A Typing Discipline for Hardware Interfaces

Jan de Muijnck-Hughes 🄳
University of Glasgow, UK
Jan.deMuijnck-Hughes@glasgow.ac.uk

Wim Vanderbauwhede 🄳
University of Glasgow, UK
Wim.Vanderbauwhede@glasgow.ac.uk

Abstract

Modern Systems-on-a-Chip (SoC) are constructed by composition of IP (Intellectual Property) Cores with the communication between these IP Cores being governed by well described interaction protocols. However, there is a disconnect between the machine readable specification of these protocols and the verification of their implementation in known hardware description languages. Although tools can be written to address such separation of concerns, the tooling is often hand written and used to check hardware designs *a posteriori*. We have developed a dependent type-system and proof-of-concept modelling language to reason about the physical structure of hardware interfaces using user provided descriptions. Our type-system provides correct-by-construction guarantees that the interfaces on an IP Core will be well-typed if they adhere to a specified standard.

1 Introduction

Hardware Description Languages (HDLs) such as Verilog, SystemVerilog and VHDL are designed to realise both the structure and behaviour of hardware systems. Hardware is modelled as interconnected *components* (modules) that are connected through *ports*; ports being individual wires or a collection of wires. Ports carry data, and the flow of data on a port is directional. HDLs abstract over groupings of ports (*port groups*) as an *interface*, and present values at higher levels of abstraction such as integers and strings. A component can have multiple interfaces that each send multiple values, and can each be characterised differently. An initiating interface (*initiator*) initiates communication, and the targeted interface (*target*) is the recipient of the communication.

Modern hardware design is not just about digital circuits, it is also about describing systems of systems. For instance, System-on-a-Chip (SoC) views hardware modules (IP Cores) as boxes connected using well-known and bespoke interfaces. The structure, and behaviour, of these interfaces are described in natural language documents [3, 41, 4]. Such standards documents will present an *abstract interface description* which is a global view of

an interface agnostic to its endpoint usage, and will provide salient structural information (using natural language) about each port in the interface required for realisation in a HDL. Details provided include a port's size, sensitivity, necessity, flow, and dependencies between the details specified. Further, these documents describe behavioural characteristics of the interfaces as a whole. For example, how ports are grouped to describe different channels.

While circuit-level designs are required to implement behaviour at a low-level, the designer must also ensure that the components in a SoC design are correctly connected according to the provided specification. Standardised machine readable formats such as `IP-XACT` capture many of the structural information found within the standards document's natural language descriptions [23]. However, interface specifications written using `IP-XACT` cannot be parameterised, nor can structural dependencies be specified between ports and over interfaces. Further, not all the information contained within a natural language document can be specified using `IP-XACT`. For instance, `IP-XACT` does not support the definition of *strobe*, a signal carried on a separate port that is linked to an individual bit in multi-wire data bus. The number of strobe is dependent on the size of the bus. Conversely, the machine readable specification can present information more clearly than the specification document itself. For example, port necessity for the `APB` specification is more clearly described in the `IP-XACT` specification than in the standards document.

Generally speaking, there is a disconnect between the description of an interface's structure in a standards document, its representation in a standardised machine readable format, and its enforcement in a HDL. When instantiating these interfaces in a HDL there are no mechanisms to ensure that the characterised interfaces respect the specifications. As a result, mismatches between the specifications and their implementations are common.
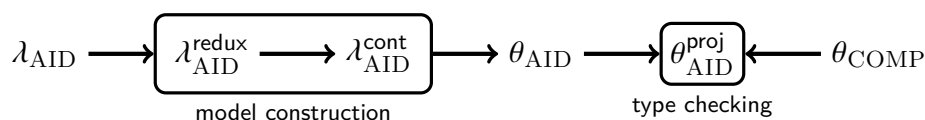
1.1 Contributions

The aim of our work is to improve the security and safety of SoC design by utilising state-of-the-art concepts from programming language theory to provide greater correct-by-construction guarantees over the structural and behavioural aspects of SoC designs.

Dependent type-systems present a rich and expressive setting that allows for precise program properties to be stated and verified directly in the language's type-system [28]. Such type-systems also support modelling of resource usage in the style of substructural typing [40, 6]. By building upon existing work from hardware design we can use these concepts to construct a type-based formal description of abstract interfaces, and formally validate that concrete component interfaces adhere to these descriptions at design-time using type checking.

Specifically, we make the following contributions:

1. We present a type-driven modelling framework (CORDIAL) for reasoning about interfaces on components within a SoC design.

2. We show the use of CORDIAL for describing an exemplar protocol MUNGO, and discuss how CORDIAL can be used to model real-world protocol specifications: `APB`, `LocalLink` and `AXI` [4, 3, 41].

3. We describe the formalisation of our framework in the dependently typed programming language Idris [9] that also constitutes a proof-of-concept implementation.

Figure 1 summarises the core constructs that comprise CORDIAL and their relations. Modelling information is taken from the `IP-XACT` standard [23] and existing work [30] to construct a model (θ_{AID}) to represent *abstract interface descriptions*. Our model construction language (λ_{AID}) is a simple extension to the Simply Typed Lambda Calculus (STLC) and

$$\lambda_{\mathrm{AID}} \longrightarrow \boxed{\lambda_{\mathrm{AID}}^{\mathsf{redux}} \longrightarrow \lambda_{\mathrm{AID}}^{\mathsf{cont}}} \longrightarrow \theta_{\mathrm{AID}} \longrightarrow \boxed{\theta_{\mathrm{AID}}^{\mathsf{proj}}} \longleftarrow \theta_{\mathrm{COMP}}$$

<div align="center">model construction   type checking</div>

Figure 1 Relationships between various languages, models, and intermediate representations.

models parameterised specifications as computable functions, and allows dependencies to be made between signals. The type-system of λ_{AID} follows a substructural design [40, 5] allowing correctness guarantees towards labelling of signals to be lifted into the type-system. Model construction is from reduction of λ_{AID} instances to a reduced form ($\lambda_{\mathrm{AID}}^{\mathsf{redux}}$) which is then evaluated to construct θ_{AID} instances using continuation passing. Concrete interfaces are modelled using θ_{COMP} to present components in a SoC with multiple interfaces.

Inspired by notions of global and local types from *Session Types* [22] abstract interface specifications are treated as a global description that is characterised to a local description – $\theta_{\mathrm{AID}}^{\mathsf{proj}}$. By embedding the projected model ($\theta_{\mathrm{AID}}^{\mathsf{proj}}$) into the type of the interface description (θ_{COMP}) the model's type-system ensures that a local type is satisfied by its global type. Further, the concept of *thinnings* [1, § 3] captures a specification's optional ports, and allows optional ports to be knowingly skipped.

Application of CORDIAL would see it embedded within existing SoC tooling and to enrich existing HDLs with static design-time mechanisms that would make mismatches between interface specification and implementation impossible and thus reduce errors, increase design productivity and enhance safety and security of the SoC designs. The transformations of specification instances, and model projections would be automatic and hidden from users. Protocol designers would have a tool (based on λ_{AID}) to design interface specifications. During the SoC design phase SoC designers use these specifications to annotate their components (θ_{COMP}) and ensure their port selections are correct.

1.2 Outline

Section 2 presents a running example that further motivates our work. Section 3 introduces our model for abstract interface descriptions (θ_{AID}) and the specification language (λ_{AID}) used to construct θ_{AID} model instances. Section 4 details our model (θ_{COMP}) for describing concrete components and how projected θ_{AID} instances are used to type-check interfaces. Section 5 briefly describes the formalisation of CORDIAL in Idris, and Section 6 considers use of the framework to model real-world interaction protocol specifications. Section 7 discusses the efficacy of the framework, and considers related work. The paper concludes with a discussion over future work in Section 8.

Notation.   For simplicity the syntax for standard algebraic types are abstracted over. Similar abstractions are used for dependent types. Single-field variant types are presented with a constructor name as the label and the body being a n-ary tuple. Where possible simple typing rules are embedded within the presentation of abstract syntax and types. Model types are denoted using blackboard style letters. Types from construction languages are denoted using uppercase Greek letters. Constructs subscripted with: d are from θ_{AID}; and p are from $\theta_{\mathrm{AID}}^{\mathsf{proj}}$.

2   The Mungo Protocol

Presentation of CORDIAL will be aided through consideration of an exemplar protocol
(MUNGO) that captures salient physical properties common to many interaction protocols.

Table 1 Signal descriptions for MUNGO.

Name	Width	Direction	Necessity	Source	Sensitivity
SYS_CLK	1	Always System	Optional	System	High
CTRL_R	1	To Initiator	Required	IP	High
CTRL_W	1	To Initiator	Required	IP	High
DATA	32/16	Bi-Directional	Required	IP	High
ADDR	8/4	To Target	Required	IP	High
ERR_MODE	2	To Initiator	Target Optional	IP	High
ERR_INFO	user defined	To Initiator	Target Optional	IP	High

Table 1 presents the signal descriptions (abstract interface description) for MUNGO.
Behaviourally the protocol represents the reading and writing of data from the initiating
IP Core to the target[1]. MUNGO provides unicast style communication, it does not support
broadcast communication through a shared bus. A system clock (SYS_CLK) can send signals
to both the target and initiator. The clock is optional as the clock source for the specified
component might not go through this interface. Reading and writing are dictated by the
initiator using control wires CTRL_R and CTRL_W. A data bus is bidirectional and data can
have a width of 32 or 16 bits. The address bus is eight or four bits in width. Error reporting
is optional where: ERR_MODE indicates the type of error; and ERR_INFO is the message itself.
The width of error messages are left to the implementer. All wires have high sensitivity.

```
interface Mungo #(AWIDTH = 8, DWIDTH = 32, EWIDTH) (input bit clk);

  logic [AWIDTH-1:0] addr;
  logic [DWIDTH-1:0] data;
  logic [2:0]        errType;
  logic [EWIDTH-1:0] errInfo;
  logic ctrlr, ctrlw;

  modport initiator(input clk,
                    input errType, errInfo,
                    output addr, ctrlw, ctrlr,
                    inout data);

  modport target(input clk,
                 output errType, errInfo,
                 input addr, ctrlw, ctrlr,
                 inout data);
endinterface
```

Figure 2 Realising MUNGO using SystemVerilog.

---

[1] Only the physical structure of the interface is considered. How the framework can be extended to
capture behaviour properties is discussed later.

Figure 2 shows how Mungo can be realised in SystemVerilog. The interface is parameterised, however, SystemVerilog only allows such interfaces to have a single default parameter. Mungo has multiple default parameters. Two characterised interfaces for both an initiator and target are presented as modports. Error related signals and clock information are optional. Interfaces can take many other valid structural forms. SystemVerilog supports unrestricted use of *dangling ports*, in which a receiving port is unconnected. In these cases the value received is taken as the default value as dictated by the port's type. A designer can also deviate from the specification and make required ports optional. When connecting two modules together that support Mungo the wrong interface might be left out. Further, not all HDLs support the concept of dangling ports.


```
module System (output wire clock);
```

```
module Initiator (
  input  wire    clock,
  output wire    write,
  output wire    read,
  input  [1:0]   errorType,
  input  [7:0]   errorInfo,
  output [7:0]   address,
  inout  [31:0]  data
);
```

```
module Target(
  input wire    clock,
  input wire    write,
  input wire    read,



  input [7:0]   address,
  inout [31:0]  data
);
```

```
module Top ();
  wire clock, write, read;
  wire [7:0]  address;
  wire [31:0] data;

  System s (.clock(clock));
  Initiator i (.clock(clock), .write(write), .read(read), .errorType(), .errorInfo(),
          .address(address), .data(data));
  Target t (.clock(clock), .write(write), .read(read), .address(address), .data(data));
endmodule
```

■ **Figure 3** SystemVerilog module definitions adhering to Mungo, and signal flow indicators.

Figure 3 presents a second use of SystemVerilog to declare two components that support Mungo, and connect them together. Within Figure 3 we present a visualisation of the module interconnections. Within this example, the initiating component has two dangling ports for optional error reporting, and we have deviated from the specification in using different labels. SystemVerilog provides name and positional oriented connection of modules. Ultimately, the programmer is responsible for ensuring that the ports are connected correctly, and can wire or name ports freely. We need to ensure that the interfaces on a module are valid against their respective specifications.

3   Abstract Interface Descriptions

This section presents a model (θ_{AID}) for reasoning about abstract interface descriptions, together with a language (λ_{AID}) for model construction. How λ_{AID} instances are transformed into θ_{AID} instances is also described.

Taking inspiration from IP-XACT [23], abstract interfaces are modelled as a named-tuple of port descriptions and other metadata. This is a common approach as seen by existing work [30, 19]. For each port a variety of emergent properties are also tracked. Dependent

types control invariants over model structure and property values. θ_{AID} model instances are not parameterised, the construction language (λ_{AID}) facilitates creation of parameterised specifications and ensures models created use unique labels using substructural typing.

3.1 Properties

Ghica et al. [19] modelled ports according to their size and signal direction. However, there are other important properties as shown by McKechnie [30]. Ports are uniquely identified using *labels*. Similar types of ports share similar behaviour. A port can: communicate data; provide addressing information; provide clock ticks; trigger a reset; signal an interrupt; indicate control; provide port-level behavioural information; or is used in a general sense. Differentiating between these behaviours is important when connecting two (or multiple) ports together. Not all ports in an interface are required, and how a port responds to changes in signal (sensitivity) should also be captured.

For interfaces, salient properties concern the style of communication. Does the interface expect to interact with a set number of other interfaces, or interact directly with another interface?

3.2 Model Components

			Metadata
$i, n : \mathbb{N}^*$	::=	Natural numbers greater than zero.	
$r : L$	::=	User provided labels	
$s : \mathbb{S}$	::=	High | Low | Rising | Falling | Insensitive	Wire Sensitivity
$h : \mathbb{H}$	::=	System | IP	Port Origin
$c_{\text{cstyle}} : \mathbb{C}_{\text{style}}$	::=	Broadcast | Unicast	Comm. Style
			Ports
$k_p : \mathbb{K}_P$	::=	WIRE | ARRAY	Kind
$\mathbb{A} : \mathbb{K}_P \to \text{Type}$			Type
$t : \mathbb{A}(\text{ARRAY})$	::=	Data | Address	Values
$t : \mathbb{A}(\text{WIRE})$	::=	Clock | Reset | Interrupt | Control	
$t : \mathbb{A}(k_p)$	::=	General | Info	
			Labels
$k_l : \mathbb{K}_L$	::=	NMD | IDX	Kind
$\mathbb{L} : \mathbb{K}_L \to \text{Type} \to \text{Type}$			Type
$l : \mathbb{L}(\text{NMD}, L)$	::=	Named(r)	Values
$l : \mathbb{L}(\text{IDX}, L)$	::=	Indexed(r, i)	

$$e_c ::= n \mid r \mid s \mid h \mid c_{\text{style}} \mid k_p \mid t \mid l$$

$$\mathcal{T}_c : \text{Type} ::= L \mid \mathbb{N}^* \mid \mathbb{S} \mid \mathbb{H} \mid \mathbb{C}_{\text{style}} \mid \mathbb{K}_L \mid \mathbb{L}(k_l, L) \mid \mathbb{K}_P \mid \mathbb{A}(k_p)$$

Figure 4 Common terms and their types.

Figure 4 presents the shared terms and types used throughout the models and languages presented. Numbers originate from the set of natural numbers greater than zero. Port labels are specification dependent and assumed to be typed enumerations. Signals are either sensitive or insensitive. Sensitive wires are level sensitive (high, or low) or edge sensitive – rising or falling. Signals either originate from a system interface, or from another component – IP Core. An interface's communication style is either broadcast or unicast.

Of interest is how a port's type and label are modelled. A "kind" provides type-level disambiguation between different kinds of labels and ports. θ_{AID} & λ_{AID} support several types of port, and different port types have different shapes. Data and address ports will always be an array of ports. Clocking information, resets, interrupts, and control ports will always use a single wire. General and information ports can have either shape. When describing widths, the shape of the port will dictate possible values.

Ports must be labelled, however, they can also share a common name with a fixed set of other ports – cf. strobes in `APB` and `AXI`. A label is either *named* and is used once, or *indexed* and used i times. To prevent ambiguities between different label families, the type for labels is indexed with the type associated with the underlying name used.

3.3 A Model for Abstract Interfaces

Figure 5 presents the core modelling constructs, and typing rules, for θ_{AID} model instances. Within θ_{AID}, signal flow is directional. Signals flow from: initiator to target ($\rightsquigarrow$); target to initiator ($\leftsquigarrow$); bidirectional ($\leftrightsquigarrow$); always received ($\overline{\leftsquigarrow}$); or always produced – ($\overline{\rightsquigarrow}$). Ports can be completely optional (?), target optional ($?_t$), initiator optional ($?_i$), or are required – (!). Wire ports have width ($\mathbb{1}_d$), and array ports have width ($\mathsf{w}_d(n)$) where n is greater than one. Ports can be specified with an arbitrary width – (∞_d). The type for port widths is paramterised by a port kind. This enforces the relation that ports will have the correct width for their kind i.e. a wire can only have length one.

A port description is a named tuple comprising of the port's label (l), kind (k_p), type (t), flow (f), necessity (o), width (w), sensitivity (s), and origin – (h). The type for ports is a type synonym for the following dependent function:

$$\mathsf{port}_p : \mathbb{L}(L, k_l) \rightarrow (k_p : \mathbb{K}_P) \rightarrow \mathbb{A}(k_p) \rightarrow \mathbb{F} \rightarrow \mathbb{O}_d \rightarrow \mathbb{W}_d(k_p) \rightarrow \mathbb{S} \rightarrow \mathbb{H} \rightarrow \mathbb{P}_d(L)$$

Dependently typed terms allow for an invariant to hold during term construction. The port kind associated with a port type and width, must respect the specified port kind. Thus, if the port has kind WIRE then its width and type must be suitable for a wire. Further, the port type itself ($\mathbb{P}_d(L)$) is indexed by the type associated with label.

Ports are grouped in a cons-style collection ($ps : \mathbb{PG}_d(L)$) whose type is also parameterised by the type associated with labels. All ports in a group must have the same type of label. An abstract interface is a named tuple containing the interface's communication style, max number of initiators and targets, and a collection of ports.

3.3.1 Example

Figure 6 presents a θ_{AID} instance for MUNGO – Table 1. An enumerated type provides labelling information. θ_{AID} instances are, however, not parameterised. MUNGO is an interface that can be instantiated with several address and data bus widths. The example instance for MUNGO in Figure 6 provides holes ($\square$) in place of precise widths. Exact values for widths must be presented. Further, there are no restrictions on label use, one can easily duplicate the use of a name. The next section presents a language to present parameterised specifications and ensure label uniqueness.

$$f : \mathbb{F} ::= \rightsquigarrow | \leftsquigarrow | \leftrightsquigarrow | \overline{\leftsquigarrow} | \overline{\rightsquigarrow} \qquad\qquad \text{Signal Flow}$$

$$o_d : \mathbb{O}_d ::= ? \ | ?_i \ | ?_t \ | ! \qquad\qquad \text{Necessity}$$

$$w : \mathbb{W}_d \, (k_p) ::= \mathbb{1}_d \ | \ \mathsf{w}_d \, (n) \ | \ \infty_d \qquad\qquad \text{Widths}$$

$$p_d : \mathbb{P}_d \, (L) ::= \mathsf{port}_d (l, k_p, t, f, o_d, w_d, s, h) \qquad\qquad \text{Port}$$

$$ps_d : \mathbb{PG}_d \, (L) ::= \emptyset_d \ | \ p_d ::_d ps_d \qquad\qquad \text{Portgroup}$$

$$i_d : \mathbb{I}_d \, (L) ::= \mathsf{iface}_d (\mathsf{cstyle}, n, n, ps_d) \qquad\qquad \text{Interface}$$

$$e_{\mathsf{aidl}} ::= e_c \ | \ f \ | \ o_d \ | \ w_d \ | \ p_d \ | \ ps_d \ | \ i_d \qquad\qquad \text{Expressions}$$

$$\mathcal{T}_{\mathsf{aidl}} ::= T \in \mathcal{T}_c \ | \ \mathbb{F} \ | \ \mathbb{O}_d \ | \ \mathbb{W}_d \, (k_p) \ | \ \mathbb{P}_d \, (L) \ | \ \mathbb{PG}_d \, (L) \ | \ \mathbb{I}_d \, (L) \qquad\qquad \text{Types}$$

(a) Terms and types.

$$\mathrm{DWU} \ \frac{k_p : \mathbb{K}_P}{\infty_d : \mathbb{W}_d \, (k_p)} \qquad\qquad \mathrm{DWO} \ \frac{}{\mathbb{1}_d : \mathbb{W}_d \, (\mathsf{WIRE})} \qquad\qquad \mathrm{DWM} \ \frac{i : \mathbb{N}^*, [i \geq 2]}{\mathsf{w}_d \, (i) : \mathbb{W}_d \, (\mathsf{ARRAY})}$$

$$\mathrm{PD} \ \frac{l : \mathbb{L}(L, k_l) \qquad ty : \mathbb{A} \, (k_p) \qquad f : \mathbb{F} \qquad \overset{k_p : \mathbb{K}_P}{w_d : \mathbb{W}_d \, (k_p)} \qquad s : \mathbb{S} \qquad o_d : \mathbb{O}_d \qquad h : \mathbb{H}}{\mathsf{port}_d (l, k_p, ty, f, o_d, w_d, s, h) : \mathbb{P}_d \, (L)}$$

$$\mathrm{PGD\text{-}E} \ \frac{}{\emptyset_d : \mathbb{PG}_d \, (L)} \qquad\qquad \mathrm{PGD\text{-}C} \ \frac{p : \mathbb{P}_d \, (L) \qquad ps : \mathbb{PG}_d \, (L)}{p ::_d ps : \mathbb{PG}_d \, (L)}$$

$$\mathrm{ID} \ \frac{c : \mathbb{C}_{\mathsf{style}} \qquad \mathrm{maxI} : \mathbb{N}^* \qquad \mathrm{maxT} : \mathbb{N}^* \qquad ps : \mathbb{PG}_d \, (L)}{\mathsf{iface}_d (c, \mathrm{maxI}, \mathrm{maxT}, ps) : \mathbb{I}_d \, (L)}$$

(b) Typing Rules.

🟨 **Figure 5** Definition of θ_{AID}.

$$L ::= C \ | \ R \ | \ W \ | \ D \ | \ A \ | \ E \ | \ I$$

$$\mathsf{iface}_d(\mathsf{Unicast}, 1, 1, \mathsf{port}_d(\mathsf{Named}(C), \mathsf{WIRE}, \mathsf{Clock}, \overline{\leftsquigarrow}, ?, \mathbb{1}_d, \mathsf{High}, \mathsf{System})$$

$$::_d \mathsf{port}_d(\mathsf{Named}(R), \mathsf{WIRE}, \mathsf{Control}, \rightsquigarrow, !, \mathbb{1}_d, \mathsf{High}, \mathsf{IP})$$

$$::_d \mathsf{port}_d(\mathsf{Named}(W), \mathsf{WIRE}, \mathsf{Control}, \rightsquigarrow, !, \mathbb{1}_d, \mathsf{High}, \mathsf{IP})$$

$$::_d \mathsf{port}_d(\mathsf{Named}(D), \mathsf{ARRAY}, \mathsf{Data}, \leftrightsquigarrow, !, \mathsf{w}_d \, (\square), \mathsf{High}, \mathsf{IP})$$

$$::_d \mathsf{port}_d(\mathsf{Named}(A), \mathsf{ARRAY}, \mathsf{Address}, \rightsquigarrow, !, \mathsf{w}_d \, (\square), \mathsf{High}, \mathsf{IP})$$

$$::_d \mathsf{port}_d(\mathsf{Named}(E), \mathsf{ARRAY}, \mathsf{Info}, \leftsquigarrow, ?_t, \mathsf{w}_d \, (2), \mathsf{High}, \mathsf{IP})$$

$$::_d \mathsf{port}_d(\mathsf{Named}(I), \mathsf{ARRAY}, \mathsf{Info}, \leftsquigarrow, ?_t, \infty_d, \mathsf{High}, \mathsf{IP})$$

$$::_d \emptyset_d)$$

🟨 **Figure 6** MUNGO as a partial θ_{AID} instance.

3.4 Specifying Interface Descriptions

Figure 5 presents a model instance that is dependently typed, however, the model design itself has several limitations. First, labels are not required to be unique. Second, model instances cannot be parameterised. We address these issues through creation of a description language λ_{AID}. An extension of the STLC, λ_{AID} describes the construction of model instances. Specifications are a sequencing of port descriptions, and other metadata. Function, and application, in λ_{AID} provide parameterisation of specifications, and descriptions of structural dependencies. Evaluation of λ_{AID}, using continuation passing, constructs instances of a θ_{AID} model. A substructural type-system provides further correct-by-construction guarantees that labels are unique. Construction semantics detail model instance construction from λ_{AID} programs.

3.4.1 Counting Label Usage

Substructural type-system's extend existing type-systems with extra information [40]. Labels in θ_{AID} instances are required to be unique. The type-system for λ_{AID} is designed to ensure that label usage is linear: A label can only be used once.

Inspired by the work of McBride [29] we utilise a "rig" to capture label usage. For our bespoke use case a rig of the same style is not required. McBride's rig is for computation (addition and multiplication), and our rig is for usage accounting only. We define our rig as:

▶ **Definition 1** (Rig o' 2). *Let* $\mathbb{R} = \{\text{used}, \text{free}\}$ *be a set, with an operation* $\text{use}(u)$ *to change a* $u \in \mathbb{R}$ *as follows:*

$$\text{use}(u) ::= \begin{cases} \text{free} & \mapsto \text{used} \\ \text{used} & \mapsto \text{used} \end{cases}$$

3.4.2 Terms

$$
\begin{array}{llr}
e ::= & i \mid n \mid r \mid f \mid k_p \mid s \mid h \mid c & \text{Constants} \\
& \mid (\text{add } e\, e) \mid (\text{sub } e\, e) \mid (\text{mul } e\, e) \mid (\text{div } e\, e) & \text{Maths} \\
& \mid \star_1 \mid \omega & \text{Unit Values \& Variables} \\
& \mid e; e \mid \lfloor e \rfloor \mid \underline{\text{let }} \omega \underline{\text{ be }} e \underline{\text{ in }} e & \text{Statements} \\
& \mid (\lambda(i) \cdot e) \mid e \,\$\, n \mid (\lambda(i; [i \in ps]) \cdot e) \mid e \,\$_{[i \in ps]} i & \text{Function \& Application} \\
& \mid \text{label}(n) & \text{Label Creation} \\
& \mid \text{portDesc}(\omega, k_p, ty, f, o, w, s, h) \mid \text{replicate}(i, e) & \text{Port Specification} \\
& \mid \underline{\text{stop}} & \text{Stopping} \\
& \mid \text{setCommunication}(c) \mid \text{setMaxInitiators}(n) & \text{Metadata} \\
& \mid \text{setMaxTargets}(n) &
\end{array}
$$

▪ **Figure 7** Terms for λ_{AID}.

Figure 7 presents the terms for λ_{AID}. Common structures from Figure 4 are included except for the terms for labels. Terms can be sequenced, and bound to variables using let-bindings. Pure values are indicated with $\lfloor e \rfloor$. Combined the terms "Let", "Seq", "Unit"

and "Pure" form a monadic computation context in which the labels and their usage are the computation in context. Although, sequencing is presented separately from "Let"-bindings, sequencing can also be described as a "Let"-binding where ω is bound to $\star_1$.

The term stop denotes the end of a specification such that all labels are used. Terms are presented to represent functions, and function application. Predicated versions of functions and application exist to restrict parameters of type $\mathbb{N}^*$ to predefined sets of whole numbers. Whole labels are created using a single term. Port declarations are similar to port construction in Figure 5a, except that rather than a direct label, port descriptions must take a label variable. There are terms for setting communication style, and max number of initiators and targets. Within λ_{AID}, labels are not indexed. Ports with an indexable label are indicated using replicate.

A simple arithmetic language with binary operators to operate on whole numbers is embedded within λ_{AID}. Supported operations are addition, subtraction, multiplication, and division. With this, user provided widths can be used to construct arithmetic dependencies on the number of ports in a specification. This is described using replicate. Allowing for data dependent port specifications (i.e. strobes) to be supported.

3.4.3 Type-System

$$T \in \mathcal{T} ::= L \mid T_c \in \mathcal{T}_c \setminus (\mathbb{L}(k_l, L)) \mid \mathbb{F} \mid \mathbb{O}_d \mid \mathbb{W}_d(k_p) \qquad \text{Types for Constants}$$
$$\mid 1 \mid \Lambda(L, u) \mid \Psi(L) \qquad\qquad\qquad \text{Types for Unit/Labels/Port}$$
$$\mid T \to T \mid (i : \mathbb{N}^*) \overset{[i \in ps]}{\longrightarrow} T \qquad\qquad \text{Types for Functions}$$
$$\Gamma ::= \varnothing \mid \Gamma + (e : T) \mid \Gamma \pm (e : T) \qquad\qquad \text{Context}$$

🟨 **Figure 8** Types & typing context for λ_{AID}.

Figure 8 presents the types for λ_{AID}. Here L is a placeholder to represent a user defined set of labels. Several types are taken from existing constructs (Figures 4 and 5a) without the type for labels, ports, port groups and interfaces. Three new types are introduced. First is the unit type (1) to represent terms that do not represent computations. Second is the type for label variables ($\Lambda(L, u)$), indexed by the type of the underlying label value and parameterised with usage information from the "Rig o' 2". Function types follow the standard definition, and predicated functions are restricted to acting on whole numbers.

Within λ_{AID} well-typed contexts (Γ) comprise of name type pairings. Contexts can be extended using (+), and named terms updated using (±).

Section 3.4.3 presents the typing rules for λ_{AID}. For brevity the typing rules for maths expressions are not provided. Like the syntax definition for λ_{AID} itself, the typing rules follow that of the STLC, but with extensions for describing abstract interfaces. Rules VAR, LAM, APP, LET, PURE, and SEQ follow standard conventions with one noticeable difference that follows from the work of Atkey [5]: Usage information associated with labels presents stateful information. The monad form by "Let", "Seq", "Unit", and "Pure" is a Hoare Monad that allows state information for label usage to be threaded through the entire computation [8]. The notation $\Gamma_{old} \vdash e \dashv \Gamma_{new}$ represents updating the context from Γ_{old} to Γ_{new}. The context will change only if the rules are well-typed. Where the notation is not used implies the context does not change.

$$\text{VAR} \; \frac{\omega : T \in \Gamma}{\Gamma \vdash \omega : T} \qquad \text{LET} \; \frac{\Gamma_1 \vdash a : T_1 \dashv \Gamma_2 \qquad \Gamma_2 + (\omega : T_1) \vdash b : T_2 \dashv \Gamma_3}{\Gamma_1 \vdash \underline{\text{let}} \; \omega \; \underline{\text{be}} \; a \; \underline{\text{in}} \; b : T_2 \dashv \Gamma_3} \qquad \text{PURE} \; \frac{x : T \in \Gamma}{\Gamma \vdash \lfloor x \rfloor : T}$$

$$\text{LAM} \; \frac{\Gamma_1 + (i : T_1) \vdash e : T_2 \dashv \Gamma_2}{\Gamma_1 \vdash (\lambda(i) \cdot e) : T_1 \rightarrow T_2 \dashv \Gamma_2} \qquad \text{APP} \; \frac{\Gamma_1 \vdash f : T_1 \rightarrow T_2 \dashv \Gamma_2 \qquad \Gamma_1 \vdash i : T_1}{\Gamma_1 \vdash f \; \$ \; i : T_2 \dashv \Gamma_2}$$

$$\text{PLAM} \; \frac{\Gamma_1 + (i : \mathbb{N}^*) \vdash e : T_2 \dashv \Gamma_2 \qquad ps = \{i_1, \, \ldots, \, i_n\} \qquad [i \in ps]}{\Gamma_1 \vdash (\lambda(i ; [i \in ps]) \cdot e) : \mathbb{N}^* \xrightarrow{[i \in ps]} T_2 \dashv \Gamma_2}$$

$$\text{PAPP} \; \frac{\Gamma_1 \vdash f : (i : \mathbb{N}^*) \xrightarrow{[i \in ps]} T \dashv \Gamma_2 \qquad \Gamma_1 \vdash i' : \mathbb{N}^* \qquad ps = \{i_1, \, \ldots, \, i_n\} \qquad [i' \in ps]}{\Gamma_1 \vdash f \; \$_{[i \in ps]} i' : T \dashv \Gamma_2}$$

$$\text{SEQ} \; \frac{\Gamma_1 \vdash e_1 : T_1 \dashv \Gamma_2 \qquad \Gamma_2 \vdash e_2 : T_2 \dashv \Gamma_3}{\Gamma_1 \vdash e_1 ; e_2 : T_2 \dashv \Gamma_3} \qquad\qquad \text{UNIT} \; \frac{}{\Gamma \vdash \star_1 : 1}$$

$$\text{LBL} \; \frac{\Gamma \vdash L : \mathsf{Type} \qquad \Gamma \vdash n : L}{\Gamma \vdash \mathsf{label}(n) : \Lambda(L, \mathsf{free})} \qquad \text{STOP} \; \frac{\forall \omega : \Lambda(L, u) \in \Gamma \, [u \equiv \mathsf{used}]}{\Gamma \vdash \underline{\mathsf{stop}} : 1 \dashv \varnothing}$$

$$\text{PORT} \; \frac{\Gamma \vdash l : \Lambda(L, \mathsf{free}) \qquad \Gamma \vdash k_p : \mathbb{K}_P}{\Gamma \vdash ty : \mathbb{A}(k_p) \qquad \Gamma \vdash f : \mathbb{F} \qquad \Gamma \vdash o : \mathbb{O}_d \qquad \Gamma \vdash w : \mathbb{W}_d(k_p) \qquad \Gamma \vdash s : \mathbb{S} \qquad \Gamma \vdash h : \mathbb{H}}{\Gamma \vdash \mathsf{portDesc}(l, k_p, ty, f, o, w, s, h) : \Psi(L) \dashv \Gamma \pm (l : \Lambda(L, \mathsf{used}))}$$

$$\text{REP} \; \frac{\Gamma_1 \vdash i : \mathbb{N}^* \qquad [i > 2] \qquad \Gamma_1 \vdash e : \Psi(L) \dashv \Gamma_2}{\Gamma_1 \vdash \mathsf{replicate}(i, e) : 1 \dashv \Gamma_2} \qquad \text{MAX-I} \; \frac{\Gamma \vdash n : \mathbb{N}^*}{\Gamma \vdash \mathsf{setMaxInitiators}(n) : 1}$$

$$\text{MAX-T} \; \frac{\Gamma \vdash n : \mathbb{N}^*}{\Gamma \vdash \mathsf{setMaxTargets}(n) : 1} \qquad \text{COM} \; \frac{\Gamma \vdash c : \mathbb{C}_{\mathsf{style}}}{\Gamma \vdash \mathsf{setCommunication}(c) : 1}$$

Figure 9 Typing rules for λ_{AID}.

Predicated functions, and their application, mirror their plain counterparts but have a side-condition that requires the predicate to hold true for type-checking to be successful.

The typing rules for labels and ports use the "Rig o' 2" to instantiate and augment the usage count for types for labels – $\Lambda(L, u)$. These are the type-level computations that enforce correct label usage. Rule LBL presents the typing rule for label creation, initialising the type with usage free. Rule STOP describes the end conditions for λ_{AID} programs, and results in erasure of the context. λ_{AID} programs will successfully type-check only if all label variables have been used. Rule PORT specifies a new port, and consumes labels. A label ω with type $\Lambda(L, u)$, and usage u can only be used if the usage is free. If the label is available to use the type for ω in resulting computations will be $\Lambda(L, \mathsf{used})$, and thus be unavailable.

Replication of a port description (Rule REP) details that a port will be replicated if the number of replications is greater than two. The remaining rules detail the simple typing rules for the remaining terms.

3.4.4 Example

$$\varnothing \vdash \text{M\scriptsize UNGO} : (x : \mathbb{N}^*) \overset{[x \in \{32,16\}]}{\longrightarrow} (y : \mathbb{N}^*) \overset{[y \in \{8,4\}]}{\longrightarrow} 1 \dashv \varnothing$$

$$\text{M\scriptsize UNGO} := (\lambda(x; [x \in \{32, 16\}]) \; \cdot \; (\lambda(y; [y \in \{8, 4\}]) \; \cdot$$

setCommunication(Unicast); setMaxInitiators(1); setMaxTargets(1)

; <u>let</u> c <u>be</u> label(C) <u>in</u>

<u>let</u> r <u>be</u> label(R) <u>in</u>

<u>let</u> w <u>be</u> label(W) <u>in</u>

<u>let</u> d <u>be</u> label(D) <u>in</u>

<u>let</u> a <u>be</u> label(A) <u>in</u>

<u>let</u> e <u>be</u> label(E) <u>in</u>

<u>let</u> i <u>be</u> label(I) <u>in</u>

portDesc(c, WIRE, Clock, $\overline{\leftsquigarrow}$, ?, $\mathbb{1}_d$, High, System)

; portDesc(r, WIRE, Control, $\rightsquigarrow$, !, $\mathbb{1}_d$, High, IP)

; portDesc(w, WIRE, Control, $\rightsquigarrow$, !, $\mathbb{1}_d$, High, IP)

; portDesc(d, ARRAY, Data, $\leftrightsquigarrow$, !, $\mathsf{w}_d\,(x)$, High, IP)

; portDesc(a, ARRAY, Address, $\rightsquigarrow$, !, $\mathsf{w}_d\,(y)$, High, IP)

; portDesc(e, ARRAY, Info, $\leftsquigarrow$, $?_t$, $\mathsf{w}_d\,(2)$, High, IP)

; portDesc(e, ARRAY, Info, $\leftsquigarrow$, $?_t$, ∞_d, High, IP)

; <u>stop</u>))

■ **Figure 10** M\scriptsize UNGO specified using λ_{AID}.

Figure 10 demonstrates how M\scriptsize UNGO is specified using λ_{AID}. The concrete set of labels from Figure 6 are reused. The specification for M\scriptsize UNGO is parameterised through function specification. This is reflected in the specification's type signature. The use of the specification to create θ_{AID} instances is also restricted to values of $x \in \{32, 16\}$ and $y \in \{8, 4\}$. A type error will occur if values for x and y were chosen that were not in the provided sets. The type signature also shows the initial and end contexts for the function; start with nothing, end with nothing. Different specification instances are generated through application of the function to different values.

3.5 Building Models from Specifications

In this section the set of transformations to construct θ_{AID} instances from λ_{AID} programs are described. We first reduce λ_{AID} programs to core terms, and use continuations to represent model construction as evaluation of a reduced λ_{AID} program.

Figure 11 presents the reduction rules for reducing λ_{AID} programs to core terms following standard conventions. Reduction of $\mathbb{N}^*$ values mirrors reduction of natural numbers but with smallest value being one not zero. The reduced version of a λ_{AID} program is called $\lambda_{\text{AID}}^{\text{redux}}$, and the reduction of a λ_{AID} program l to its reduced form (l') by the operation $l \Downarrow^{redux} l'$. Much like the STLC reduction of λ_{AID} programs will also be strongly normalising.

$$\text{LET } \frac{e_1 \Downarrow^{\mathsf{redux}} \lfloor e_1' \rfloor}{\underline{\mathsf{let}}\ \omega\ \underline{\mathsf{be}}\ e_1\ \underline{\mathsf{in}}\ e_2 \Downarrow^{\mathsf{redux}} e_2[\omega/e_1']} \qquad \text{LAM } \frac{f \Downarrow^{\mathsf{redux}} (\lambda(x) \cdot b) \qquad a \Downarrow^{\mathsf{redux}} \lfloor a' \rfloor}{f\ \$\ a \Downarrow^{\mathsf{redux}} b[x/a']}$$

$$\text{PLAM } \frac{f \Downarrow^{\mathsf{redux}} (\lambda(x; [x \in ps]) \cdot b) \qquad a \Downarrow^{\mathsf{redux}} a' \qquad [a \in ps]}{f\ \$_{[x \in ps]}a \Downarrow^{\mathsf{redux}} b[x/a']} \qquad \text{ADD } \frac{e_1 \Downarrow^{\mathsf{redux}} i_1 \qquad e_2 \Downarrow^{\mathsf{redux}} i_2}{(\mathsf{add}\ e_1\ e_2) \Downarrow^{\mathsf{redux}} i_1 + i_2}$$

$$\text{SUB } \frac{e_1 \Downarrow^{\mathsf{redux}} i_1 \qquad e_2 \Downarrow^{\mathsf{redux}} i_2}{(\mathsf{sub}\ e_1\ e_2) \Downarrow^{\mathsf{redux}} i_1 - i_2} \quad \text{MUL } \frac{e_1 \Downarrow^{\mathsf{redux}} i_1 \qquad e_2 \Downarrow^{\mathsf{redux}} i_2}{(\mathsf{mul}\ e_1\ e_2) \Downarrow^{\mathsf{redux}} i_1 \times i_2} \quad \text{DIV } \frac{e_1 \Downarrow^{\mathsf{redux}} i_1 \qquad e_2 \Downarrow^{\mathsf{redux}} i_2}{(\mathsf{div}\ e_1\ e_2) \Downarrow^{\mathsf{redux}} i_1 \div i_2}$$

Figure 11 Reduction rules for λ_{AID}.

To construct θ_{AID} model instances, the reduced form, $\lambda_{\mathrm{AID}}^{\mathsf{redux}}$, is first transformed into a continuation ($\lambda_{\mathrm{AID}}^{\mathsf{cont}}$) in the style of Hatcliff and Danvy [21]. This transformation is denoted using $[\![\ l\ ]\!]_{\mathsf{cont}}$, where l is an instance of $\lambda_{\mathrm{AID}}^{\mathsf{redux}}$. Using this approach we can make model construction more easily checkable for termination, and model construction comes from evaluation of $\lambda_{\mathrm{AID}}^{\mathsf{cont}}$ instances. For brevity we do not provide the definitions of $\lambda_{\mathrm{AID}}^{\mathsf{cont}}$ and $[\![\ l\ ]\!]_{\mathsf{cont}}$, and remark that they follow standard contructions [21].

Evaluation of $\lambda_{\mathrm{AID}}^{\mathsf{cont}}$ instances, which we denote using $\Downarrow^{\mathsf{cont}}$, transforms: label variables into labels; port declarations into port descriptions; and repeated port declarations into port groups. Each evaluation of the accessor functions for setting description values replaces the previously see value, and a default set of values are supplied initially. When evaluated, the continuation returns a tuple containing the final interface metadata and collated port groups.

The complete steps to construct a θ_{AID} from λ_{AID} are defined as:

▶ **Definition 2** (Construction of θ_{AID} from λ_{AID}). *Let m be a θ_{AID} model instance, and l be a λ_{AID} program. The construction of m is defined as the reduction of l to an instance l' of $\lambda_{\mathrm{AID}}^{\mathsf{redux}}$. This instance l', is then evaluated to an instance of $\lambda_{\mathrm{AID}}^{\mathsf{cont}}$ that is then reduced using $\Downarrow^{\mathsf{cont}}$ to produce m:*

$$[\![\ l\ ]\!]_{\lambda \mapsto \theta}^{\mathrm{AID}} = [\![\ l \Downarrow^{\mathsf{redux}} l'\ ]\!]_{\mathsf{cont}} \Downarrow^{\mathsf{cont}} m$$

4 Specifying IP Core Interfaces

Section 3 described the specification and construction of θ_{AID} instances, these are *abstract interface descriptions*. This section looks at the specification of components in a SoC design and how guarantees are made that the physical interfaces satisfy given θ_{AID} instances.

A model for reasoning about components (θ_{COMP}) is introduced. Each component comprises of a set of physical interface models whose interfaces are satisfied by a θ_{AID} instance. Interface satisfaction is a two-step process: first a θ_{AID} instance is projected ($\upharpoonright$) using the specified endpoint e, creating the projected model $\theta_{\mathrm{AID}}^{\mathsf{proj}}$. Second, the projected model is used in the type of an interface to provide a type-level invariant that the presented interface satisfies the projected model. Dependently typed model terms ensures that if an interface is well-typed then the interface satisfies the provided specification.

$$t : \mathbb{E} ::= \mathsf{Initiator} \mid \mathsf{Target} \qquad\qquad \text{Endpoint}$$
$$d_p : \mathbb{D}\,(t, f) ::= + \mid - \mid \pm \qquad\qquad \text{Direction}$$
$$o_p : \mathbb{O}\,(t, o_d) ::= \,? \mid ! \qquad\qquad \text{Necessity}$$
$$p_p : \mathbb{P}_p\,(L, t, p_d) ::= \mathsf{port}_p(l, k_p, t, d_p, o_p, w_d, s, h) \qquad\qquad \text{Port}$$
$$ps_p : \mathbb{PG}_p\,(L, t, ps_d) ::= \emptyset_p \mid p_p ::_p ps_p \qquad\qquad \text{Portgroup}$$
$$i_p : \mathbb{I}_p\,(L, t, i_d) ::= \mathsf{iface}_p(c_{\mathsf{style}}, n, n, ps_p) \qquad\qquad \text{Interface}$$
$$e_p ::= e_d \mid t \mid d_p \mid o_p \mid p_p \mid ps_p \mid i_p \qquad\qquad \text{Expressions}$$
$$\mathcal{T}_p ::= T \in \mathcal{T}_d \mid \mathbb{E} \mid \mathbb{D}\,(t, f) \mid \mathbb{O}\,(t, o_d) \qquad\qquad \text{Types}$$
$$\mid \mathbb{P}_p\,(L, t, p_d) \mid \mathbb{PG}_p\,(L, t, p_d) \mid \mathbb{I}_p\,(L, t, i_d)$$

Figure 12 Terms and types for $\theta_{\mathsf{AID}}^{\mathsf{proj}}$.

4.1 Projecting Abstract Interfaces

Figure 12 presents the terms, and salient types for a projected interface model instance. A projected interface represents the local view of an abstract interface. The structure of a projected interface mirrors that of an abstract interface, and values only differ w.r.t. a port's signal flow (direction), and necessity. The type's for ports, port groups, and interfaces are parameterised by the type of labels associated with ports, the endpoint that the term was projected to, and the originating abstract interface. The types are indexed with the originating term to allow for structural invariants, for example a port's shape, to be specified. – cf. Section 3.3.

A projected port is either unidirectional in receiving (+) or sending signals (−); or is bidirectional – (±). A port is required (!) or optional (?). The types for directions and necessity are both dependent, each containing the endpoint being projected and the original projected value. These values are not free to choose.

$(\upharpoonright_d)$	?	$?_i$	$?_t$	!
Target	?	!	?	!
Initiator	?	?	!	!

$(\upharpoonright_n)$	⇝	⬳	⬌	$\overline{⬳}$	$\overline{⇝}$
Target	−	+	±	+	−
Initiator	+	−	±	+	−

(a) Necessity.　　　　　　　　**(b)** Direction.

Figure 13 Typing rules for flow and necessity projection.

Figure 13 presents the typing rules that constrain the values in the projected model to predetermined pairings. Dependent types ensure the correctness of projection transformation. The projection for port flow and necessity are defined as functions $((\upharpoonright_d)$ and $(\upharpoonright_n))$ that, given a flow or optional description will compute the required direction – or necessity. Their type signatures are:

- $(\upharpoonright_d) : (d_o : \mathbb{O}_d) \to (e : \mathbb{E}) \to \mathbb{O}\,(e, o_d)$
- $(\upharpoonright_n) : (f : \mathbb{F}) \to (e : \mathbb{E}) \to \mathbb{D}\,(e, f)$

These projection functions will only type check if the given inputs match the allowed pairing of values for the returned type.

$$l : \mathbb{L}(L, k_l)$$

$$\text{PP} \quad \frac{k_p : \mathbb{K}_P \qquad ty : \mathbb{A}(k_p) \qquad d : \mathbb{D}(e, f) \qquad o : \mathbb{O}(e, o_d) \qquad w_d : \mathbb{W}_d(k_p) \qquad s : \mathbb{S} \qquad h : \mathbb{H}}{\mathsf{port}_p(l, k_p, ty, d, o, w_d, s, h) : \mathbb{P}_p(L, e, \mathsf{port}_d(l, k_p, ty, f, o_d, w_d, s, h))}$$

$$\text{PGP-E} \quad \frac{}{\emptyset_p : \mathbb{PG}_p(L, e, \emptyset_d)} \qquad\qquad \text{PGP-C} \quad \frac{p_p : \mathbb{P}_p(L, e, p_d) \qquad ps_p : \mathbb{PG}_p(L, e, ps_d)}{p_p ::_p ps_p : \mathbb{PG}_p(L, e, p_d ::_d ps_d)}$$

$$\text{IP} \quad \frac{c : \mathbb{C}_{\text{style}} \qquad \mathrm{maxI} : \mathbb{N}^* \qquad \mathrm{maxT} : \mathbb{N}^* \qquad ps_p : \mathbb{PG}_p(L, e, ps_d)}{\mathsf{iface}_p(c, \mathrm{maxI}, \mathrm{maxT}, ps_p) : \mathbb{I}_p(L, e, \mathsf{iface}_d(c, \mathrm{maxI}, \mathrm{maxT}, ps_d))}$$

■ **Figure 14** Typing rules for $\theta_{\text{AID}}^{\mathsf{proj}}$.

Figure 14 presents the remaining typing rules for projected interfaces, ports, and port groups. Like the structural definition, the typing rules mirror those for their abstract counterparts, and are indexed by the type associated with labels. However, the types are further indexed by their abstract counterparts, and also by the endpoint the term is being projected under. The θ_{AID} instance provides as a type-level meta-model from which information is sourced. This approach allows for several invariants on the structure of the projected interface to be established.

1. Non-projected values *must* match.
2. Values parameterising the type of projected values are sourced from the abstract description and *must* match.
3. The endpoint that terms are being projected under *must* match.
4. The structure of the projected interface *must* match the structure of the abstract interface.

$$\boxed{\theta_{\text{AID}} \mapsto \theta_{\text{AID}}^{\mathsf{proj}}}$$

$$\mathsf{port}_d(l, k_p, t, d_p, o_p, w_d, s, h) \upharpoonright_p e ::= \mathsf{port}_p(l, k_p, t, (d_p \upharpoonright_d e), (o_p \upharpoonright_n e), w_d, s, h)$$

$$\emptyset_d \upharpoonright_g e ::= \emptyset_p$$

$$p_d ::_d ps_d \upharpoonright_g e ::= (p_d \upharpoonright_p e) ::_p (ps_d \upharpoonright_g e)$$

$$\mathsf{iface}_d(\text{cstyle}, a, b, ps_p) \upharpoonright_i e ::= \mathsf{iface}_p(\text{cstyle}, a, b, (ps_p \upharpoonright_g e))$$

■ **Figure 15** Projection semantics for $\theta_{\text{AID}}^{\mathsf{proj}}$.

Figure 15 presents the projection semantics for projecting θ_{AID} instances to $\theta_{\text{AID}}^{\mathsf{proj}}$ instances. The type signatures for each projection function are omitted for brevity, but they follow those for $(\upharpoonright_d)$ and $(\upharpoonright_n)$. By design, projections and their invariants are well-typed. Malformed projections will fail to type-check, for example if the widths or calculated directions are wrong.

4.1.1 Example

Figure 16 presents example projections for the θ_{AID} instance for Mungo, applied to parameters 32 and 8 using predicated function application. Figure 16a shows a projection for an initiator interface, and Figure 16b shows a projection for a target interface. The set of

$\llbracket\,(\textsc{Mungo}\,\$_{\{32,16\}}32\,\$_{\{8,4\}}8)\,\rrbracket^{\text{AID}}_{\lambda\mapsto\theta}\,\upharpoonright_i\,\textsf{Initiator}$

$\textsf{iface}_p(\textsf{Unicast},1,1,\textsf{port}_p(\textsf{Named}(C),\textsf{WIRE},\textsf{Clock},+,\,?,\mathbb{1}_d,\textsf{High},\textsf{System})$

$::_p\textsf{port}_p(\textsf{Named}(R),\textsf{WIRE},\textsf{Control},-,\,!,\mathbb{1}_d,\textsf{High},\textsf{IP})$

$::_p\textsf{port}_p(\textsf{Named}(W),\textsf{WIRE},\textsf{Control},-,\,!,\mathbb{1}_d,\textsf{High},\textsf{IP})$

$::_p\textsf{port}_p(\textsf{Named}(D),\textsf{ARRAY},\textsf{Data},\pm,\,!,\textsf{w}_d\,(32),\textsf{High},\textsf{IP})$

$::_p\textsf{port}_p(\textsf{Named}(A),\textsf{ARRAY},\textsf{Address},-,\,!,\textsf{w}_d\,(8),\textsf{High},\textsf{IP})$

$::_p\textsf{port}_p(\textsf{Named}(E),\textsf{ARRAY},\textsf{Info},+,\,!,\textsf{w}_d\,(2),\textsf{High},\textsf{IP})$

$::_p\textsf{port}_p(\textsf{Named}(I),\textsf{ARRAY},\textsf{Info},+,\,!,\infty_d,\textsf{High},\textsf{IP})$

$::_p\emptyset_p)$

(a) Mungo projected as an initiator interface.

$\llbracket\,(\textsc{Mungo}\,\$_{\{32,16\}}32\,\$_{\{8,4\}}8)\,\rrbracket^{\text{AID}}_{\lambda\mapsto\theta}\,\upharpoonright_i\,\textsf{Target}$

$\textsf{iface}_p(\textsf{Unicast},1,1,\textsf{port}_p(\textsf{Named}(C),\textsf{WIRE},\textsf{Clock},+,\,?,\mathbb{1}_d,\textsf{High},\textsf{System})$

$::_p\textsf{port}_p(\textsf{Named}(R),\textsf{WIRE},\textsf{Control},+,\,!,\mathbb{1}_d,\textsf{High},\textsf{IP})$

$::_p\textsf{port}_p(\textsf{Named}(W),\textsf{WIRE},\textsf{Control},+,\,!,\mathbb{1}_d,\textsf{High},\textsf{IP})$

$::_p\textsf{port}_p(\textsf{Named}(D),\textsf{ARRAY},\textsf{Data},+,\,!,\textsf{w}_d\,(32),\textsf{High},\textsf{IP})$

$::_p\textsf{port}_p(\textsf{Named}(A),\textsf{ARRAY},\textsf{Address},+,\,!,\textsf{w}_d\,(8),\textsf{High},\textsf{IP})$

$::_p\textsf{port}_p(\textsf{Named}(E),\textsf{ARRAY},\textsf{Info},-,\,?,\textsf{w}_d\,(2),\textsf{High},\textsf{IP})$

$::_p\textsf{port}_p(\textsf{Named}(I),\textsf{ARRAY},\textsf{Info},-,\,?,\infty_d,\textsf{High},\textsf{IP})$

$::_p\emptyset_p)$

(b) Mungo projected as a target interface.

Figure 16 Mungo projected as $\theta^{\text{proj}}_{\text{AID}}$ instances.

directions for each port are mirror images of each other, aside from the constant direction for the system clock and the bi-directional data port. Further, the definition of a port's necessity have been calculated to respect if the port is optional or not. The ports for returning error information are optional if the interface's endpoint is a target interface and required if the endpoint is an initiator.

4.2   Specifying Physical Interfaces

Abstract interfaces, and their projections, represent *descriptions* of a component's interface, we must also model the component itself. Figure 17 presents the terms and types for θ_{COMP} model instances. Figure 18 presents the typing rules.

The structure of concrete interfaces mirrors that of the abstract interface and projection. However, a concrete interface does not have optional ports, within our model dangling ports are not allowed. To model skippable ports the concept of thinnings is used [1]. A thinning allows for structures to be weakened using some decision procedure [13, 2]. We can use this concept to *weaken* the specified ports in an interface's portgroup w.r.t. a given specification. Our decision procedure is simple: a port can be skipped if the projected port is optional. The

$$w : \mathbb{W}(k_p, w_d) ::= \mathsf{u}(i) \mid \mathbb{1} \mid \mathsf{w}(i) \qquad\qquad\qquad \text{Widths}$$

$$p : \mathbb{P}(L, e, p_p) ::= (l, k_p, t, d, w, s, h) \qquad\qquad\qquad \text{Ports}$$

$$ps : \mathbb{PG}(L, e, ps_p) ::= \emptyset \mid p :: ps \mid \sqcup ps \mid p ::\approx ps \qquad\qquad \text{Port Groups}$$

$$i : \mathbb{I}(L, e, i_p) ::= (ps) \qquad\qquad\qquad \text{Interfaces}$$

$$is : \mathbb{IG}(is_d, es) ::= \emptyset_i \mid i ::_i is \qquad\qquad\qquad \text{Interface Group}$$

$$c : \mathbb{C}(xs) ::= \mathsf{comp}(is) \qquad\qquad\qquad \text{Components}$$

$$e_{\mathrm{CSL}} ::= e_{\mathrm{aidl}} \mid w \mid p \mid ps \mid i \mid is \mid c \qquad\qquad \text{Expressions}$$

$$\mathcal{T}_{\mathrm{csl}} ::= \mathcal{T}_{\mathrm{aidl}} \mid \mathbb{W}(k_p, w_d) \mid \mathbb{P}(L, e, p_p) \qquad\qquad \text{Types}$$

$$\mid \mathbb{PG}(L, e, ps_p) \mid \mathbb{I}(L, e, i_p) \mid \mathbb{IG}(is_d, es) \mid \mathbb{C}(xs)$$

Figure 17 Terms for concrete interfaces.

$$\text{W-Z} \ \frac{i : \mathbb{N}^* \qquad \mathsf{shape} : \mathbb{K}_P}{\mathsf{u}(i) : \mathbb{W}(\mathsf{shape}, \infty_d)} \qquad \text{W-O} \ \frac{}{\mathbb{1} : \mathbb{W}(\mathsf{WIRE}, \mathbb{1}_d)} \qquad \text{W-A} \ \frac{i : \mathbb{N}^*}{\mathsf{w}(i) : \mathbb{W}(\mathsf{ARRAY}, \mathsf{w}_d(i))}$$

$$\text{PORT} \ \frac{k_p : \mathbb{K}_P \quad ty : \mathbb{A}(k_p) \quad w : \mathbb{W}(k_p, w_d) \quad s : \mathbb{S} \quad h : \mathbb{H} \quad d : \mathbb{D}(e, f)}{(l, k_p, t, d, w, s, h) : \mathbb{P}(L, e, \mathsf{port}_p(l, k_p, ty, d, o, w_d, s, h))}$$

with $l : \mathbb{L}(L, k_l)$ above.

$$\text{PGP-E} \ \frac{}{\emptyset : \mathbb{PG}_p(L, e, \emptyset_d)} \qquad\qquad \text{PG-C} \ \frac{p : \mathbb{P}(L, e, p_p) \qquad ps : \mathbb{PG}(L, e, ps_p)}{p :: ps : \mathbb{PG}(L, e, p_p :: ps_p)}$$

$$\text{PG-S} \ \frac{ps : \mathbb{PG}(L, e, ps_p)}{\sqcup ps : \mathbb{PG}(L, e, \mathsf{port}_p(l, k_p, t, d_p, ?, w_d, s, h) :: ps_p)}$$

$$\text{PG-SC} \ \frac{p : \mathbb{P}(L, e, p_p) \qquad ps : \mathbb{PG}(L, e, ps_p)}{p ::\approx ps : \mathbb{PG}(L, e, p_p ::_p \mathsf{port}_p(l, k_p, t, d_p, ?, w_d, s, h) ::_p ps_p)}$$

$$\text{INTERFACE} \ \frac{ps : \mathbb{PG}(L, e, ps_p)}{(ps) : \mathbb{I}(L, e, \mathsf{iface}_p(c, \mathrm{maxI}, \mathrm{maxT}, ps_p))} \qquad \text{IG-E} \ \frac{}{\emptyset_i : \mathbb{IG}(\emptyset_d, \emptyset_e)}$$

$$\text{IG-C} \ \frac{i : \mathbb{I}(L, e, i_d \upharpoonright_i e) \qquad is : \mathbb{IG}(is_d, es)}{i ::_i is : \mathbb{IG}(i_d ::_i is_d, e ::_e es)}$$

$$\text{COMPONENT} \ \frac{xs = \{(i_{d,0}, e_0), \dots, (i_{d,j}, e_j)\} \qquad is : \mathbb{IG}(\cup_{k=0}^{j} i_{d,k}, \cup_{k=0}^{j} e_k)}{\mathsf{comp}(is) : \mathbb{C}(xs)}$$

Figure 18 Typing rules for concrete interfaces.

thinning decision procedure does not occur at the value level. Thus a concrete port group is either: empty – $\emptyset$; extended by a port (::); skipped by an optional port ($\sqcup$); or an optional port is skipped when extending the group with a port – (::$\approx$). The operator (::$\approx$) can be defined as the combination of the (::) and ($\sqcup$) operators. That is $p :: (\sqcup ps) \equiv p ::\approx ps$. The typing rules (Figure 18) show how thinning works for interface specifications. The $\theta_{\mathrm{AID}}^{\mathsf{proj}}$ is a type-level invariant, the specification of the necessity of the projected ports is what allows the thinning to occur, or not.

A component is modelled as a collection of interfaces. The type for a component is indexed by a collection (xs) of θ_{AID}-endpoint pairings. The type for a collection of interfaces is indexed by the separated elements of each pair in xs. As a collection of interfaces is constructed, the θ_{AID} instances are projected (at the type-level) by the endpoint type to construct the $\theta_{\mathrm{AID}}^{\mathsf{proj}}$ instance indexing the collected interface. Use of projection at the type-level ensures that the type for a concrete interface is sourced from the specified projection.

In θ_{AID} model instances, wires have width one, arrays have fixed width greater than one, or are unrestricted. When projecting an abstract port, the width does not need to be projected into a local value. However, the port width in a θ_{COMP} instance needs to respect the width in the θ_{AID}. Widths are modelled using a dependent data type that captures, and reasons with, the width of an abstract port.

An instantiated port with an abstract port and an unrestricted width has no restrictions on kind or width. A port with an abstract port of width one *must* also have a width of one. Similarly, a port with an abstract port of fixed width *must* also have the same fixed width.

4.2.1   Example

Figure 19 presents two example interfaces that model Mungo. Figure 19b shows the interface with a port to receive the clock, and Figure 19c without a clock port. Within Figure 19c the skip term ($\sqcup$) allows for the clock to the be skipped. If other required ports were to be skipped this will result in a type error. For both interfaces we have chosen the user defined error message width to be of width 32 and 16. This will not cause a type error as the Mungo allows the signal to have a user-defined width. Further, should other value level information (e.g. port width and label) be incorrectly specified the example will also fail to type-check.

4.3   Type-Checking Interfaces

The type of interfaces in θ_{COMP} are parameterised by projected interfaces from $\theta_{\mathrm{AID}}^{\mathsf{proj}}$. A satisfaction relation is defined to link programs written in λ_{AID} to interfaces from θ_{COMP}.

▶ **Definition 3** (Interface Satisfaction). *Given a λ_{AID} specification $(\varnothing \vdash \nu : 1 \dashv \varnothing)$, and an interface $\vartheta : \mathbb{I}(L, e, \vartheta_p)$ then the interface ϑ satisfies ν, which is defined as $\vartheta \models \nu$, if $[\![\,\nu\,]\!] \upharpoonright_i e \mapsto \vartheta_p$.*

Proof. By construction. The evaluation of ν ($[\![\,\nu\,]\!]$) produces a model ($\vartheta_d : \mathbb{I}_d(L')$), for some ($L' : \mathsf{Type}_L$). The type of ϑ is $\mathbb{I}(L, e, \vartheta_p)$. The projected interface ϑ_p has type $\mathbb{I}_p(L, e, \vartheta'_d)$. If $\vartheta_d \equiv \vartheta'$ and $L \equiv L'$ then $\vartheta_d \upharpoonright_i e \equiv \vartheta'_d \upharpoonright_i e$. If $\vartheta_d \not\equiv \vartheta'_d$ or $L \not\equiv L'$ then ϑ would fail to type check. ◀

5   Implementation

We have realised Cordial using Idris, a general purpose dependently typed programming language [9]. This provides both a practical proof-of-concept implementation, and mechanised formalisation that the framework's type-system holds. The models representing interfaces, port groups, and ports translate directly to standard dependently (and non-dependently)

$$\mathbb{I}\,(L, \mathsf{Initiator}, [\![\,(\textsc{Mungo}\ \$_{\{32,16\}}32\ \$_{\{8,4\}}8)\,]\!]^{\mathrm{AID}}_{\lambda\mapsto\theta}\ \upharpoonright_i \mathsf{Initiator})$$

(a) Type for both Interfaces.

$$(\mathsf{Named}(C), \mathsf{WIRE}, \mathsf{Clock}, +, \mathbb{1}, \mathsf{High}, \mathsf{System})$$
$$::_i (\mathsf{Named}(R), \mathsf{WIRE}, \mathsf{Control}, -, \mathbb{1}, \mathsf{High}, \mathsf{IP})$$
$$::_i (\mathsf{Named}(W), \mathsf{WIRE}, \mathsf{Control}, -, \mathbb{1}, \mathsf{High}, \mathsf{IP})$$
$$::_i (\mathsf{Named}(D), \mathsf{ARRAY}, \mathsf{Data}, \pm, \mathsf{w}\,(32), \mathsf{High}, \mathsf{IP})$$
$$::_i (\mathsf{Named}(A), \mathsf{ARRAY}, \mathsf{Address}, -, \mathsf{w}\,(4), \mathsf{High}, \mathsf{IP})$$
$$::_i (\mathsf{Named}(E), \mathsf{ARRAY}, \mathsf{Info}, +, \mathsf{w}\,(2), \mathsf{High}, \mathsf{IP})$$
$$::_i (\mathsf{Named}(I), \mathsf{ARRAY}, \mathsf{Info}, +, \mathsf{u}\,(32), \mathsf{High}, \mathsf{IP})$$
$$::_i \emptyset_i$$

(b) With a clock port.

$$\sqcup(\mathsf{Named}(R), \mathsf{WIRE}, \mathsf{Control}, -, \mathbb{1}, \mathsf{High}, \mathsf{IP})$$
$$::_i (\mathsf{Named}(W), \mathsf{WIRE}, \mathsf{Control}, -, \mathbb{1}, \mathsf{High}, \mathsf{IP})$$
$$::_i (\mathsf{Named}(D), \mathsf{ARRAY}, \mathsf{Data}, \pm, \mathsf{w}\,(32), \mathsf{High}, \mathsf{IP})$$
$$::_i (\mathsf{Named}(A), \mathsf{ARRAY}, \mathsf{Address}, -, \mathsf{w}\,(4), \mathsf{High}, \mathsf{IP})$$
$$::_i (\mathsf{Named}(E), \mathsf{ARRAY}, \mathsf{Info}, +, \mathsf{w}\,(2), \mathsf{High}, \mathsf{IP})$$
$$::_i (\mathsf{Named}(I), \mathsf{ARRAY}, \mathsf{Info}, +, \mathsf{u}\,(16), \mathsf{High}, \mathsf{IP})$$
$$::_i \emptyset_i$$

(c) Without a clock port.

■ **Figure 19** Sample θ_{COMP} instances that show port skipping.

typed algebraic data structures. λ_{AID}, and its substructural type-system, has been implemented as an Embedded Domain Specific Language (EDSL) using standard techniques [12, Chp. 14]. Predicated functions were realised using Idris' support for *auto implicits* that attempt to search for constructors that can satisfy the type of the implicit variable. The "model construction" process (i.e. $[\![\,l\,]\!]^{\mathrm{AID}}_{\lambda\mapsto\theta}$), again uses standard techniques for working with EDSLs in dependently typed languages [10, 11, 7].

6 Case Studies

Using our implementation we have constructed models for several interaction protocols of varying sizes. We describe the structural properties of the protocols and report on their modeling in CORDIAL as λ_{AID} programs.

6.1 ARMs Advanced Peripheral Bus

The first protocol considered was ARMs *Advanced Peripheral Bus* (APB) [3]. The protocol is a legacy protocol comprising of at least eleven signals, and can be connected to many target IP Cores using an intermediary IP Core called an interconnect. This requires that we construct two specifications one for target connections to the interconnect, and the other for

initiating connections. At least seven signals are required, and two were target optional. At least seven signals have a known width. The data and address width can be up to 32 bits in width. There are three interesting features of `APB` worth noting. First, the specification is parameterised depending on the number of IP Cores that an initiator is connecting too. When connecting to x targets, the signal `PSelx` will be replicated x times in an initiating interface, which will only be seen once in a target interface. Second, the protocol has optional signals for the clock and a signal to enable the clock. The dependency between the two clock signals is not clear from the specification: Can one be skipped when the other is required? Third, the specification requires a set of strobes that connects to every 8th bit on the data bus. This restricts data widths to multiples of eight.

$(\lambda(\text{nrSlaves}) \cdot (\lambda(\text{dwidth}; \{8, 16, 32\}) \; \cdot (\lambda(\text{awidth}; \{8, 16, 32\}) \cdot$

  $\underline{\text{let}} \; x \; \underline{\text{be}} \; (\text{div dwidth } 8) \; \underline{\text{in}}$

  $\underline{\text{let}} \; \alpha \; \underline{\text{be}} \; \text{label(T)} \; \underline{\text{in}} \; \underline{\text{let}} \; \omega \; \underline{\text{be}} \; \text{label(S)} \; \underline{\text{in}}$

  $\cdots$

  $; \; \text{replicate}(\text{nrSlaves}, \text{portDesc}(\alpha, \text{WIRE}, \text{Info}, \rightsquigarrow, !, \mathbb{1}_d, \text{High}, \text{IP}))$

  $; \; \text{portDesc}(\omega, \text{ARRAY}, \text{Info}, \leftsquigarrow, ?, \text{w}_d \, (x), \text{High}, \text{IP})$

  $; \; \ldots)))$

Figure 20 Partial presentation of the "Master Interface" for the `APB` specification.

We chose to implement the specifications using two predicated functions, and one normal function. The two predicated functions was used to ensure that all bus widths are multiples of eight and less than 32 i.e. 8, 16, and 32. The normal function was used to represent the number of targets. Specification of labels and ports followed the known information taken from the specification. The term `replicate` was used to ensure that the correct number of `PSelx` signals were generated. For strobes, the maths operations were used to calculate the number of strobes required, based of the size of the data bus. Figure 20 shows part of the `APB` specification detailing the predicated functions, use of replicate, and strobe specifcation.

6.2 Xilinx's LocalLink

The second protocol chosen was a legacy protocol from Xilinx called `LocalLink` [41]. The `LocalLink` protocol requires seven signals and thirteen optional signals. As with the `APB` protocol many of the signals were encoded without complications. However, and unlike `APB`, `LocalLink` does not specify a set of bus widths and requires that the presented bus width is a multiple of eight. Predicated functions require that the list of possible values are known *a priori*. Thus, we modelled the specification as a predicated function on bus widths of 8, 16, and 32, together with a normal function that takes the number of channels.

The `LocalLink` specification contains various size dependencies based on the data bus. The size of the remainder bus is dependent on how they are implemented within the IP Core. Specifically, if the remainder is encoded then the bus size will be $\lceil \log_2 \frac{d}{8} - 1 \rceil$. If the remainder is masked then the size is $\frac{d}{8} - 1$. Our framework is not expressive enough to encode these differences mathematically, we support simple arithmetic operations and these operations are not simple. Moreover, our framework does not support related specifications that overlap to be collapsed into a single definition. The `LocalLink` specification is *too* value dependent.

Another interesting aspect of the `LocalLink` specification is that of *channels*. These are a set of optional signals whose flow is dependent on the application context. The size of *channel* related signals are too calculated using a floating point operation. These three

signals are directional, and whether they send data from target to initiator is dependent on the application, and how the other two signals are specified. Although we can represent these channels as being bidirectional our language is not expressive enough to capture these application specific, and inter signal, dependencies.

6.3 ARMs Advanced eXtensible Interface

ARMs *Advanced eXtensible Interface* (`AXI`) is a widely used family of interaction protocols [4] for transferring addressable data between IP Cores. There are several previous versions of `AXI` each building upon previous versions. Each version differs by number of signals and changes to specific signal properties. The `AXI` specification defines three protocols that offer three different interaction styles: Full, Lite, and Stream. The protocol can be directly connected to another interface, or it can be used to connect to multiple other IP Cores using an interconnect. Version four of the protocol requires 47 signals comprising of: two global signals for the clock and a reset; thirteen signals each for specifying writing and reading of an address; seven signals for reading and writing data; and five signals for writing responses from the target to the initiator. Of the 47 signals, 36 are required and eleven are either optional, target optional, or initiator optional. Several signals have user defined widths. The `AXI` protocol is parameterised such that the address and data busses can be: 8, 16, 32, 64, 128, 256, 512, or 1024 bits wide. Further, the protocol specification supports custom sets of signals that are completely user definable. In fact the `AXI` standard explicitly warns against their use due to potential interoperability issues if different modules present user-defined signals that behave differently.

We report on describing version four of the `AXI` protocol for direct interfaces only. For versions of the protocol that connect to an interconnect, the techniques presented in Section 6.1, can be leveraged. The specification was modelled in λ_{AID} using two predicated functions to control the width of the address and data busses. Each of the signals were translated into the required port descriptions. Like the `APB` protocol, `AXI` has the concept of strobes. The same technique as used in Section 6.1 was used.

We chose not to include user-defined signals in this study. Cordial requires that signal details are known *a priori*. Ideally, designers would create parameterised specifications (functions) that take as parameters the user-defined signals. However, functions in λ_{AID} take pure values as parameters, port declarations modify the typing environment to update label usage information and are thus not pure. This is a restriction from the substructural type-system for λ_{AID} – see Section 7.3.

The protocol specification divides the signals among several channels. Such a grouping is only required for the specification. At the module level these groupings do not exist. Although, Cordial does not support this grouping incorporating this into the framework would be a potential benefit for reasoning about subgroupings of an interface's portgroup.

7 Discussion & Related Work

This section discusses the efficacy of the framework, and related work.

7.1 Discussion

Section 6 described three case studies that modelled three interaction protocols using λ_{AID}. For each of the protocols we were able to encode most of the ports correctly Cordial is suitable for capturing each port's values. However, both `LocalLink` and `APB` illustrated the limitations of λ_{AID} in capturing all of the specification's dependencies.

While CORDIAL can capture limited value dependencies, for example strobes and number of targets within `APB`, the framework prohibits the construction of concise descriptions based on other value dependencies. Specifically, for ports whose direction is dependent on a mode of operation (`LocalLink`), and how to version protocols – `AXI` versions 1–4. Although we can write multiple different versions, a dependently typed construction should be able to capture these properties concisely, and without resorting to *copying and pasting* protocol specifications. We need to explore what other dependencies there are within a protocol specification, how prevalent these dependencies are, and how we can capture and reason about the relevant dependent properties.

The construction of λ_{AID} exposes the resource tracking of the type systems directly as resources are associated with variables. This does leads to a more verbose language such that for *n* signals there will be *n* variables, and *n* additional ports declared. This is not optimal. An alternative approach would be to embedd the resource tracking directly in our monad's type to remove the need for variables–cf. Swiestra's Hoare & Atkey's parameterised monads [37, 5]. However, our current formulation is more extensible allowing arbitrary new states to be added by indexing the type of new variables.

The type-level resource tracking in λ_{AID} also prohibits the creation of higher-order descriptions. The typing rule LAM requires a pure value, and sequencing using LET and SEQ require that the *knowledge* contained in the environment is passed from the previous construct to the next. This is a limitation of the Hoare Monad used to sequencing expressions.

7.2   Modelling Hardware Interfaces

Many attempts at reasoning about hardware have centred on formalising hardware systems as a collection of digital circuits and capturing the behaviour of signals through the specified circuits. Ghica et al. exploited category theory to investigate connection of components [18, 17, 19]. EDSLs have been developed for Haskell such as Lava [20] and Cλash [35] that take other mathematical approaches to reasoning about hardware behaviour. ΠWare utilises dependent types to reason about hardware [15, 16]. Vijayaraghavan et al. [38] presents a complete formalisation of the behaviour of SoC designs, however, their approach does not look at the validation of interfaces against a specification, and concentrates on modelling the behaviour of components as a distributed system. Our framework complements existing work by providing guarantees about the physical structure of a component's ports.

Tooling such as Vivado IP Integrator [39] and Kactus2 [24] can automatically construct, and connect, components in a SoC architecture correctly. Such tooling is based on `IP-XACT` and vendor extensions. Examination of the Vivado toolchain reveals handwritten TCL scripts bespoke for the `AXI` family of protocols. Our work presents a specification agnostic framework for type-checking hardware interfaces against a richer specification than seen with `IP-XACT` solutions. We position our work as possible foundation for machine derivable code to develop richer integration and construction checks to that seen with IP Integrator and Kactus2.

Click is an untyped SoC design language for describing the routing of data [25]. McKechnie [30] developed a type-system for typing the interconnections found within Click specifications. Our work provides a natural extension to McKechnie's work and provides a means to type components in a Click design against external specifications.

7.3   Substructural Typing

The substructural type-system for λ_{AID} is based upon Hoare logic [5, 8]. Unfortunately, Hoare logics do not support the *frame rule*, a means to divide and share invariants in a composable manner. This results in λ_{AID} not being able to support higher-order descriptions.

Separation Logic is an advancement that does support the frame rule [34], and has been used to construct substructural type-systems for EDSLs [26], However, it is not clear how straightforward it would be to realise such a type-system for an EDSL within a dependently typed language.

There are other formal models upon which one can realise substructural type-systems for EDSLs, namely TypeStates [31], and Refinement Types from Hoare Types [8, 32]. All allow for reasoning about type-level resource usage protocols, however, how straightforward these models can be realised within a dependently typed language is not clear.

λ_{AID} was realised as an EDSL, perhaps realising it as a standalone Domain Specific Language (DSL) written in Idris might allow for Idris' rich type-system to better realise the substructural typing for λ_{AID}. Future work will be to investigate how to realise, and implement, the substructural typing for λ_{AID}.

7.4 Implementing Cordial

CORDIAL has been implemented within Idris. Any other dependently typed language that supports full-spectrum dependent types, such as Agda [33], would also be suitable host language.

Although CORDIAL uses dependent type theory and substructural typing, non-dependently type languages can also realise the framework. The ideas are transferable, but the implementation would not be as clean nor concise. Racket is a general purpose language that supports EDSL creation through fine-grained control over the language's type-system [14]. $F^\star$ is a general purpose language with value-dependent types [36]. Whereas Idris provides full-spectrum dependent types, $F^\star$ provides value-dependencies using refinement types. This provides a novel, alternate, environment in which to construct "value-dependently-typed" programs. How the approach behind CORDIAL is transferable to these languages is worth investigating.

8 Conclusion

We presented a framework (CORDIAL) to provide correct-by-construction guarantees over interface specifications in SoC designs. We have demonstrated use of the framework to model real world protocols, and noted limitations in the models expressiveness and future work to enrich said expressiveness. There are other areas for future work:

Checking Existing Systems

Our approach lends itself well to the *generation* of designs from model instances. We can easily extend our Idris implementation to generate stubs for various HDLs. However, how do we evaluate existing interfaces? To do so, not only do we need to be able to extract interface model descriptions from existing HDL code, but also associate these model descriptions with abstract interface model descriptions. That is, we need to be able to infer from a component specification the concrete interfaces, and for those interfaces their abstract descriptions and which characterisation corresponds to the found interface. The problem of model inference is difficult as component interfaces are not always cleanly defined. Multiple interfaces for a component can be presented as a flat port group, sending ports can send to multiple recipients, and the ordering of ports does not necessarily reflect the ordering in the specification. Further, the names given to ports may not match the labels described in the specification. Developers, and code generation tools, have complete freedom in structuring their components interfaces. Further work will be to explore how to infer models from such "messy" SoC descriptions.

Enriching existing HDL

We have formalised CORDIAL in an existing general purpose language. A more interesting area for future work, and to increase adoption of the ideas mentioned, would be to develop extensions for various HDL such as SystemVerilog with the presented framework. A similar approach would be to extend existing design environments such as Vivado from Xilinx to incorporate our tooling and ideas.

Checking Behaviour

Our solution reasons about the *structural correctness* of SoC architectures. These provided guarantees are a design time check. Standards documents also describe a protocol's *behavioural correctness*. Our models do not capture a component's behaviour, a correctly connected component may show incorrect behaviour (as described in the specification) at run time. We saw this when modelling the `AXI` family of protocols. CORDIAL borrows notions of global and local projections from *Session Types*. We could also look to use Session Types to reason about hardware behaviour. While there have been attempts at extending Session Types to fit communication models similar to those found in hardware [27], none have been directly applied to checking hardware. Future work will be to explore how we can extend our model descriptions to capture the behaviour of a component's interface.

Modelling complete SoC architectures

SoC designs are about connecting components. A natural extension to our work would be to provide an orchestration language that uses θ_{COMP} to model components and their connections. Existing work has investigated verifying IP Core connections using static typing to ensure substructural properties of a SoC design hold – Section 7.2. Integrating the work of McKechnie [30] into the expressive type-system of our framework can serve as the basis for a more complete solution to SoC design. We leave this aspect of future work as an open problem.

─── **References** ───

1   Guillaume Allais, Robert Atkey, James Chapman, Conor McBride, and James McKinna. A type and scope safe universe of syntaxes with binding: their semantics and proofs. *PACMPL*, 2(ICFP):90:1–90:30, 2018. `doi:10.1145/3236785`.

2   Thorsten Altenkirch, Martin Hofmann, and Thomas Streicher. Categorical Reconstruction of a Reduction Free Normalization Proof. In David H. Pitt, David E. Rydeheard, and Peter T. Johnstone, editors, *Category Theory and Computer Science, 6th International Conference, CTCS '95, Cambridge, UK, August 7-11, 1995, Proceedings*, volume 953 of *Lecture Notes in Computer Science*, pages 182–199. Springer, 1995. `doi:10.1007/3-540-60164-3_27`.

3   ARM Limited. *AMBA APB Protocol*, ARM IHI 0024C edition, 2010.

4   ARM Limited. *AXI and ACE Protocol Specification*, ARM IHI 0022F.b edition, 2017.

5   Robert Atkey. Parameterised notions of computation. *J. Funct. Program.*, 19(3-4):335–376, 2009. `doi:10.1017/S095679680900728X`.

6   Robert Atkey. Syntax and Semantics of Quantitative Type Theory. In Anuj Dawar and Erich Grädel, editors, *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018*, pages 56–65. ACM, 2018. `doi:10.1145/3209108.3209189`.

7   Lennart Augustsson and Magnus Carlsson. An Exercise in Dependent Types: A Well-Typed Interpreter. In *In Workshop on Dependent Types in Programming, Gothenburg*, 1999.

8 Johannes Borgström, Juan Chen, and Nikhil Swamy. Verifying stateful programs with substructural state and hoare types. In Ranjit Jhala and Wouter Swierstra, editors, *Proceedings of the 5th ACM Workshop Programming Languages meets Program Verification, PLPV 2011, Austin, TX, USA, January 29, 2011*, pages 15–26. ACM, 2011. `doi:10.1145/1929529.1929532`.

9 Edwin Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *J. Funct. Program.*, 23(5):552–593, 2013. `doi:10.1017/S095679681300018X`.

10 Edwin Brady. Programming and reasoning with algebraic effects and dependent types. In Greg Morrisett and Tarmo Uustalu, editors, *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*, pages 133–144. ACM, 2013. `doi:10.1145/2500365.2500581`.

11 Edwin Brady. Resource-Dependent Algebraic Effects. In Jurriaan Hage and Jay McCarthy, editors, *Trends in Functional Programming - 15th International Symposium, TFP 2014, Soesterberg, The Netherlands, May 26-28, 2014. Revised Selected Papers*, volume 8843 of *Lecture Notes in Computer Science*, pages 18–33. Springer, 2014. `doi:10.1007/978-3-319-14675-1_2`.

12 Edwin Brady. *Type-Driven Development with Idris.* Manning, 1st edition, 2016.

13 James Maitland Chapman. *Type checking and normalisation.* PhD thesis, School of Computer Science, University of Nottingham, July 2009. URL: `http://eprints.nottingham.ac.uk/10824/`.

14 Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi, Eli Barzilay, Jay A. McCarthy, and Sam Tobin-Hochstadt. A programmable programming language. *Commun. ACM*, 61(3):62–71, 2018. `doi:10.1145/3127323`.

15 J Pizani Flor. π-Ware: An Embedded Hardware Description Language using Dependent Types. Masters, Department of Information and Computing Sciences, 2014. URL: `https://dspace.library.uu.nl/bitstream/handle/1874/298576/`.

16 João Paulo Pizani Flor, Wouter Swierstra, and Yorick Sijsling. Pi-Ware: Hardware Description and Verification in Agda. In Tarmo Uustalu, editor, *21st International Conference on Types for Proofs and Programs, TYPES 2015, May 18-21, 2015, Tallinn, Estonia*, volume 69 of *LIPIcs*, pages 9:1–9:27. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015. `doi:10.4230/LIPIcs.TYPES.2015.9`.

17 Dan R. Ghica. The Geometry of Synthesis - How to Make Hardware Out of Software. In Jeremy Gibbons and Pablo Nogueira, editors, *Mathematics of Program Construction - 11th International Conference, MPC 2012, Madrid, Spain, June 25-27, 2012. Proceedings*, volume 7342 of *Lecture Notes in Computer Science*, pages 23–24. Springer, 2012. `doi:10.1007/978-3-642-31113-0_3`.

18 Dan R. Ghica, Achim Jung, and Aliaume Lopez. Diagrammatic Semantics for Digital Circuits. In Valentin Goranko and Mads Dam, editors, *26th EACSL Annual Conference on Computer Science Logic, CSL 2017, August 20-24, 2017, Stockholm, Sweden*, volume 82 of *LIPIcs*, pages 24:1–24:16. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017. `doi:10.4230/LIPIcs.CSL.2017.24`.

19 Dan R. Ghica, Alex I. Smith, and Satnam Singh. Geometry of synthesis iv: compiling affine recursion into static hardware. In Manuel M. T. Chakravarty, Zhenjiang Hu, and Olivier Danvy, editors, *Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming, ICFP 2011, Tokyo, Japan, September 19-21, 2011*, pages 221–233. ACM, 2011. `doi:10.1145/2034773.2034805`.

20 Andy Gill, Tristan Bull, Garrin Kimmell, Erik Perrins, Ed Komp, and Brett Werling. Introducing Kansas Lava. In Marco T. Morazán and Sven-Bodo Scholz, editors, *Implementation and Application of Functional Languages - 21st International Symposium, IFL 2009, South Orange, NJ, USA, September 23-25, 2009, Revised Selected Papers*, volume 6041 of *Lecture Notes in Computer Science*, pages 18–35. Springer, 2009. `doi:10.1007/978-3-642-16478-1_2`.

21 John Hatcliff and Olivier Danvy. A Generic Account of Continuation-Passing Styles. In Hans-Juergen Boehm, Bernard Lang, and Daniel M. Yellin, editors, *Conference Record of POPL'94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland,*

Oregon, USA, January 17-21, 1994, pages 458–471. ACM Press, 1994. `doi:10.1145/174675.178053`.

22 Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. In George C. Necula and Philip Wadler, editors, *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*, pages 273–284. ACM, 2008. `doi:10.1145/1328438.1328472`.

23 IEEE. *IEEE Standard for IP-XACT, Standard Structure for Packaging, Integrating, and Reusing IP within Tool Flows*, ieee std 1685-2014 edition, September 2014. `doi:10.1109/IEEESTD.2014.6898803`.

24 Antti Kamppi, Lauri Matilainen, Joni-Matti Määttä, Erno Salminen, Timo D. Hämäläinen, and Marko Hännikäinen. Kactus2: Environment for Embedded Product Development Using IP-XACT and MCAPI. In *14th Euromicro Conference on Digital System Design, Architectures, Methods and Tools, DSD 2011, August 31 - September 2, 2011, Oulu, Finland*, pages 262–265. IEEE Computer Society, 2011. `doi:10.1109/DSD.2011.36`.

25 Eddie Kohler, Robert Tappan Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. The click modular router. *ACM Trans. Comput. Syst.*, 18(3):263–297, 2000. `doi:10.1145/354871.354874`.

26 Neelakantan R. Krishnaswami, Aaron Turon, Derek Dreyer, and Deepak Garg. Superficially substructural types. In Peter Thiemann and Robby Bruce Findler, editors, *ACM SIGPLAN International Conference on Functional Programming, ICFP'12, Copenhagen, Denmark, September 9-15, 2012*, pages 41–54. ACM, 2012. `doi:10.1145/2364527.2364536`.

27 Julien Lange, Nicholas Ng, Bernardo Toninho, and Nobuko Yoshida. A static verification framework for message passing in Go using behavioural types. In Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman, editors, *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, pages 1137–1148. ACM, 2018. `doi:10.1145/3180155.3180157`.

28 Per Martin-Löf and Giovanni Sambin. *Intuitionistic Type Theory*. Bibliopolis, 1984.

29 Conor McBride. I Got Plenty o' Nuttin'. In Sam Lindley, Conor McBride, Philip W. Trinder, and Donald Sannella, editors, *A List of Successes That Can Change the World - Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*, volume 9600 of *Lecture Notes in Computer Science*, pages 207–233. Springer, 2016. `doi:10.1007/978-3-319-30936-1_12`.

30 Paul Edward McKechnie. *Validation and verification of the interconnection of hardware intellectual property blocks for FPGA-based packet processing systems*. PhD thesis, University of Glasgow, 2010. URL: `http://theses.gla.ac.uk/1879/`.

31 Filipe Militão, Jonathan Aldrich, and Luís Caires. Substructural typestates. In Nils Anders Danielsson and Bart Jacobs, editors, *Proceedings of the 2014 ACM SIGPLAN Workshop on Programming Languages meets Program Verification, PLPV 2014, January 21, 2014, San Diego, California, USA, Co-located with POPL '14*, pages 15–26. ACM, 2014. `doi:10.1145/2541568.2541574`.

32 Aleksandar Nanevski, J. Gregory Morrisett, and Lars Birkedal. Hoare type theory, polymorphism and separation. *J. Funct. Program.*, 18(5-6):865–911, 2008. `doi:10.1017/S0956796808006953`.

33 Ulf Norell. Dependently typed programming in Agda. In Andrew Kennedy and Amal Ahmed, editors, *Proceedings of TLDI'09: 2009 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation, Savannah, GA, USA, January 24, 2009*, pages 1–2. ACM, 2009. `doi:10.1145/1481861.1481862`.

34 John C. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings*, pages 55–74. IEEE Computer Society, 2002. `doi:10.1109/LICS.2002.1029817`.

35 Gerard J. M. Smit, Jan Kuper, and Christiaan P. R. Baaij. A mathematical approach towards hardware design. In Peter M. Athanas, Jürgen Becker, Jürgen Teich, and Ingrid

Verbauwhede, editors, *Dynamically Reconfigurable Architectures, 11.07. - 16.07.2010*, volume 10281 of *Dagstuhl Seminar Proceedings*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany, 2010. URL: `http://drops.dagstuhl.de/opus/volltexte/2010/2840/`.

36 Nikhil Swamy, Juan Chen, Cédric Fournet, Pierre-Yves Strub, Karthikeyan Bhargavan, and Jean Yang. Secure distributed programming with value-dependent types. *J. Funct. Program.*, 23(4):402–451, 2013. `doi:10.1017/S0956796813000142`.

37 Wouter Swierstra. A Hoare Logic for the State Monad. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings*, volume 5674 of *Lecture Notes in Computer Science*, pages 440–451. Springer, 2009. `doi:10.1007/978-3-642-03359-9_30`.

38 Muralidaran Vijayaraghavan, Adam Chlipala, Arvind, and Nirav Dave. Modular Deductive Verification of Multiprocessor Hardware Designs. In Daniel Kroening and Corina S. Pasareanu, editors, *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part II*, volume 9207 of *Lecture Notes in Computer Science*, pages 109–127. Springer, 2015. `doi:10.1007/978-3-319-21668-3_7`.

39 Product page for Vivado IP Integrator by Xilinx. Online, 2019. URL: `https://www.xilinx.com/products/design-tools/vivado/integration.html`.

40 David Walker. *Advanced Topic in Types and Programming Languages*, chapter Substructural Type Systems, pages 3–43. The MIT Press, 2004.

41 Xilinx. *LocalLink Interface Specification*, SP006 (v2.0) edition, 2005.