Language-based Hardware Communication Safety and Liveness

Aditya Ranjan Jha*
National University of Singapore
Singapore

Jason Zhijingcheng Yu*
National University of Singapore
Singapore

Umang Mathur
National University of Singapore
Singapore

Trevor E. Carlson
National University of Singapore
Singapore

Prateek Saxena
National University of Singapore
Singapore

ABSTRACT

Emerging hardware description languages (HDLs) provide new type annotations to abstract from low-level details [3, 8, 10, 11, 13], enabling opportunities for modular and scalable structural verification at design stage. However, verification of behavioural properties among communicating hardware modules—such as ensuring liveness and communication safety—remain largely unsupported in modern HDLs. In this paper, we present **Pact**, a language extending a recent HDL called Anvil [12] with a novel session type-based system. Our proposed type system explicitly models synchronous clock cycles and introducing new type constructs to capture communication contracts between hardware modules. It offers abstractions to reason about liveness and communication safety between hardware components. Additionally, we give a sound and complete algorithm for verifying them statically. *Our evaluation shows that our type abstractions offer a factor of* 2^{70} *to* 2^{270} *in state space reduction over that of concrete RTL designs, when verifying* 10 *real hardware designs.*

1 INTRODUCTION

Hardware designs typically consist of numerous concurrent modules that communicate with one another. In such communication, a sender shares a value with a receiver in a particular clock cycle. Two properties are highly desirable: (1) *communication safety*—the time the sender intends to send the value and the time the receiver intends to receive the value are synchronized; (2) *liveness*—all hardware modules eventually make progress, and in particular, do not deadlock for message communication.

Traditionally, such properties are verified post design through testing or model checking, which suffers from unsoundness and state explosion. While recent research has started investigating language-based approaches that allow sound design-time hardware verification, the properties studied so far have been limited to local low-level properties surrounding wire and register use, such as timing safety, combinational loop freedom, structural hazard freedom, and so on. Verifying communication safety and liveness, however, requires reasoning about the overall behaviour of modules. In this work, we aim to fill in this gap and propose a language-based approach for verifying communication safety and liveness in hardware. The result of our effort is Pact, a hardware description language (HDL) with a type system for verifying such properties.

A natural option is to introduce multiparty session types, which have been used to verify similar properties for software systems, to HDLs. Session types capture relevant behaviour of communication participants. When a collection of session types are compatible, they guarantee communication safety and deadlock freedom. However, adapting session types is not straightforward due to a fundamental mismatch between existing session type abstractions and the execution model of synchronous digital hardware. Specifically, such a mismatch requires us to overcome three main challenges: (1) Existing session types [4, 6, 7] typically abstract away cycle-level timing hardware communication; (2) They restrict internal and external choice structures, limiting the expressiveness for communication patterns common to hardware designs; (3) Prior approaches to session types define compatibility syntactically [1, 5] or compromise modularity [2, 9], which is essential for hardware design.

Our work, Pact, addresses these challenges first by extending session types with modelling for synchronous hardware behaviour such as synchronous latency and unrestricted multiparty choice. We then devise an automata-based type compatibility algorithm that is sound and complete, overcoming the limitations of existing type compatibility formulations. We implement Pact as an extension to Anvil, and evaluate its effectiveness. Our experiments across 10 real-world hardware designs show that compared to concrete register-transfer-level (RTL) designs, Pact achieves a state space reduction by a factor of 2^{70} to 2^{270} across different types of designs. Once the development has matured, Pact will be open-sourced and merged into the upstream Anvil.

*Equal Contribution

$$e ::= x \mid v \mid e\ \texttt{bop}\ e \qquad v ::= \texttt{true} \mid \texttt{false} \qquad B ::= \emptyset \mid B, \hat{a} : P$$

$$a ::= \texttt{send}\ m\ e \mid x = \texttt{recv}\ m\ \texttt{in}\ P \qquad \hat{a} ::= \texttt{send}\ m\ e \mid x = \texttt{recv}\ m$$

$$\begin{aligned} P ::= &\ \texttt{skip} \mid a \mid X \mid \texttt{cycle} \mid \texttt{end}_t \mid P \parallel P \mid P; P \mid \texttt{let}\ x = e\ \texttt{in}\ P \\ &\mid \texttt{if}\ e\ \texttt{then}\ P\ \texttt{else}\ P \mid \texttt{fair-if}_\eta\ e\ \texttt{then}\ P\ \texttt{else}\ P \\ &\mid \texttt{offer}\ B\ \texttt{else}\ P \mid \texttt{xoffer}\ B\ \texttt{else}\ P \mid \mu X.P \end{aligned}$$

Figure 1: Abstract syntax of processes in Pact.

2 PACT

Following Anvil, Pact models hardware modules as *processes* which communicate by passing messages across *channels.* Such *message passing* is the only way for processes to communicate.

2.1 Processes

2.1.1 Syntax. Figure 1 presents the abstract syntax of Pact processes. In addition to the basic primitives cycle and message passing primitives in Anvil, Pact includes explicit constructs for more expressive communication patterns, particularly for multiparty external choice, and for expressing safety specification.

The exclusive (`xoffer`) and non-exclusive (`offer`) constructs both default to $P_{\texttt{else}}$ if no messages in B can be exchanged. However, if multiple matches exist, `xoffer` non-deterministically executes one, while `offer` executes all in parallel. These unique constructs enable external choices dependent on multiple parties.

Some lightweight constructs help specify aspects of the desired safety properties. For example, consider the branching construct ($\texttt{fair-if}_\eta\ e\ \texttt{then}\ P_1\ \texttt{else}\ P_2$) that is *assumed* to be fair, i.e., either neither or both of P_1 and P_2 are executed infinite times. Here, the identifier η allows us to track which branching construct each branching decision corresponds to in the semantics. Likewise, $\texttt{end}_t$ is used to specify a state of interest, which when reached demarcates progress in the process identified by t. In Pact, the RTL design consists of a set of looping processes $P_1, P_2, \cdots, P_n$ provided by the designer. Our safety specification requires that all processes perform infinitely many loop iterations. Therefore, Pact designs are expected to be structured as

$$\mu X.(P_1; \texttt{end}_1; X) \parallel \mu X.(P_2; \texttt{end}_2; X) \parallel \ldots \parallel \mu X.(P_n; \texttt{end}_n; X).$$

2.1.2 Semantics. The semantics of Pact processes is defined using a labelled transition system (LTS) (S, Σ, T), where each transition label $\sigma \in \Sigma$ represents an *action*:

$$\sigma ::= m\checkmark \mid \# \mid \$_t \mid \texttt{true}_\eta \mid \texttt{false}_\eta.$$

An $m\checkmark$ action represents a message exchange over channel m and # represents elapse of a cycle. A $\$_t$ marks reaching of a state that indicates progress, referred to as a *progress state*, e.g., completion of a process iteration, and $\texttt{true}_\eta$ and $\texttt{false}_\eta$ encode choice decisions.

Each state in S is a process (note that a collect of parallel processes can also be expressed as one process). The run of such an LTS from a state s_0 is a finite/infinite sequence $\rho = (s_0 \xrightarrow{\sigma_0} s_1) \cdot (s_1 \xrightarrow{\sigma_1} s_2) \cdots$ such that for every i, $(s_i \xrightarrow{\sigma_i} s_{i+1}) \in T$. A run is *complete* if it is either infinite or if it terminates in a state s with no outgoing transition.

2.2 Safety

We define safety based on the LTS. Such a definition needs to capture both communication safety and liveness (under specified fairness assumptions). Informally, the definition requires that all complete run of the LTS from a given start state to satisfy either (1) for all (relevant) process identifiers t, $\$_t$ transitions are made infinite times in the run, or (2) for some η, $\texttt{true}_\eta$ or $\texttt{false}_\eta$ transitions are made infinite times, and transitions of the other are made finite times. Note that (1) states progress and (2) states the specified fairness assumptions. Communication safety is implicit: The LTS only allows transitions corresponding to safe communication, and the conditions above imply that the run must be infinite.

2.3 Types

Compared to prior session type formulations, Pact types capture more expressive communication patterns, and use a compatibility checking algorithm that is sound and complete (Section 2.4).

The abstract syntax of types is as follows:

$$\tau ::= \alpha \mid \$_t \mid \# \mid \texttt{skip} \mid \tau + \tau \mid \tau; \tau \mid \tau \parallel \tau \mid X \mid \mu X.\tau \mid \tau \dotplus_\eta \tau$$
$$\mid \oplus(\tau, \alpha : \tau, \cdots) \mid \wedge(\tau, \alpha : \tau, \cdots)$$
$$\alpha ::= m! \mid m?$$

Table 1: Comparison of state space: RTL vs type ($\log_2(|N|)$)

Component	Participants	RTL State Space	Type State Space	Compatiblity
FIFO	3	81	9.7	✓
Stream FIFO	3	80	8.8	✓
Spill Register	3	70	8.8	✓
RR Arbiter	6	176	18.5	✓
TileLink Crossbar	10	184	34.5	✓
ACE Cache Controller	11	241	39.9	✓
AXI Lite Demux (Read)	8	155	27.4	✓
AXI Lite Demux (Write)	10	144	38.5	✓
AXI Lite Mux (Read)	9	226	33.2	✓
AXI Lite Mux (Write)	11	311	41.7	✓

Similar to processes, types include internal choice with fairness assumptions ($\tau \dotplus_\eta \tau$), exclusive and non-exclusive multiparty external choice ($\oplus(\tau, \alpha : \tau, \cdots)$ and $\wedge(\tau, \alpha : \tau, \cdots)$), as well as the marker for a progress state ($\$_t$).

The satisfiability relation between a type and a process is mostly standard, defined inductively over the syntactic structure. We define the operational semantics of types in the same framework (Section 2.1.2), except that the states are here types instead of processes.

2.4 Type Compatibility

We devise an automata-based sound and complete algorithm for deciding compatibility between types. The high-level idea of the algorithm is straightforward. We start by showing that starting with any type, the set of reachable types in the type semantics is finite (under the restriction that all recursions are at tail positions). This allows us to construct a *finite* LTS which captures the semantics of $\tau_{\text{system}} = \tau_1 \parallel \cdots \parallel \tau_n$. We then observe that the safety properties (Section 2.2) can be encoded as Rabin conditions for ω-automata. This allows us to apply model checking on types, which can be achieved through checking for emptiness of the product Rabin automaton. Since types abstract away many irrelevant details in concrete designs, we expect the LTS thus constructed to be smaller in state space. We show this empirically in Section 4.

3 SOUNDNESS

At a high level, we prove soundness—if τ_i satisfies P_i and $\tau_{\text{system}} = \tau_1 \parallel \cdots \parallel \tau_n$ is compatible, then $P_{\text{system}} = P_1 \parallel \cdots \parallel P_n$ is safe—by showing that (1) satisfiability between any τ and P implies that the semantics of P refines that of τ, thus safety of τ transfers to P, and (2) compatibility of τ implies its safety.

4 EVALUATION

We evaluate expressivity using open-source IPs, comparing the state space of our behavioural type models (product of automata states) against SystemVerilog implementations (via yosys-abc latch counts). Table 1 demonstrates orders-of-magnitude reductions in state bits ($\log_2 |N|$).

5 CONCLUSION

In this work, we present Pact, an HDL for verifying communication safety and liveness. We address the expressiveness limits of multiparty session types for synchronous hardware and provide a sound, complete compatibility algorithm. Evaluations on end-to-end designs show orders-of-magnitude state space improvement compared to concrete RTL.

REFERENCES

[1] Marco Carbone, Sonia Marin, and Carsten Schürmann. 2023. A Logical Interpretation of Asynchronous Multiparty Compatibility. In *Logic-Based Program Synthesis and Transformation: 33rd International Symposium, LOPSTR 2023, Cascais, Portugal, October 23-24, 2023, Proceedings* (Cascais, Portugal). Springer-Verlag, Berlin, Heidelberg, 99–117. https://doi.org/10.1007/978-3-031-45784-5_7

[2] Giuseppe Castagna, Mariangiola Dezani-Ciancaglini, and Luca Padovani. 2011. On Global Types and Multi-party Sessions. In *Formal Techniques for Distributed Systems*, Roberto Bruni and Juergen Dingel (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1–28.

[3] Michael Christensen, Timothy Sherwood, Jonathan Balkind, and Ben Hardekopf. 2021. Wire sorts: a language abstraction for safe hardware composition. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI)*. Association for Computing Machinery, New York, NY, USA, 175–189. https://doi.org/10.1145/3453483.3454037

[4] Pierre-Malo Deniélou and Nobuko Yoshida. 2012. Multiparty session types meet communicating automata. In *Proceedings of the 21st European Conference on Programming Languages and Systems* (Tallinn, Estonia) *(ESOP'12)*. Springer-Verlag, Berlin, Heidelberg, 194–213. https://doi.org/10.1007/978-3-642-28869-2_10

[5] Pierre-Malo Deniélou and Nobuko Yoshida. 2013. Multiparty compatibility in communicating automata: characterisation and synthesis of global session types. In *Proceedings of the 40th International Conference on Automata, Languages, and Programming - Volume Part II* (Riga, Latvia) *(ICALP'13)*. Springer-Verlag, Berlin, Heidelberg, 174–186. https://doi.org/10.1007/978-3-642-39212-2_18

[6] Kohei Honda, Nobuko Yoshida, and Marco Carbone. 2016. Multiparty Asynchronous Session Types. *J. ACM* 63, 1, Article 9 (March 2016), 67 pages. https://doi.org/10.1145/2827695

[7] Hans Hüttel, Ivan Lanese, Vasco T. Vasconcelos, Luís Caires, Marco Carbone, Pierre-Malo Deniélou, Dimitris Mostrous, Luca Padovani, António Ravara, Emilio Tuosto, Hugo Torres Vieira, and Gianluigi Zavattaro. 2016. Foundations of Session Types and Behavioural Contracts. *ACM Comput. Surv.* 49, 1, Article 3 (2016), 36 pages. https://doi.org/10.1145/2873052

[8] Minseong Jang, Jungin Rhee, Woojin Lee, Shuangshuang Zhao, and Jeehoon Kang. 2024. Modular Hardware Design of Pipelined Circuits with Hazards. *Proceedings of the ACM on Programming Languages* 8, PLDI (June 2024), 28–51. https://doi.org/10.1145/3656378

[9] Elaine Li, Felix Stutz, Thomas Wies, and Damien Zufferey. 2023. Complete Multiparty Session Type Projection with Automata. In *Computer Aided Verification: 35th International Conference, CAV 2023, Paris, France, July 17–22, 2023, Proceedings, Part III* (Paris, France). Springer-Verlag, Berlin, Heidelberg, 350–373. https://doi.org/10.1007/978-3-031-37709-9_17

[10] Xun Li, Mohit Tiwari, Jason K. Oberg, Vineeth Kashyap, Frederic T. Chong, Timothy Sherwood, and Ben Hardekopf. 2011. Caisson: A Hardware Description Language for Secure Information Flow. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, San Jose California USA, 109–120. https://doi.org/10.1145/1993498.1993512

[11] Rachit Nigam, Pedro Henrique Azevedo de Amorim, and Adrian Sampson. 2023. Modular Hardware Design with Timeline Types. *Proc. ACM Program. Lang.* 7, PLDI, Article 120 (June 2023), 25 pages. https://doi.org/10.1145/3591234

[12] Jason Zhijingcheng Yu, Aditya Ranjan Jha, Umang Mathur, Trevor E Carlson, and Prateek Saxena. 2026. Anvil: A General-Purpose Timing-Safe Hardware Description Language. In *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '26)*. Association for Computing Machinery, Pittsburgh, PA, USA. https://doi.org/10.1145/3779212.3790125 To appear. Preprint available at arXiv:2503.19447.

[13] Danfeng Zhang, Yao Wang, G. Edward Suh, and Andrew C. Myers. 2015. A Hardware Design Language for Timing-Sensitive Information-Flow Security. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '15)*. Association for Computing Machinery, New York, NY, USA, 503–516. https://doi.org/10.1145/2694344.2694372