: *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*. Sept. 2019, pp. 270–277. doi: 10.1109/FPL.2019.00051.

[38] Johan Peltenburg et al. “Supporting Columnar In-memory Formats on FPGA: The Hardware Design of Fletcher for Apache Arrow”. In: *Applied Reconfigurable Computing*. Ed. by Christian Hochberger et al. Cham: Springer International Publishing, 2019, pp. 32–47. isbn: 978-3-030-17227-5.

[39] Johan Peltenburg et al. “Tens of gigabytes per second JSON-to-Arrow conversion with FPGA accelerators”. In: *2021 International Conference on Field-Programmable Technology (ICFPT)*. Auckland, New Zealand: IEEE International Conference on Field-Programmable Technology (FPT), Dec. 2021, pp. 1–9. doi: 10.1109/ICFPT52863.2021.9609833.

[40] Johan Peltenburg et al. “Tydi: An Open Specification for Complex Data Structures Over Hardware Streams”. In: *IEEE Micro* 40.4 (2020), pp. 120–130. doi: 10.1109/MM.2020.2996373.

[41] Benjamin C. Pierce. *Types and Programming Languages*. Cambridge, MA: MIT Press, 2002, p. 648. isbn: 9780262162098.

[42] Gordon D Plotkin. “The origins of structural operational semantics”. In: *The Journal of Logic and Algebraic Programming* 60-61 (2004). Structural Operational Semantics, pp. 3–15. issn: 1567-8326. doi: https://doi.org/10.1016/j.jlap.2004.03.009. url: https://www.sciencedirect.com/science/article/pii/S1567832604000268.

[43] Craig Ramsay, Louise H. Crockett, and Robert W. Stewart. “On Applications of Dependent Types to Parameterised Digital Signal Processing Circuits”. In: *2021 IEEE International Midwest Symposium on Circuits and Systems (MWSCAS)*. Aug. 2021, pp. 787–791. doi: 10.1109/MWSCAS47672.2021.9531730.

[44] Matthijs Reukers et al. *An Intermediate Representation for Composable Typed Streaming Dataflow Designs*. Aug. 2023. doi: 10.48550/arXiv.2308.13436.

[45] J.J.M.M. Rutten. “Universal coalgebra: a theory of systems”. In: *Theoretical Computer Science* 249.1 (2000). Modern Algebra, pp. 3–80. issn: 0304-3975. doi: https://doi.org/10.1016/S0304-3975(00)00056-6. url: https://www.sciencedirect.com/science/article/pii/S0304397500000566.

[46] Klas Segeljakt, Seif Haridi, and Paris Carbone. “AquaLang: A Dataflow Programming Language”. In: *Proceedings of the 18th ACM International Conference on Distributed and Event-Based Systems*. DEBS ’24. Villeurbanne, France: Association for Computing Machinery, 2024, pp. 42–53. isbn: 9798400704437. doi: 10.1145/3629104.3666030. url: https://doi.org/10.1145/3629104.3666030.

[47] Michael Stonebraker, Uundefinedur Çetintemel, and Stan Zdonik. “The 8 requirements of real-time stream processing”. In: *SIGMOD Rec.* 34.4 (Dec. 2005), pp. 42–47. issn: 0163-5808. doi: 10.1145/1107499.1107504. url: https://doi-org.tudelft.idm.oclc.org/10.1145/1107499.1107504.

[48] Harold Johannis Arnold Struik. *TinyTydi simulator*. Version 0.2.0. Delft University of Technology, June 1, 2025. url: https://gitlab.com/hstruik/tinytydi.

[49] Douglas Terry et al. “Continuous queries over append-only databases”. In: *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data*. SIGMOD ’92. San Diego, California, USA: Association for Computing Machinery, 1992, pp. 321–330. isbn: 0897915216. doi: 10.1145/130283.130333. url: https://doi-org.tudelft.idm.oclc.org/10.1145/130283.130333.

[50] James Thomas, Pat Hanrahan, and Matei Zaharia. “Fleet: A Framework for Massively Parallel Streaming on FPGAs”. In: *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS ’20. Lausanne, Switzerland: Association for Computing Machinery, 2020, pp. 639–651. isbn: 9781450371025. doi: 10.1145/3373376.3378495. url: https://doi-org.tudelft.idm.oclc.org/10.1145/3373376.3378495.

[51] Yongding Tian et al. “Tydi-lang: A Language for Typed Streaming Hardware”. In: *Proceedings of the SC ’23 Workshops of the International Conference on High Performance Computing, Network, Storage, and Analysis*. SC-W ’23. Denver, CO, USA: Association for Computing Machinery, 2023, pp. 521–529. isbn: 9798400707858. doi: 10.1145/3624062.3624539. url: https://doi.org/10.1145/3624062.3624539.

[52] Stavros Tripakis et al. “Tokens vs. Signals: On Conformance between Formal Models of Dataflow and Hardware”. In: *J. Signal Process. Syst.* 85.1 (Oct. 2016), pp. 23–43. issn: 1939-8018. doi: 10.1007/s11265-015-0971-y. url: https://doi.org/10.1007/s11265-015-0971-y.

[53] Deepak Vohra. “Apache Avro”. In: *Practical Hadoop Ecosystem: A Definitive Guide to Hadoop-Related Frameworks and Tools*. Berkeley, CA: Apress, 2016, pp. 303–323. isbn: 978-1-4842-2199-0. doi: 10.1007/978-1-4842-2199-0_7. url: https://doi.org/10.1007/978-1-4842-2199-0_7.

[54] Matei Zaharia et al. “Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing”. In: *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*. NSDI’12. San Jose, CA: USENIX Association, 2012, p. 2.

[55] Shuhao Zhang et al. “Hardware-Conscious Stream Processing: A Survey”. In: *SIGMOD Rec.* 48.4 (Feb. 2020), pp. 18–29. issn: 0163-5808. doi: 10.1145/3385658.3385662. url: https://doi-org.tudelft.idm.oclc.org/10.1145/3385658.3385662.

A

Detailed Simulation Trace

Figure A.1: Three consecutive steps of the simulator, showing the parse tree and the transfer on stream 0

```
--- Simulation Step 101 ---
STREAM
 NGroup <-----
    NBits(0)
    Stream
       NBits(1)
    Stream
        NBits(2)

--- Global Transfer History
    ---
Stream 0:
28     46 (t) 69
55     *      18
93     *      25
87     *      73
80 (t) *      69

Stream 1:
13     7
23     61
27     97
14 (t) 6
*      80 (t)
*      *
*      *
*      *
*      *

Stream 2:
49     1
100    10
61 (t) 33
*      38
*      89
*      40
*      31 (t)
*      *
*      *
```

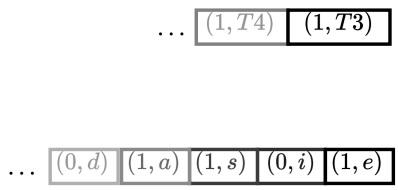
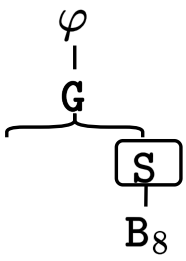
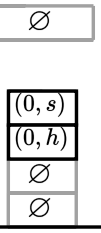
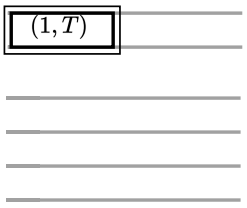
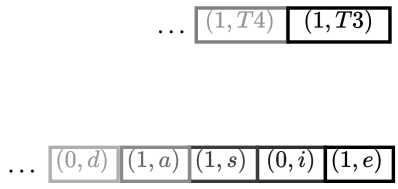
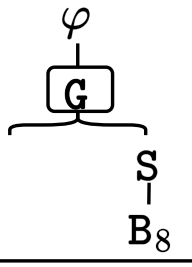
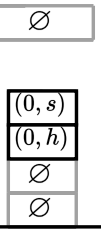
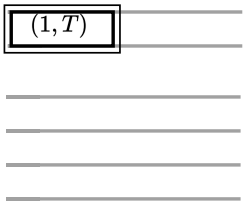
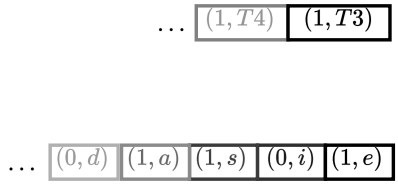
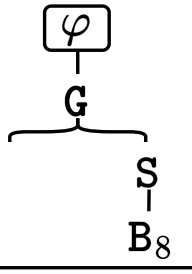
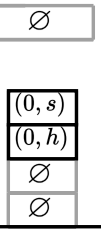
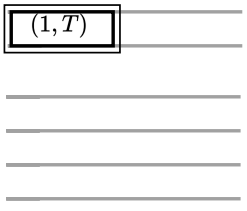
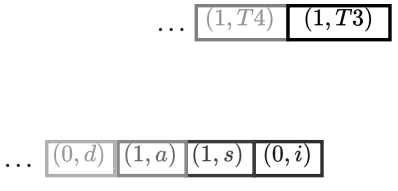
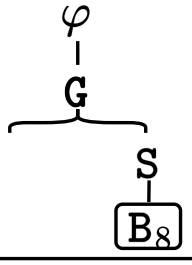
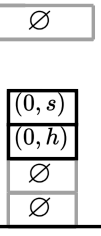
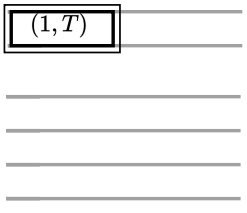
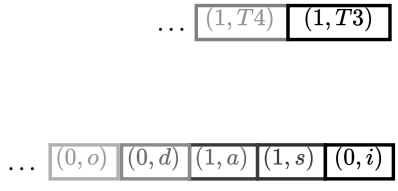
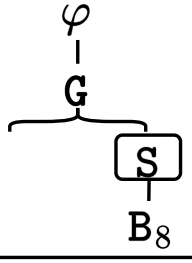
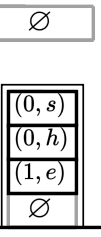
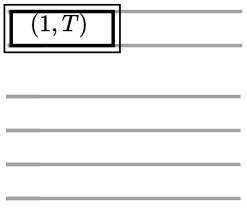
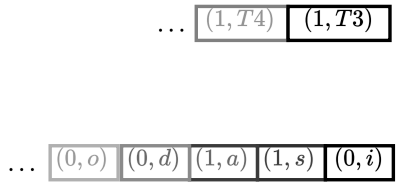
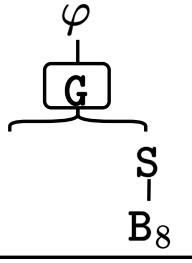
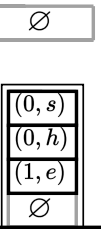
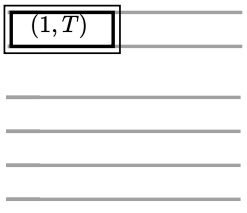
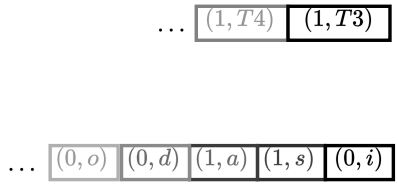
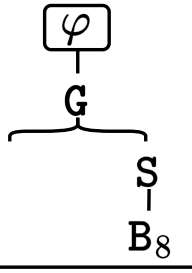
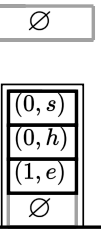
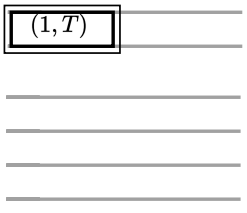
```
--- Simulation Step 102 ---
STREAM
 NGroup
    NBits(0) <-----
    Stream
       NBits(1)
    Stream
        NBits(2)

--- Global Transfer History
    ---
Stream 0:
28     46 (t) 69
55     *      18
93     *      25
87     *      73
80 (t) *      69

Stream 1:
13     7
23     61
27     97
14 (t) 6
*      80 (t)
*      *
*      *
*      *
*      *

Stream 2:
49     1
100    10
61 (t) 33
*      38
*      89
*      40
*      31 (t)
*      *
*      *

Output Transfers from apply_
    semantics (step 102):
Transfer(stream_idx=0,
    payloads=[
  Element(stream_idx=0,
      payload=13, is_terminal=
      False),
  Element(stream_idx=0,
      payload=50, is_terminal=
      False),
  Element(stream_idx=0,
      payload=7, is_terminal=
      False),
  Element(stream_idx=0,
      payload=25, is_terminal=
      True),
  None
])
```

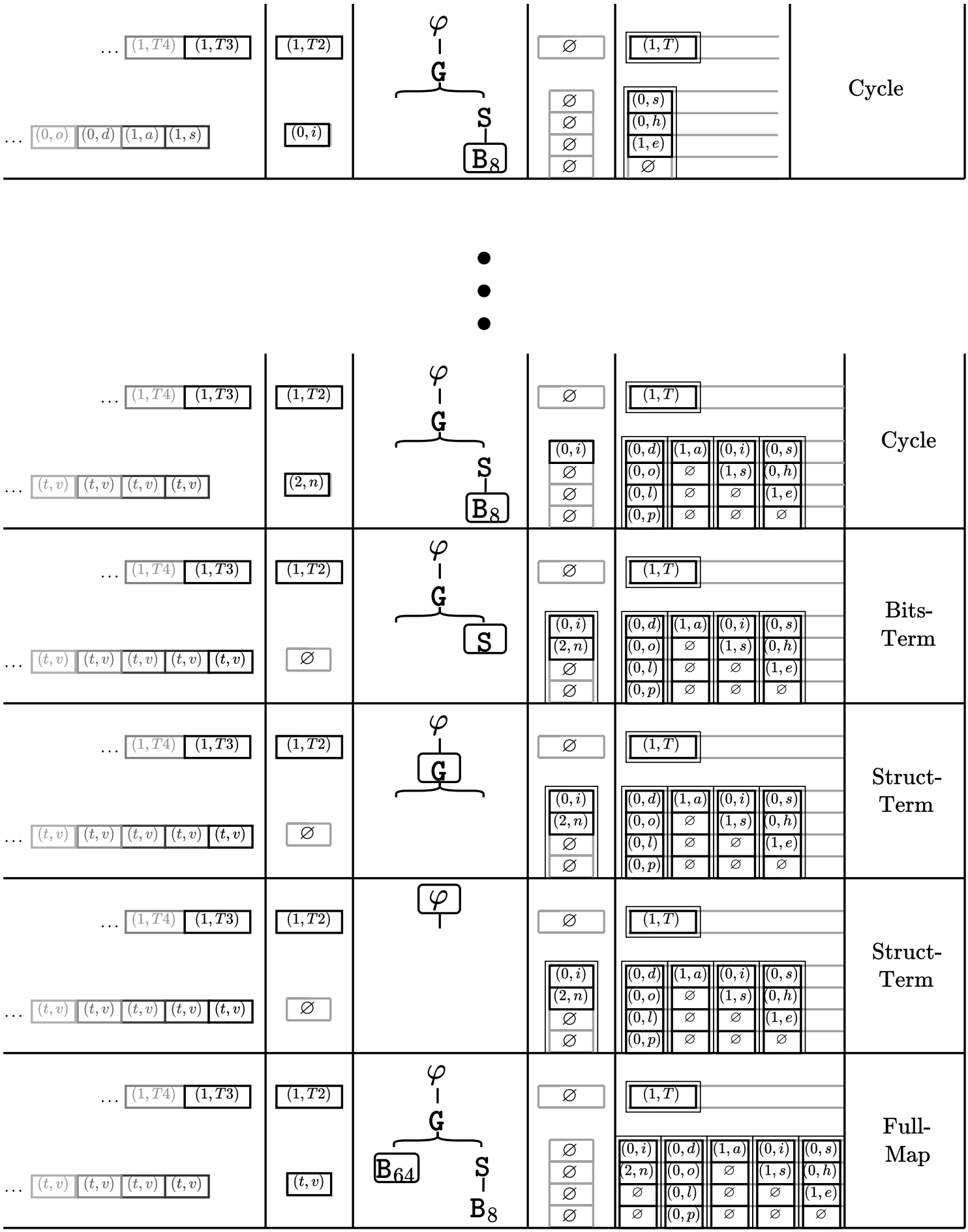
```
--- Simulation Step 103 ---
STREAM
 NGroup
    *NBits(0)
    Stream <-----
       NBits(1)
    Stream
        NBits(2)

--- Global Transfer History
    ---
Stream 0:
13     28     46 (t) 69
50     55     *      18
7      93     *      25
25 (t) 87     *      73
*      80 (t) *      69

Stream 1:
13     7
23     61
27     97
14 (t) 6
*      80 (t)
*      *
*      *
*      *
*      *

Stream 2:
49     1
100    10
61 (t) 33
*      38
*      89
*      40
*      31 (t)
*      *
*      *s
```

B

Full Parsing Example Diagram

$\Gamma := \{\ I$	$Z_{\mathcal{N}}$	$T_B\ \}$	*Stream Space*	Justification
… $(1,T2)$ $(1,T)$ … $(1,s)$ $(0,i)$ $(1,e)$ $(0,h)$ $(0,s)$	φ — G — B_{64}, S — B_8	∅ ∅ ∅ ∅ ∅		Initial state
… $(1,T2)$ … $(1,s)$ $(0,i)$ $(1,e)$ $(0,h)$	φ — G — B_{64}, S — B_8	$(1,T)$ $(0,s)$	∅ ∅ ∅ ∅ ∅	Cycle
… $(1,T3)$ $(1,T2)$ … $(1,a)$ $(1,s)$ $(0,i)$ $(1,e)$ $(0,h)$	φ — G — S — B_8	∅ $(0,s)$	$(1,T)$ ∅ ∅ ∅ ∅	Bits-Term
… $(1,T3)$ $(1,T2)$ … $(1,a)$ $(1,s)$ $(0,i)$ $(1,e)$ $(0,h)$	φ — G — S — B_8	∅ ∅	$(1,T)$ $(0,s)$ ∅ ∅ ∅	Bits-Parse
… $(1,T3)$ $(1,T2)$ … $(1,a)$ $(1,s)$ $(0,i)$ $(1,e)$ $(0,h)$	φ — G — S — B_8	∅ ∅	$(1,T)$ $(0,s)$ ∅ ∅ ∅	Struct-Enter
… $(1,T3)$ $(1,T2)$ … $(1,a)$ $(1,s)$ $(0,i)$ $(1,e)$ $(0,h)$	φ — G — S — B_8	∅ ∅	$(1,T)$ $(0,s)$ ∅ ∅ ∅	Struct-Enter
… $(1,T4)$ $(1,T3)$ … $(1,a)$ $(1,s)$ $(0,i)$ $(1,e)$	φ — G — S — B_8	$(1,T2)$ $(0,h)$	∅ $(0,s)$ ∅ ∅ ∅	$(1,T)$ Cycle

$\Gamma := \{\ I$	$Z_{\mathcal{N}}$	$T_B\ \}$	*Stream Space*	Justification
				Bits-Parse
				Struct-Enter
				Struct-Enter
				Cycle
				Bits-Send-Inner
				Struct-Enter
				Struct-Enter


$\Gamma := \{\ I$
$Z_{\mathcal{N}}$
$T_B\}$
Stream Space
Justification
φ
G
S
B_8
B_{64}
Cycle
Bits-Term
Struct-Term
Full-Map