Automated Generation of Interval Properties From Trace-Based Function Models

Robert Kunzelmann*†, Aishwarya Sridhar*, Daniel Gerl*†,
Lakshmi Vidhath Boga*, Wolfgang Ecker*†
*Infineon Technologies AG
Am Campeon 1-15, 85579 Neubiberg, Germany
{robertniklas.kunzelmann, aishwarya.sridhar, daniel.gerl, lakshmividhath.boga, wolfgang.ecker}@infineon.com
†Technical University of Munich
Arcisstr. 21, 80333 Munich, Germany

***Abstract*—Function set modeling is a specification methodology striving for the unified description of general hardware systems. Based on the Instruction Set Architecture (ISA) of processors, function set modeling specifies systems exclusively by their executable functions and the relevant system state. While this methodology has been used in formal verification and behavioral modeling, the abstraction gap between system-level function models and design implementation limits its significance. We use traces, time-annotated representations of functions, to additionally model architecture parameters. Hierarchical State Machines (HSMs) are leveraged as a notation to capture the refined and conditional execution of the underlying function set. Moreover, we show the transformation of traces into formal assertion properties spanning over fixed-length intervals. The extracted interval set jointly verifies the complete behavior captured by the traces, thereby checking the functional and temporal correctness of the Design Under Verification (DUV).**

I. Introduction

As the complexity of integrated systems increases, formally specifying and modeling general digital hardware on a high abstraction level has recently gained interest. Approaches like Instruction-Level Abstraction (ILA) apply the specification methodology of Instruction Set Architectures (ISAs) of Central Processing Units (CPUs) to general hardware [1]. The resulting specifications model systems by their sets of executable functions and all required system states accessed by these functions. Both formal ISA specifications and ILA models have been used to generate formal properties for pre-silicon verification [2], [3]. However, such properties are limited by the wide abstraction gap between a system-level function description and a concrete Register-Transfer Level (RTL) implementation. Especially regarding timing, the abstraction gap presents an issue. Further information is required to state the exact time points at which specified expressions must hold.

Devarajegowda et al. present an architecture modeling and verification methodology based on traces [4]. In their terminology, a trace models the temporal and conditional execution of a single processor instruction. To this end, they use a Finite State Machine (FSM) notation (Section II-C) where each state represents a concrete time point of the multi-cycle instruction execution in a CPU pipeline. Transitions model the conditional propagation of an instruction from one function unit to the next based on control signals. Formal interval properties are generated from such traces using a metamodel-based code generator (Section II-A). While their methodology shows a noticeable reduction in manual work effort for verifying CPUs, we propose extensions to the trace notation and property generator, making traces applicable to verifying general hardware designs.

To extend traces for broader applicability and increased reusability, we introduce hierarchical structuring of traces using a Hierarchical State Machine (HSM) notation that allows for multiple traces to share common sub-traces (Section III). In combination with hierarchical traces, we present transformations to refine traces. Further, we present an extraction algorithm that automatically generates a complete set of interval properties from the HSM model of a function trace (Section IV). By applying our modeling and verification methodology to systems of various complexity, we show the wide applicability and effectiveness of the presented approach (Section V).

In summary, this work presents the following contributions:

- A notation to define and transform complex function traces by HSMs. This notation simplifies annotating system-level function specifications with the Design Under Verification (DUV)'s architectural structure.
- The integration of traces into a Model Driven Architecture (MDA)-based generator framework. The resulting workflow enables the automated generation of formal properties to verify general hardware designs.
- An algorithm generating formal properties from function traces. Each generated property spans over a fixed-length interval, thereby achieving a joint verification of non-time-deterministic traces.

II. Background

The following section presents the background knowledge this work is based on. This includes the property generator framework based on metamodeling, theoretical background on complete formal verification and completeness checks, and an existing definition of traces.

A. MDA-Based Property Generation

MDA and metamodeling describe concepts in code generation to stepwise refine models from high to low abstraction levels [5]. We apply this approach to generate formal properties in a 3-step workflow [6]. As shown in Figure 1, the property generation starts from a formal system specification. In the next step, a property generator transforms the parameters from the specification to an intermediate property model. Since the formal specification and property model are instances of their respective metamodels, the property generator is reusable for any instance of the same metamodel. In the third and last step, the intermediate property model is mapped to a specific Hardware Verification Language (HVL) using a templating engine. Since both the MetaProp metamodel and the HVL templates are agnostic of the specified system, the HVL generator is highly reusable.

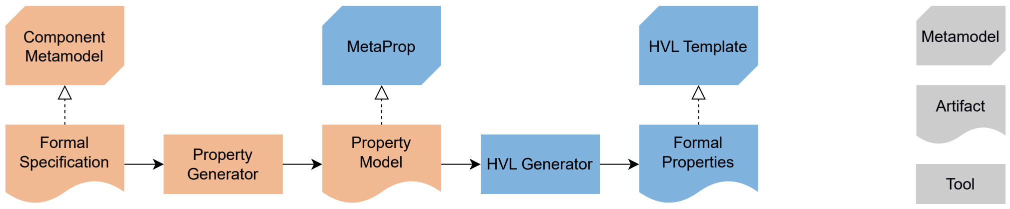

Figure 1: Generator framework of the MDA- and metamodeling-based property generation. Component-specific parts are colored orange, while component-agnostic elements are depicted in blue.

B. Complete Formal Verification

An essential question in formal verification is: *Have we written enough properties to completely verify the design's behavior?* While code coverage is often used as a metric to answer this question, it may not provide a comprehensive analysis of the completeness of the verification. To address this issue, researchers have developed additional metrics, known as completeness checks, to analyze whether the formal properties fully verify the DUV [7]–[9]. Completeness checks determine if a sequence of properties exists that verify the entire sequence of system states and all outputs for any input sequence. In other words, a complete formal verification is achieved when all behaviors of the DUV have been covered and verified.

To check whether a set of properties is complete, four tests have been proposed [7]:

- **Case Split Test**: Checks whether at least one property is active at every time point for any input sequence
- **Successor Test**: Checks for each property whether the succeeding property is uniquely determined
- **Determination Test**: Checks whether a set of properties determines the outputs and visible state variables of the DUV at every time point
- **Reset Test**: Checks whether the reset state is determined

C. Trace Notation

Traces are cycle-accurate models of operations of digital hardware [8]. Like waveforms—commonly used to visualize the behavior of digital circuits—traces specify values and expressions at each clock cycle. Devarajegowda defines traces as a state transition system where each state is aligned to a specific clock cycle, and transitions model the conditional execution of operations. In contrast to design FSMs—commonly used to model sequential logic—a trace focuses on modeling the temporal execution of a single function. Hence, there are key differences to FSMs. First, transitions in traces can take any number of clock cycles to execute. Second, states in traces represent time points instead of values stored in a state register. Thus, trace states do not necessarily have a state encoding.

Similar to an FSM, a trace is formally represented by a 7-tuple $\mathcal{T}_f := (S, S_i, I, O, \delta^l, \lambda^l, \theta)$ [8]:

- S: set of states
- $S_i \in S$: initial state
- I: set of input symbols
- O: set of output symbols
- δ^l: state transition function of l clock cycles with $\delta^l : S \times I \rightarrow S$
- λ^l: output function of l clock cycles with $\lambda^l : S \times I \rightarrow O$

- θ: interval function

The interval function $\theta := (s_s, s_e, \Delta, \Lambda)$ in turn is a 4-tuple of intervals of constant length within the trace:

- $s_s \in S$: start state of the interval
- $s_e \in S$: end state of the interval
- Δ: sequence of states throughout the interval
- Λ: sequence of output symbols throughout the interval

III. Trace-Based Modeling Methodology

While the trace notation from Section II-C allows us to model any system with clock cycle accuracy, we propose an extension to simplify its application for general hardware designs. For one, this includes the introduction of hierarchical states for easier reusability. Second, we model the data flow specification using state transitions and separate it from the control flow specification.

A. Formalism and Metamodel

Our extended trace definition is based on the original definition of traces shown in Section II-C and the formalism of *higraphs* from Harel [10]. We define a trace as a 10-tuple $\mathcal{T}_h := (M, S, S_i, T^l, I, O, \sigma, \tau, \lambda^l, \theta)$:

- M: finite set of sub-traces which are either hierarchical or flat transition systems
- S: finite, non-empty set of states
- S_i: finite, non-empty set of initial elements with $S_i \subseteq M \cup S$ and $|S_i| \geq |M|$
- T^l: set of transitions of l clock cycles with $T^l \subseteq (M \cup S) \times (M \cup S)$
- I: finite set of input symbols
- O: finite set of output symbols
- σ: sub-state function that maps a sub-trace to its set of comprising states with $\sigma : M \to 2^S$
- τ: transition map that assigns each transition to a set of ordered state pairs with $\tau : T^l \to 2^{S \times S}$
- λ^l: output function of l clock cycles with $\lambda^l : (M \cup S) \times I \to O$
- θ: interval function

Since the interval function θ is not directly applicable to hierarchical traces, its definition is equivalent to the formalism in Section II-C applied to the flattened trace $\mathcal{T}_f = \text{flat}(\mathcal{T}_h)$.

Figure 2 shows the implementations of such formalized hierarchical traces as a class diagram. Here, the classes *Transition* and *State* model flat traces. Each transition references a source and sink state and is executed if its trigger expression evaluates to *true*. In contrast to FSMs, transitions hold a *Length* attribute indicating that their execution requires l clock cycles. States reference any number of transitions as incoming or outgoing to this state. Further, a state is set as a trace's initial or final state, or neither, or both. The *Action* class models the output function. It comprises separate expressions for the Left-Hand Side (LHS) and Right-Hand Side (RHS), combined with an equivalence check or assignment, depending on the application. Finally, hierarchy is introduced by the *SubTrace* class, which references a trace. These sub-traces can comprise an arbitrary complex hierarchical structure. Since states and sub-traces share specific characteristics, their respective classes inherit from the same abstract parent class.

B. Instantiating Traces

In addition to formalizing trace models by the shown class diagram (see Figure 2), our in-house MDA-based generator framework supports the concept of instantiating sub-traces. This is indicated by the dashed reference from class *SubTrace* to *Trace* in Figure 2. The concept of instantiation allows us to reference sub-traces that are not part of the trace model itself. Such sub-traces can, therefore, be created and changed independently at a later point in time. This also allows us to easily exchange trace models of sub-components. For instance, we can choose from different bus interfaces—modeled by a set of independent traces—and only reference the desired one at the time of property generation. Therefore, this set of bus interface models is highly reusable and can also be provided to other component models.

C. Mapping Transitions

There are different—and conflicting—semantics of transitions between sub-HSMs [10]–[12]. Thus, we introduce the concept of a transition map τ for our hierarchical trace model. As previously defined in Section III-A, these maps allow us to explicitly bend a transition between sub-traces to a transition between their comprising states. To visualize the issue of ambiguous transition mapping, Figure 3 shows a simple example. When the HSM on the

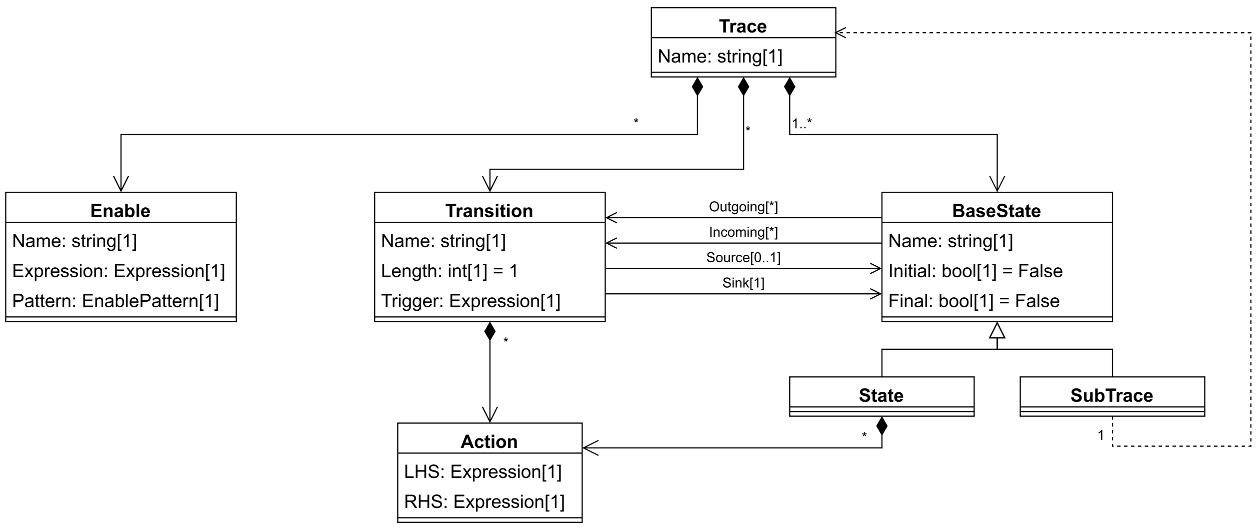

Figure 2: Trace metamodel depicted as a class diagram with a class for sub-traces allowing hierarchical structuring and a class to specify the enable condition of traces.

left-hand side is flattened to an FSM, multiple mappings of the colored transition are feasible. Our transition map lets us manually guide the flattening by either mapping t to $1 \rightarrow 3$, $2 \rightarrow 3$, or both. By default, if no explicit transition mapping is given, we follow the approach of Harel and map the transition to every state in the source sub-trace [10].

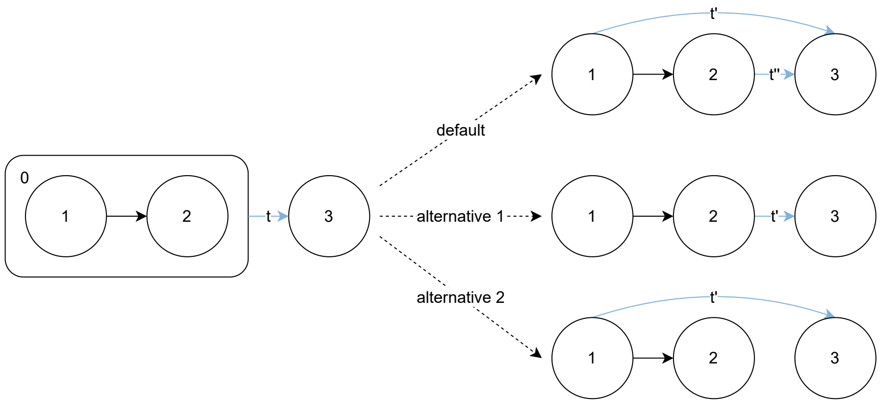

Figure 3: Example of all transition mappings from a hierarchical trace to its comprising states. By default, transition t is mapped to all internal states but may also be mapped according to the user-defined transition map.

Using these transition maps makes flattening hierarchical traces straightforward. Each transition referencing at least one sub-trace is transformed into a set of transitions according to the transition map. Finally, this transformation renders all higher-order sub-traces unconnected. Thus, they can safely be removed from the trace model, leaving a flat transition system only comprising states and transitions.

One issue during flattening arises from potentially ambiguous transition triggers. Here, we prioritize transitions based on the hierarchy level of their referenced sub-traces. Further, flattening requires identifying the *true* initial state since each sub-trace specifies its local initial element. The hierarchical trace is recursively traversed, starting from the top level and following the initial sub-traces. When a state is reached, it alone is stored as the initial state.

D. Separating Control and Data Flow

Besides the hierarchical modeling of traces, we present an extension to traces to separate control and data flow. We envisage only modeling the data flow of functions using state transitions of traces. Hence, modeling traces becomes more organized since only the primary execution strain of a function is explicitly represented. Additional control signals are set as attributes of traces and interpreted during property generation. These control flow attributes are captured using the *Enable* class shown in Figure 2. Objects of this class include the control flow expression and a description of the control flow pattern. The pattern is chosen from a predefined list of supported elements, which are interpreted differently by the generator.

An example trace is shown in Figure 4. For one, it includes the state transition system modeling the function execution. Second, two control elements are specified for this trace: an *Enable* and a *Start* condition.

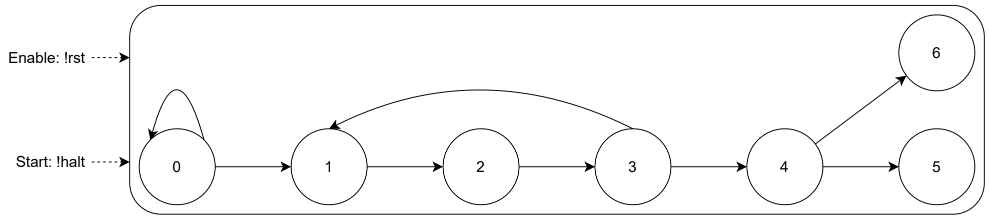

Figure 4: Example trace model of a function's data flow with separate control signals. The enable and start conditions are specified for the entire trace and interpreted by the property generator.

The keywords *Enable* and *Start* (see Figure 4) are elements of the predefined pattern list instructing the property generator to handle both expressions differently. The enable expression is global to the entire trace and potential sub-traces. Each property generated from this trace will be disabled during verification if the specified expression evaluates to *false*. In contrast, the starting condition instructs the property generator to insert an additional transition to the initial state of the trace that is only triggered if the specified expression evaluates to *true*.

The list of available enable patterns is finite since the property generator must uniquely interpret each pattern. Further, additional states and transitions in the trace must manually amend component and trace-specific control flow. Figure 4 shows the primary execution strain $0 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5$ and additional custom control flow elements by the transitions: $0 \rightarrow 0$, $3 \rightarrow 1$, and $4 \rightarrow 6$.

E. *Refining Traces*

Traces allow us to quickly transform and re-specify the execution duration of a function. Since each state in a trace is aligned to a fixed time grid, i.e., corresponds to a concrete clock cycle, adding and removing states changes the modeled execution time. Instead of modeling new traces for each variant of functionally equivalent components, we can quickly refine an existing trace that already models the function. Thus, trace refinement also contributes to higher reusability of existing traces.

Figure 5 shows a concrete example of trace refinement using a 32-bit adder. Performing the complete addition will take a specific number of cycles depending on the adder architecture in the DUV. Figure 5 presents how traces are stretched and compressed by adding and removing states.

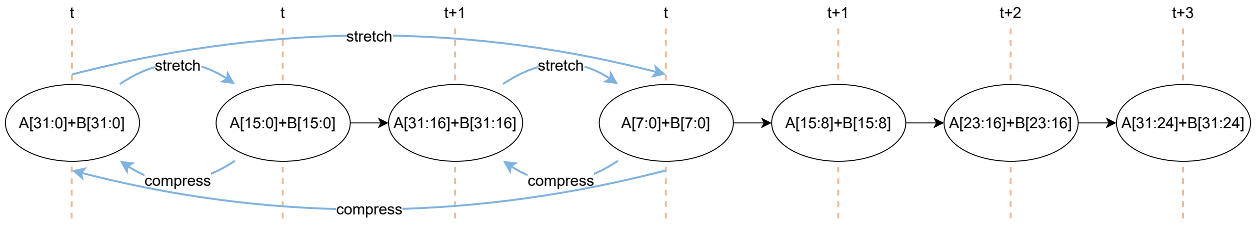

Figure 5: Example of the refinement of traces into different cycle-accurate models by the example of a 32-bit addition performed in either one, two, or four clock cycles.

IV. Formal Property Generation and Verification

In addition to the presented extensions of the trace notation, we show the systematic and fully automated generation of a set of intervals from traces in the following section. Expressing these intervals as formal properties jointly verifies the functional and temporal behavior captured by traces.

A. *Handling Non-Time-Deterministic Traces*

An issue of trace-based function modeling is the non-time-deterministic behavior of traces. Transitions—modeling the conditional execution of operations—can result in cyclic traces. Hence, the execution time of the trace depends on the number of loop iterations within the trace. This issue is addressed by the interval function θ of traces, which returns a set of fixed-length intervals. Generally, a single interval does not cover the entirety of a trace. It includes a subset of all states and transitions traversed under the assumption of a specific sequence of input values. Thus, completely covering the entire behavior captured by a trace requires extracting a set of intervals.

Based on the completeness checks presented in Section II-B, Bormann defines rules to extract a minimal set of intervals covering the behavior of a non-time-deterministic system and proves their completeness [7]. In summary, these rules require the following interval construction for cyclic traces:

1) One interval starts before the loop and ends at the first state inside the loop
2) One interval covers a single loop iteration of constant time
3) One interval starts from the last state inside the loop and ends after the loop
4) Additional branches from a loop or state result in additional intervals starting from the branching state

Figure 6 presents an example of the interval construction from a cyclic trace. The intervals $0 \rightarrow 0$ and $1 \rightarrow 2 \rightarrow 3 \rightarrow 1$ implement the second rule spanning over single iterations of the two loops. Interval $0 \rightarrow 1$ implements the third and first rule. It starts from the last state in the first loop and ends at the first state, which is part of the second loop. Interval $3 \rightarrow 4 \rightarrow 5$ also implements the third rule. The fourth rule for branches applies to the interval $4 \rightarrow 6$.

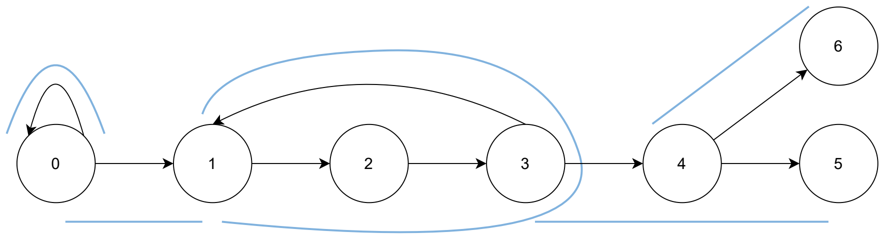

Figure 6: Example of a flat trace with its minimal set of complete intervals with each interval spanning over a constant duration and all intervals jointly covering all states and transitions.

B. Graph-Based Interval Extraction

Our generator toolchain uses a systematic and fully automated code generation approach, producing the complete interval set based on the interval extraction rules from Section IV-A. First, the hierarchical trace model is flattened according to the transition map τ described in Section III-C. To extract the interval set, this resulting trace is interpreted as a directed cyclic graph and traversed recursively using an adaption of the Depth-First Search (DFS) algorithm.

The rules for extracting a set of interval properties, as stated in Section IV-A, require identifying loops in the trace model. Hence, we first use Algorithm 1 to find all loops in the potentially cyclic trace graph. Starting from the initial state, we keep a list of all visited states (see line 11). There must be a loop if the current state is already in this list (see line 2). The found loop comprises all states starting from the index in the list where the current state was already inserted until the end of the list (see lines 4 - 6). According to the second rule, each found loop is translated into an interval property spanning over a single iteration (see Section IV-A).

After identifying all loops, we use Algorithm 2 to traverse the trace graph a second time. DFS is used again to recursively find all remaining intervals. As shown in Algorithm 2, we must differentiate between transitions that are not part of loops (see lines 5 - 16) and transitions that are (see lines 17 - 21). Since Algorithm 1 already generated a set of properties covering all loops, their comprising transitions must not be included in further intervals. Following the first rule (see Section IV-A), intervals are cut at states with no non-visited transitions that are not part of a loop, i.e., transitions that are not included in another interval (see lines 14 and 15). Nonetheless, transitions within a loop must not be ignored during traversal since there could be multiple branches from different states in the loop. Thus, all non-visited transitions within a loop are traversed if no other transitions are left (see lines 17 - 21). Since the taken transition is already covered by a loop, an empty interval is passed to the next recursion level (see line 20).

C. Formal Property Verification

After all loops and intervals are extracted from the trace model, they are directly mapped to an HVL. Our generator framework supports Interval Temporal Logic (ITL) and SystemVerilog Assertions (SVA) as target languages. Both languages allow structuring formal properties in a pre-condition and commitment. The pre-condition instructs the used formal tool to drive the DUV in a system state where said pre-condition evaluates to *true*. Starting from this state, the commitment is checked, resulting in a fail or a pass of the entire property. Generating properties from intervals, the pre-condition comprises the starting state and the conjunction of the trigger conditions of all

Algorithm 1: Adaption of the DFS algorithm to find all loops in the trace graph and translate them to properties.

```
function findLoops(state, parentStates) is
    input : state and list of all its parent states
    if state in parentStates then
        $loop \leftarrow \{state\}$
        foreach parent in parentStates after state do
            $loop.insert(parent)$
        end
        $loop.toHVL()$
    else
        $parentList.insert(state)$
        foreach non-visited outgoing transition of state do
            $transition.visited \leftarrow true$
            $nextState \leftarrow transition.sink$
            $findLoops(nextState, parentStates)$
        end
        $parentList.remove(state)$
    end
end
```

Algorithm 2: Adaption of the DFS algorithm to find all remaining intervals in the trace and translate them to properties.

```
function findIntervals(state, interval) is
    input : state and current interval
    if interval is not empty or state is not part of a loop then
        $interval.insert(state)$
    end
    if state has transitions that are not in a loop then
        foreach non-loop, non-visited transition of state do
            $interval.insert(state)$
            $transition.visited \leftarrow true$
            $nextState \leftarrow transition.sink$
            $findIntervals(nextState, interval)$
            $interval \leftarrow \{\}$
        end
    else
        $interval.toHVL()$
        $interval \leftarrow \{\}$
    end
    foreach non-visited transition of state do
        $transition.visited \leftarrow true$
        $nextState \leftarrow transition.sink$
        $findIntervals(nextState, \{\})$
    end
end
```

transitions along the interval. The property commitment is built from equivalence checks between the LHS and RHS expressions of all *Action* objects (see Figure 2) within the interval.

An example property spanning over a fixed-length interval is given below in Listing 1 and 2. This property verifies the execution of the 32-bit immediate *exclusive or* instruction of the RISC-V ISA [13]. Here, the pre-condition requires an *xori* instruction to be already fetched and issued, and the Arithmetic Logic Unit (ALU) to be available. In this specific example, the pre-condition requires an *exclusive or* to be executed in the next clock cycle. Verifying the behavior of the stages before the execute state and the stalling behavior if the ALU is not available is achieved by additional intervals extracted from the same function trace of the *xori* instruction.

```
property XORI_SVA;
@(posedge clk)
disable iff(rst)
  IssueReg[6:0] == 7'b0010011 &&
  IssueReg[14:12] == 3'b100 &&
  ALU_ready == 1'b1
|-> ##1
  RegisterFile[$past(IssueReg[11:7])] ==
  $past(RegisterFile[IssueReg[19:15]] ^
  $signed(IssueReg[24:20]));
endproperty
```

Listing 1: Example SVA property to formally verify the execution of the 32-bit RISC-V *xori* instruction.

```
property XORI_ITL;
assume:
  at t_IS: IssueReg[6:0] == 7'b0010011;
  at t_IS: IssueReg[14:12] == 3'b100;
  at t_IS: ALU_ready == 1'b1;
prove:
  at t_IS + 1: RegisterFile[
    IssueReg[11:7] @ t_IS)] ==
    (RegisterFile[IssueReg[19:15]] ^
    signed(IssueReg[24:20]) @ t_IS);
end property;
```

Listing 2: Example interval property to formally verify the execution of the 32-bit RISC-V *xori* instruction.

V. Application

One of the main arguments for using code generation is effort reduction by reusability. Even though developing a code generator can result in a higher development effort than a single produced artifact, it is worthwhile if the same generator is reusable for multiple systems. Therefore, to evaluate our approach, we analyze the relative work effort of applying our trace-based property generator compared to the produced properties for a set of DUVs:

- **AHB (S and M)**: set of properties to verify an Advanced High-performance Bus (AHB) slave and master interface
- **Register File**: verifies a 32×32-bit general purpose register file with dual-read ports
- **FIFO**: verifies an 8×32-bit First In First Out (FIFO) memory
- **DMA**: verifies a Direct Memory Access (DMA) peripheral
- **CPU**: verifies the instruction pipeline of a 2-stage RV32IMCZiCSR RISC-V processor

Table I shows the relative work effort of the trace modeling and formal property generation in Lines-of-Code (LoC). Further, it lists the sizes of the produced property files with SVA syntax.

TABLE I: LoC counts of various trace models, the common property generator, and produced properties to measure relative work effort.

Source Files	LoC					
	AHB (S)	AHB (M)	Register File	FIFO	DMA	CPU
Trace modeling (.py)	81	131	123	213	214	213
Formal property generator (.py)	- - - - - - - - - - - - - - - - - - - 472 - - - - - - - - - - - - - - - - - - -					
Formal properties (.sva)	144	229	500	341	994	1356

From Table I, we make two observations. Modeling the functional and temporal behavior of the shown DUVs using traces results in fewer lines of code than the generated formal properties. However, the generated properties might be less efficient regarding code size than hand-written properties. Still, the primary effort savings are apparent from the common formal property generator. This generator is component agnostic and only specific to the trace metamodel. Hence, this single component is directly reusable for all trace models combined.

VI. Conclusion

This work presents a trace-based architecture modeling and verification methodology using an HSM notation and metamodel-based code generator. Our approach applies to general hardware and enables easy reusability of sub-traces due to hierarchical modeling. We present methods for trace refinement and transformation, simplifying trace-based modeling. In combination with traces, we use a code generator to produce a set of interval properties based on established verification rules. The property set jointly verifies a trace's functional and temporal behavior. Even though the proposed property generator would result in a development overhead for the verification of a single design, we show that it is reusable for any trace model, making its development worthwhile after the application to only a few modules.

VII. Acknowledgement

This work was partly funded by the German Federal Ministry of Education and Research (BMBF) within the project Scale4Edge under contract number 16ME0122K.

References

[1] B.-Y. Huang, H. Zhang, P. Subramanyan, Y. Vizel, A. Gupta, and S. Malik, "Instruction-Level Abstraction (ILA): A Uniform Specification for System-on-Chip (SoC) Verification," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 24, no. 1, 2018, Place: New York, NY, USA Publisher: Association for Computing Machinery, ISSN: 1084-4309. DOI: 10.1145/3282444. [Online]. Available: https://doi.org/10.1145/3282444.

[2] A. Armstrong, T. Bauereiss, B. Campbell, *et al.*, "ISA Semantics for ARMv8-a, RISC-v, and CHERI-MIPS," *Proc. ACM Program. Lang.*, vol. 3, no. POPL, Jan. 2019, Place: New York, NY, USA Publisher: Association for Computing Machinery. DOI: 10.1145/3290384.

[3] Y. Xing, H. Lu, A. Gupta, and S. Malik, "Leveraging Processor Modeling and Verification for General Hardware Modules," in *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2021, pp. 1130–1135. DOI: 10.23919/DATE51398.2021.9474194.

[4] K. Devarajegowda, E. Kaja, S. Prebeck, and W. Ecker, "ISA Modeling with Trace Notation for Context Free Property Generation," in *2021 58th ACM/IEEE Design Automation Conference (DAC)*, 2021, pp. 619–624. DOI: 10.1109/DAC18074.2021.9586264.

[5] Object Management Group, *Model Driven Architecture (MDA)*, 2014. [Online]. Available: https://www.omg.org/cgi-bin/doc?ormsc/14-06-01.pdf (visited on 05/15/2023).

[6] K. Devarajegowda and W. Ecker, "Meta-model Based Automation of Properties for Pre-Silicon Verification," in *2018 IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC)*, ISSN: 2324-8440, Oct. 2018, pp. 231–236. DOI: 10.1109/VLSI-SoC.2018.8644957. [Online]. Available: https://ieeexplore.ieee.org/document/8644957 (visited on 11/10/2023).

[7] J. Bormann, *Complete Functional Verification*, Pages: 126 Type: other, 2017. [Online]. Available: http://nbn-resolving.de/urn:nbn:de:hbz:386-kluedo-46801.

[8] K. Devarajegowda, "Model-based Generation of Assertions for Pre-silicon Verification," Doctoral Thesis, Technische Universität Kaiserslautern, 2021. DOI: 10.26204/KLUEDO/6640. [Online]. Available: http://nbn-resolving.de/urn:nbn:de:hbz:386-kluedo-66403.

[9] J. Urdahl, D. Stoffel, J. Bormann, M. Wedler, and W. Kunz, "Path predicate abstraction by complete interval property checking," in *Proceedings of the 2010 Conference on Formal Methods in Computer-Aided Design*, ser. FMCAD '10, Austin, Texas: FMCAD Inc, Oct. 2010, pp. 207–215. (visited on 11/05/2023).

[10] D. Harel, "On Visual Formalisms," *Commun. ACM*, vol. 31, no. 5, pp. 514–530, May 1988, Place: New York, NY, USA Publisher: Association for Computing Machinery, ISSN: 0001-0782. DOI: 10.1145/42411.42414. [Online]. Available: https://doi.org/10.1145/42411.42414.

[11] M. Yannakakis, "Hierarchical State Machines," vol. 1872, Jan. 2000, pp. 315–330, ISBN: 978-3-540-67823-6. DOI: 10.1007/3-540-44929-9_24.

[12] A. Girault, B. Lee, and E. Lee, "Hierarchical finite state machines with multiple concurrency models," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 18, no. 6, pp. 742–760, Jun. 1999, Conference Name: IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, ISSN: 1937-4151. DOI: 10.1109/43.766725. [Online]. Available: https://ieeexplore.ieee.org/document/766725/authors#authors (visited on 11/06/2023).

[13] A. Waterman and K. Asanovic, "The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 20191213," RISC-V Foundation, Tech. Rep., Dec. 2019. [Online]. Available: https://riscv.org/technical/specifications/.