Massively Parallel Mining of Specifications for Hardware Designs

Leiqi Ye*, Guy Frankel*, Jianyi Cheng, and Elizabeth Polgreen

University of Edinburgh, UK
{leiqi.ye, G.Frankel-1, jianyi.cheng, elizabeth.polgreen}@ed.ac.uk

Abstract. Formal hardware verification ensures that a design satisfies its specifications, but writing these specifications requires substantial manual effort. Specification mining automates this process, and existing work has their own merits. The classic approaches rely on pre-defined templates, which have limited expressiveness and lack formal correctness guarantees. However, recent years have seen the emergence of using formal program synthesis for specification mining, which provides general and correct specifications but struggles to scale to complex designs.
In this work, we present MAPminer, a parallel framework for synthesis-based hardware specification mining. MAPminer exploits its novel algorithm based on the Maximal Universal Subset and partitions the synthesis problem into efficient sub-problems. These sub-problems are automatically scheduled across multiple threads for parallel synthesis. Experimental results show that MAPminer produces more effective assertions, improving verification coverage while reducing assertion size.

1 Introduction

Hardware verification ensures that a hardware design meets its functional specifications before fabrication. While writing these specifications is a laborious and time-consuming process, there has been growing interest in automatically generating them, also known as hardware specification mining [14]. Automatically mined specifications can be used to detect bugs, validate correctness, and guide formal verification, improving verification coverage and reducing human effort.

A recent theme in specification mining research exploits oracle-guided synthesis [33], which generates correct and meaningful specifications for arbitrary hardware designs compared to existing approaches such as machine learning [24,27,31,32] and template-guided [12,20] techniques. Still, it remains the case that synthesis-based techniques cannot be directly applied to verification tasks for realistic hardware designs due to two main challenges, *large problem size* and *limited parallelism*. First, synthesis scales exponentially with the number of variables, while real-world hardware designs often contain thousands of signals and state variables. Second, specification mining approaches typically compute sequentially and can take an impractically long time to achieve meaningful results.

* Equal contribution. Corresponding author: Leiqi Ye, leiqi.ye@ed.ac.uk.

Existing solutions for automated specification mining are limited and still face these challenges. In order to reduce the problem size, existing approaches typically rely on heuristics that fail to ensure that mined specifications reason about distinct and meaningful subsets of variables. GoldMine [14] exploits static analysis of the hardware source but only accepts Boolean circuits. HARM [12] and ARTmine [20] rely on template-based pattern matching and have limited variable reasoning for arbitrary hardware designs. SMART [33] relies on random choices of problem partitioning and may lead to sub-optimal specification results. Additionally, most related works follow sequential execution. While naive parallelism is possible, this could lead to substantial unnecessary duplication of computation for the same work.

In order to address these challenges, we seek an efficient solution that achieves the following design goals:

1) Efficient problem partitioning: Not all variables are equally relevant for specification mining, and they should be partitioned into smaller sets to reduce the problem size while preserving the quality of the mined specifications.

2) Redundancy-free parallelism: Only the mining work should be parallelized across threads, and shared components should be effectively shared by these threads to save computing costs.

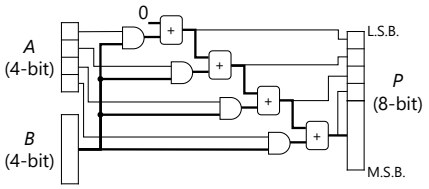
3) Efficient assertion size: The generated assertions should be concise and readable, helping further reasoning and debugging in practice and friendly to the model checker.

In this work, we present MAPminer, a MAssively Parallel specification miner that determines an efficient set of synthesis problems, schedules them into efficient parallel computation, and generates a set of efficient and meaningful assertions. MAPminer exploits a technique from the domain of satisfiability solving named Maximal Universal Subsets (mUS) to identify under-specified variables in the hardware design, and efficiently partitions the mining problem while minimizing duplicate work between threads. MAPminer generates SystemVerilog Assertions (SVA) for Verilog, a hardware description language. MAPminer supports both structural Verilog (low-level, reflecting underlying circuit topology) and behavioral Verilog (describing high-level functionality).

This paper makes the following contributions: 1) a fully automated synthesis framework for massively parallel hardware specification mining; 2) a partition technique that exploits mUS to partition the synthesis problem into efficient sets of independent sub-problems for acceleration; 3) a scheduling technique that maps partitioned problems into efficient parallel tasks; and 4) experimental results and case studies show that MAPminer can produce specifications with both significant speedup and improved assertion quality, contributing to the verification of realistic hardware designs.

2 Motivating Example

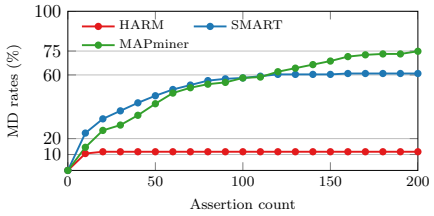
We consider the `s344` benchmark from the ISCAS'89 benchmark suite [2] as a motivating example and show how existing specification mining work produces



(a) A schematic of the hardware design to be verified ($P = A \times B$).

```
1  assert property
2   (ADDVG3VCNVOR1NF == S3);
3  assert property (CNTVCON1 == S3);
4  assert property
5   (ADDVG3VCNVOR1NF == CNTVCON1);
6  assert property
7   (ADDVG3VCNVOR1NF == CNTVCON1);
```

(b) Redundant assertions (lines 6 and 7) generated by SMART[33].



(c) A higher MD-rate means better verification coverage. HARM [12] relies on limited templates and stops improving after generating 20 assertions. SMART [33] relies on random partitioning and stops improving after generating 110 assertions. Our work continues improving while maintaining an efficient set of assertions.

Fig. 1: An example of mining specifications for a 4x4 Add-Shift Multiplier (`s344`).

inefficient assertions. This benchmark is a 4-bit multiplier using the add-shift algorithm, as shown in Figure 1a, but it has been widely used in common hardware designs. We choose this small design for simplicity, but it already shows the limitations of existing work, as further verified in Section 5.

When mining the specifications of the multiplier, the template-based approach Harm [12] enumerates its pre-defined templates and generates unverified and redundant assertions. The state-of-the-art method SMART [33] ensures the correctness of the generated assertions using formal synthesis. However, since synthesis techniques do not scale well with large sets of variables, SMART relies on random partitioning of the variable set to effectively capture specifications, which leads to redundancy in generated assertions.

Figure 1b shows four assertions generated by SMART for the multiplier. There are two types of redundancy in assertions, both logical and syntactic. The first two assertions capture the equivalence among three signals, `ADDVG3VCNVOR1NF`, `S3`, and `CNTVCON1`. The third assertion is syntactically different from the first two assertions but is logically redundant, because it describes a known relation between `ADDVG3VCNVOR1NF` and `CNTVCON1`. The fourth assertion is syntactically redundant, as it is the same as the third assertion.

We show how these mined assertions affect the verification coverage in Figure 1c. We use a metric named mutation detection (MD) rate to evaluate the quality of specifications, which, inspired by mutation testing, calculates the proportion of mutations applied to the design that can be detected by the specification. A higher mutation detection rate means better specifications. In the figure, the template-based Harm stops improving after enumerating all the templates. SMART achieves a higher mutation detection rate, but the random variable se-

lection limits the mutation detection rate to around 60%, after which it fails to identify any additional meaningful assertions.

In order to address this, our work MAPminer exploits mUS for efficient problem partitioning and generates assertions with reduced redundancy. This leads to a continuously increasing MD rate in the figure, with a significant improvement compared to other approaches. In the rest of the paper, we will describe our mUS-based partitioning algorithm and show how these partitioned problems can be effectively parallelized across multiple threads.

3 Problem Statement

Given a hardware design, our aim is to efficiently find a *meaningful* set of specifications, which accurately describe the behavior of the hardware (i.e., the specifications are never violated) and are sufficiently detailed that they can differentiate between the original hardware design and other designs, including those that are very similar but include small semantic bugs.
Formally, let:

- Σ be a set of observable design variables.
- $\mathcal{T} = \{\tau_1, \ldots, \tau_n\}$ be a set of executing traces, where each trace $\tau_1 : \mathbb{N} \rightarrow Val(\Sigma)$ maps time to signal/variable valuations.
- $\mathcal{L}$ be a specification language, i.e., System Verilog Assertions
- $\mathcal{C} \subseteq \mathcal{L}$ be a constrained hypothesis space. In this instance a grammar expressing a subset of the language $\mathcal{L}$.

Specification mining is the problem of automatically inferring a set of formal (possibly temporal) properties $\Phi = \{\varphi_1, \ldots \varphi_k\} \subseteq \mathcal{C}$ such that:

- Φ is *trace consistent* with $\mathcal{D}$, i.e., $\forall \varphi \in \Phi, \forall \tau \in \mathcal{T} : \tau \models \varphi$
- The design under test satisfies Φ, i.e., $\forall \Sigma . \mathcal{D} \models \varphi_1 \wedge \varphi_2 \wedge \ldots$, as checked by a formal verification tool.
- Φ captures a non-trivial and meaningful relationship among signals in Σ, and is thus able to differentiate between semantic mutations of $\mathcal{D}$.

The first requirement (trace consistency) is implied by the second (formally verifiable), but we present the requirements separately as many specification mining tools satisfy only the first requirement.

3.1 Specification Mining as Oracle-Guided Inductive Synthesis

Oracle-Guided Inductive Synthesis. Our specification mining approach is based around formal synthesis techniques. Formal synthesis aims to synthesize a function f such that it satisfies a formal constraint for all possible inputs. That is, it aims to solve the formula $\exists f . \forall x . \sigma$, where σ is a first-order formula which contains one or more invocations of f, and the free variables x.

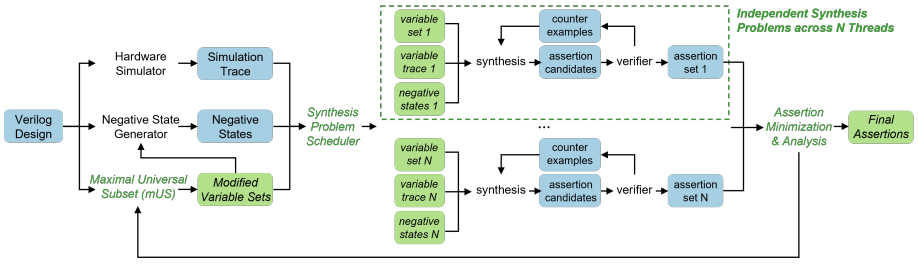


Fig. 2: An overview of the MAPminer flow. Our contributions are *highlighted*. The synthesis algorithm is taken from [33], the verifier is EBMC [1].

These problems are solved by Oracle-Guided Inductive Synthesis (OGIS) [23], whereby an algorithm alternates between a synthesis phase which guesses candidate solutions, and an oracle which validates whether the solutions are correct or not, and may provide further guidance. The synthesis phase is typically implemented as an enumerative search through the space of a context-free grammar.

Specification Mining as a Synthesis Problem. Prior work [33] demonstrates that the specification mining problem outlined above can be formulated as an oracle-guided synthesis problem. This approach requires a set of traces $\mathcal{T}$ and an additional set of negative states $\mathcal{N}$, defined as states that are unreachable in the hardware design. Instead of attempting to synthesize a specification by directly requiring the synthesizer to reason about $\mathcal{D}$, this approach generates a specification that approximates this requirement.

Namely, it generates specifications Φ that satisfy the following constraint: $\forall \tau \in \mathcal{T}.(\forall s \in \tau.s \models \Phi) \wedge \forall n \in \mathcal{N}.n \not\models \Phi$, where s is a single state in the trace τ. This constraint can then be further refined by adding more traces or more negative states. We note that negative states are used to guide the synthesis away from generating vacuously true specifications, but it is not necessary that the specifications evaluate to false in *every* unreachable state in the hardware design.

A key limitation of this approach is the scalability of the oracle guided inductive synthesizer when the number of variables in the hardware design becomes large. Given a specification problem, we aim to efficiently parallelize it into multiple tractable smaller synthesis problems.

4 Our Approach

An overview of the architecture of MAPminer is shown in Figure 2 and Algorithm 1. We build on existing synthesis-based specification mining techniques in the literature that use oracle-guided synthesis to generate specifications [33], captured by the *synthesis* block in the diagram. Specifically, we provide a means to run this method in parallel while maintaining high specification specificity and limiting the amount of redundant work carried out across threads.

Algorithm 1 MAPminer: Massively Parallel Specification Mining

procedure MAPMINER(Hardware design D, testbench TB, number of threads k, timeout T)
  $\Phi \leftarrow \emptyset$
  Collect simulation traces from $\mathcal{D}$ using TB
  Extract design variables V
  Randomly partition V into k variable sets: $\mathcal{V} = \{V_1, V_2, \ldots, V_k\}$
  while time $< T$ **do**
    for all $V_i \in \mathcal{V}$ **do**
      Generate negative states N_i for variables in V_i
      Generate grammar G_i for variables in V_i
    end for
    in parallel for $i = 1$ to k:
      $\Phi_i \leftarrow$ SYNTHESIZE$(\mathcal{D}, V_i, N_i, G_i)$
    Collect assertions: $\Phi \leftarrow \Phi \cup \bigcup_{i=1}^{k} \Phi_i$
    if no new assertions were added to Φ **then**
      break
    end if
    $M \leftarrow$ COMPUTEMUS(Φ)
    if |M| is stable **then**
      break
    end if
    Derive new variable sets $\mathcal{V}$ from M
  end while
  Minimize Φ using equivalence checking
  return Φ
end procedure

The key components of our approach are as follows:

Synthesis Problem Decomposition via Variable Selection. A key problem in enumerative search over grammars is the size of the search space. This is exacerbated when dealing with traces that contain a large number of variables. Prior work solves this by sequentially solving many simple problems through selecting random sets of variables to be fed to the synthesizer. In MAPminer, we run multiple synthesizers in parallel, each solving a smaller problem over a subset of the variables.

This is enabled by our novel approach to iteratively identify *meaningful* sets of variables for synthesis, based on identifying variables in the hardware design that are not yet constrained by the specification. We refer to these variables as "under-specified" variables, and we identify them by calculating the mUS of the specification obtained so far.

Once the under-specified variables have been identified, we partition this into multiple variable sets, and construct synthesis queries to be run in parallel to mine assertions over each variable set. Along with providing guidance to the search, these meaningful variable sets also ensure minimal duplicate work between threads.

Parallel Synthesis Problem Construction. As described in section 3.1, the synthesis tool requires three inputs: a set of traces $\mathcal{T}$ and a set of negative states $\mathcal{N}$ and a grammar G from which to construct assertions. Given a variable set generated by the previous step, we generate $\mathcal{N}$ by randomly generating states and selecting for those that are deemed unreachable using a hardware model checker. Importantly, each negative state is defined only over the variables in the current synthesis query, not over the full RTL state space, so the corresponding reachability check is small; in practice, EBMC verifies each negative state within a second. Furthermore, negative states are shared across parallel synthesis tasks, so this step is not a runtime bottleneck. We assume we are provided with a hardware testbench for $\mathcal{D}$, and we use this simulation environment with a standard Verilog simulator to generate simulation traces from the design under verification.
The grammar we provide, given a variable set V, is shown below and allows for a broad class of predicates over bitvectors and booleans.

$$
\begin{aligned}
\mathit{Expr} ::= & \mathit{Atom} \mid \mathit{Atom} \implies \mathit{Atom} \\
\mathit{Atom} ::= & \mathit{Bool} \simeq \mathit{Bool} \mid \neg \mathit{Bool} \mid \mathit{Bool} \wedge \mathit{Bool} \mid \mathit{Bool} \vee \mathit{Bool} \mid Bv \simeq Bv \mid \\
& Bv \geq Bv \mid Bv \leq Bv \mid Bv < Bv \mid Bv > Bv \mid Bv \geq_u Bv \mid \\
& Bv \leq_u Bv \mid Bv <_u Bv \mid Bv >_u Bv \mid \text{any boolean } v \text{ in } V \\
Bv ::= & -Bv \mid \sim Bv \mid Bv \ \& \ Bv \mid Bv \mid Bv \mid Bv + Bv \mid Bv - Bv \mid \\
& Bv * Bv \mid \text{any bitvector } v \text{ in } V
\end{aligned}
$$

Specification Verification and Refinement. When a synthesis job generates an assertion, it is verified using a hardware model checking tool. If the hardware model checking tool says that the assertion does not hold on the design under test, the counterexample is fed back to the synthesizer as an additional trace, refining the synthesis conjecture. This process repeats until either we reach a timeout, or we have found an assertion that passes verification.
We use the tool EBMC in k-induction mode to verify each assertion. k-induction is a standard technique for proving safety properties of transition systems, and generalizes mathematical induction to hardware models by checking two conditions: a base case and an inductive step. In the base case, the model checker verifies that the specification holds for all states reachable within the first k steps from the initial states. In the inductive step, it assumes that the specification holds for k consecutive states and proves that it also holds for the $(k+1)^{th}$ state under the system's transition relation. If both conditions succeed, the specification is guaranteed to hold for all reachable states.

Specification Minimization. Once all parallel synthesis jobs are terminated, we have a set of verified assertions. Given this set, a minimization procedure is implemented, making use of an iterative search through a lattice of assertions to reduce the amount of redundancy in the specifications. The minimized assertions are returned to the variable-selection step, the set of under-specified variables is updated, and the process repeats. The minimizer not only improves

the readability of the specifications for the user, but also reduces the size of the problem that the variable selection must reason about.

MAPminer continues in this loop until either: 1) a timeout is reached; 2) no new assertions are generated; or 3) the number of under-specified variables is stable between the last blocks. We now describe the variable selection and minimization in detail.

4.1 Variable Selection via Maximal Universal Subsets

During assertion generation, MAPminer uses subsets of the variable set in order to ensure computation feasibility and parallelism of the generation process. This requires a 'smart' method of selecting which variables are provided to each block. We hypothesize that a variable that has not yet been constrained by the specification is likely to be a variable of interest that should be considered in future specification mining attempts.

We use the concept of Minimal Satisfying Assignments (mSA) [8,19] and their dual, mUS, from satisfiability solving to identify variables of interest. Intuitively, an mSA is the minimal cost partial assignment of values to variables in the formula that guarantees the formula is true, and the corresponding mUS includes every variable not included in the mSA.

Given a formula f and a domain D, an assignment σ is a function mapping the free variables in f (denoted $free(f)$) to elements of D. An assignment is full if it maps all variables in $free(f)$ to the domain and partial otherwise. The formula is satisfiable if there exists an assignment under which the formula evaluates to true (a *satisfying assignment*). We use $vars(\sigma)$ to indicate the set of variables that σ has assigned to. For example, the formula $x + y = 7$ is satisfiable and the assignment $\sigma = [x \mapsto 1, y \mapsto 6]$ is a satisfying assignment, and $vars(\sigma) = \{x, y\}$. We use the notation $f[\sigma]$ to indicate the result of substituting $vars(\sigma)$ in f for their corresponding assignments, so $f[\sigma]$ would be $1 + 6 = 7$.

Definition 1. *Given a formula f, a partial satisfying assignment σ is a satisfying assignment, such that $vars(\sigma) \subseteq free(f)$ and for every assignment ρ to the variables in $free(f) \setminus vars(\sigma)$, $f[\sigma \cup \rho] = true$. The satisfying assignment is a minimal satisfying assignment (mSA) if it minimizes $|vars(\sigma)|$.*

Given a design under verification $\mathcal{D}$ and a specification Φ, we know that Φ is satisfiable as, by construction, the specification is true for the design under verification. We define the mSA to be the minimal set of design variables in Σ which must be assigned to in order to determine that a single state is a state which satisfies the specification Φ.

Definition 2. *Given a specification Φ, a minimal satisfying assignment (mSA) is an assignment σ such that $vars(\sigma) \subseteq free(\Phi)$ and for every assignment ρ to the variables in $free(\Phi) \setminus vars(\sigma)$, $\Phi[\sigma \cup \rho] = true$, and which minimizes $|vars(\sigma)|$.*

Intuitively, the variables in the mSA are the variables that *are* currently constrained by the specification. In order to identify the variables that we should send to the next iteration of MAPminer, we are interested in finding the dual of the mSA: the mUS, or the *under-specified variables*.

Definition 3. *Given a specification Φ, a universal set is the set of variables $X \subseteq free(\Phi)$ such that $\forall X.\Phi$ is satisfiable. A Maximal Universal Subset is the universal set $X \subseteq free(\Phi)$ maximizing $|X|$. Given an mSA σ for a specification Φ, as defined in definition 2, the mUS can be calculated as $mUS = free(\Phi) \setminus vars(\sigma)$.*

There are many possible mSAs and mUSs, and so we carry out a greedy search to find an mUS. We first note that our specification Φ is always of the form $\bigwedge_i \varphi_i$, i.e., a conjunction of predicates. As a consequence of this, we know that the mSA must include at least one variable from every $\varphi_i \in \Phi$. We thus begin by searching for the minimal set of variables for which this is true. This is known as a Minimal Hitting Set (mHS).

Definition 4. *Given a specification Φ, a hitting set is a set of variables $\mathcal{H} \subseteq free(\Phi)$ such that for all $\varphi \in \Phi$, $free(\varphi) \cap \mathcal{H} \neq \emptyset$. A mHS is a hitting set such that for all $h \in \mathcal{H}$, $\mathcal{H} \setminus \{h\}$ is not a hitting set.*

We use minimal hitting sets to search for an mUS using the following steps:

1. **Find an mHS.** To generate an initial mHS, we first order the variables in $free(\Phi)$ by the number of predicates in ϕ in which they appear. Formally, let

$$\mathcal{O}(free(\Phi)) = \{x_{(1)}, x_{(2)}, \ldots, x_{(n)}\}$$

be an ordering of the variables $free(\Phi)$ such that $\mathrm{occ}_\Phi(x_{(1)}) \geq \mathrm{occ}_\Phi(x_{(2)}) \geq \cdots \geq \mathrm{occ}_\Phi(x_{(n)})$, where $\mathrm{occ}_\Phi(x) = \left|\{\, i \mid x \in \mathrm{free}(\varphi_i) \,\}\right|$.
We traverse the variables in the order $x_{(1)}, \ldots, x_{(n)}$, adding $x_{(k)}$ to the selected set iff it appears in some predicate whose free variables are not yet covered by the previously selected variables.
Formally, define the coverage of a variable set S as $\mathrm{cov}_\Phi(S) := \{\, i \mid \mathrm{free}(\varphi_i) \cap S \neq \varnothing \,\}$. The minimal Hitting Set of Φ is then defined as follows:

$$\mathcal{H}(\Phi) := \left\{\, x_{(k)} \;\middle|\; \exists i \notin \mathrm{cov}_\Phi(\{x_{(1)}, \ldots, x_{(k-1)}\}) \text{ such that } x_{(k)} \in \mathrm{free}(\varphi_i) \,\right\}.$$

2. **Guess an mUS.** We know that any mSA must also be an mHS. So, suppose that a minimal satisfying assignment σ exists, and $vars(\sigma) = \mathcal{H}(\Phi)$. The corresponding mUS is given by $X = free(\Phi) \setminus \mathcal{H}(\Phi)$.
3. **Check the mUS.** We then check if X is a valid mUS by checking whether the formula $\forall X.\Phi$ is satisfiable. If it is satisfiable, X is a valid mUS, and we know that a corresponding mSA σ exists and $vars(\sigma) = free(\Phi) \setminus X$, and we are done. If it is unsatisfiable, we aim to find a minimal set of predicates $\varphi \in \Phi$

that are sufficient to determine that this formula is unsatisfiable. To do this, we use the fact that universal quantification distributes over conjunction.

$$\forall X. \Phi \iff \forall X. \bigwedge_{\varphi \in \Phi} \varphi \iff \bigwedge_{\varphi \in \Phi} \forall X. \varphi$$

We query an oracle with the distributed conjunction, and extract the unsat core, $\mathcal{C}$, i.e., the subset of clauses whose conjunction is still unsatisfiable.

4. Correct the mUS. We use the unsat core $\mathcal{C}$ to determine a possible fix for the invalid mUS. Given $\mathcal{C}$ is a set of clauses $c_1, \ldots, c_n$, we use the same process as before to find an mHS of $\mathcal{C}$, except that we exclude any variables already present in $\mathcal{H}$. That is, we find:

$$\{\, x_{(k)} \mid \exists i \notin \mathrm{cov}_{\mathcal{C}}(\{x_{(1)}, \ldots, x_{(k-1)}\}) \text{ such that } x_{(k)} \in \mathrm{free}(c_i) \setminus \mathcal{H}(\Phi) \,\}.$$

where $\mathrm{cov}_{\mathcal{C}}(S) := \{\, i \mid \mathrm{free}(c_i) \cap S \neq \varnothing \,\}$. By abuse of notation, we refer to this as $\mathcal{H}(\mathcal{C} \setminus \mathcal{H}(\Phi))$.
We add the variables in $\mathcal{H}(\mathcal{C} \setminus \mathcal{H}(\Phi))$ to $\mathcal{H}(\Phi)$ and return to step 2.

This is an over-approximation of the required addition to X, however, in practice often $|\mathcal{C}| = 1$ and thus we only add a single variable at a time.

Using the mUS within MAPminer. We use the mUS to select the variables that should be considered for specification mining at the beginning of each iteration. Note that in the first iteration, the mUS is equal to Σ, the full set of variables in the hardware design. We then randomly select subsets of variables from the mUS X for the next iteration of MAPminer. The size of the variable sets is determined by the number of variables available and is calculated as $\mathrm{round}(2.7 + 1.6 \log_{10} |X|)$. We avoid subsets which contain the same variable twice, and subsets that we have chosen before.

Additional positive and negative states. On finding an mUS, X, the SMT solver provides the mSA in the form of a satisfying assignment σ for the query $\forall X.\Phi$. We use this assignment to generate additional negative states: given a target variable set V, we take the assignments from σ for the variables in V. We then randomly assign values to any remaining variables in V and check whether the state corresponding to this assignment is reachable in the original hardware model using k-induction. If the state is unreachable, we add this assignment to the list of negative states used to guide the synthesis based assertion mining.

4.2 Specification Minimization

Variables are present in multiple variable sets, which is necessary in order to consider relationships between multiple different variables, but this means that we may create redundant assertions. Take $\Sigma = \{x_1, x_2, x_3, x_4\}$ to be our set of design variables. Say we split Σ into $V_1 = \{x_2, x_3\}$ and $V_2 = \{x_2, x_4\}$ and generate the following assertions $\Phi = \varphi_1 \wedge \varphi_2$, where $\varphi_1 = x_2 \vee x_3$ and $\varphi_2 = x_4$.

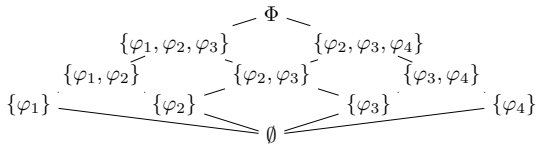


Fig. 3: Subset lattice of an assertion set illustrating greedy redundancy reduction. Starting from the full set, the search descends to smaller subsets that are implied by the original assertion set (green nodes), stopping when no further valid descendant exists.

Algorithm 2 Maximal Universal Subset

```
1: procedure MAXIMAL UNIVERSAL SUBSET(Assertion set Φ)
2:     H ← MINIMALHITTINGSET(Φ)
3:     while 1 do
4:         X ← free(Φ) \ H
5:         q ← ∀X.Φ
6:         if ISSAT(q) then
7:             return X
8:         end if
9:         C ← UNSATCORE(∀X.Φ)
10:        H ← H ∪ MINIMALHITTINGSET(C \ H)
11:    end while
12: end procedure
```

We then split Σ into $V_3 = \{x_1, x_3\}$ and $V_4 = \{x_2, x_4\}$ and generate $\varphi_3 = x_1 \vee x_3$ and $\varphi_4 = x_2$. Assertion φ_1 is redundant in $\Phi' = \varphi_1 \wedge \varphi_2 \wedge \varphi_3 \wedge \varphi_4$ since we have φ_4.

To ensure our final solution contains little redundancy and to reduce the size of the assertion set given to the mSA calculation we carry out assertion minimization at the end of each iteration of MAPminer and before returning the solutions to the user.

Given a specification Φ containing redundancy, the minimal set of assertions Π is a subset of Φ such that $\Pi \equiv \Phi$ and $\forall \varphi \in \Pi, (\Pi \setminus \varphi) \not\equiv \Phi$. Computing a perfectly minimal set is too computationally expensive to integrate into MAPminer, and so we implement a greedy search to find the best minimization we can in a practical amount of time. We take advantage of the partial order $(\mathcal{P}(\Phi), \subseteq)$ over subsets of the assertion set Φ, ordered by cardinality. Starting with the conjunction over the whole assertion set, Φ, we greedily search for a subset, $\Pi \subseteq \Phi$, such that Π is logically equivalent to Φ.

The search proceeds by traversing the subset lattice level by level, removing predicates from Π. Whenever a candidate subset Π is found to be valid, we descend in the lattice and test its descendants. Either we find another $\Pi' \subset \Pi$ such that $\Pi' \implies \Phi$ from which we continue descending, or we find no such Π' and return Π. This search is illustrated in Figure 3, where we see the search

Algorithm 3 Assertion Set Minimization

```
1: procedure MINIMIZATION(Assertion set Φ)
2:     Π ← Φ
3:     Construct lattice (P(Φ), ⊆) ordered by cardinality
4:     for k = |Φ| − 1 down to 1 do
5:         for all Π′ ⊂ Π such that |Π′| = k do
6:             if Π′ ≡ Φ then
7:                 Π ← Π′
8:                 break
9:             end if
10:        end for
11:    end for
12:    return Π
13: end procedure
```

space and the target Π, in this case $\{\varphi_2, \varphi_3\}$. The pseudocode for minimization is shown in Algorithm 3.

5 Evaluation

Evaluating the Utility of Specifications. Evaluating the utility of a specification is challenging. We use the standard metric from the literature, namely Mutation Detection. Mutation Detection, inspired by Mutation Testing [18], assesses how good a specification or test is by mutating the design under verification/test. A good specification should be able to differentiate between the original design and the mutated design. Thus, given a design under test $\mathcal{D}$ and a set of n mutated designs, $\mathcal{M} = \{D_1^*, D_2^*, \ldots, D_n^*\}$, we report the Mutation Detection Rate (MD-Rate) as the proportion of mutants for which the specification does not hold, calculated as MD-rate $= \frac{1}{|\mathcal{M}|}|\{D^* \in \mathcal{M} \mid D^* \not\models \phi\}|$.

Generating the Mutants. We use a set of mutation rules from the literature [18,33] to generate mutants. We identify the rules that could be applied to the design under verification. For each mutant, we randomly select one rule and apply it to one location in the file. These rules can generate mutations that do not meaningfully change the semantics of the design under verification, and so it should be noted that, while a higher mutation detection rate is clearly better, it might be impossible to achieve a mutation detection rate of 100%.

Formal Verification of Specifications. All specifications generated by MAPminer are formally verified against the hardware design and are therefore guaranteed to be correct to the design. However, not all specification mining tools provide this guarantee. In particular, we compare to HARM [12], which does not guarantee that all assertions hold on the design. We verify assertions generated by HARM post-hoc, using EBMC, and report the proportion of assertions generated that pass verification as the Verification Correctness (VC)-rate. In addition,

HARM produces unverified assertions, so we evaluate the MD-rates only on the assertions that pass the verification, taking the best case of their work.

5.1 Benchmarks and Experimental Setup

We implement MAPminer, using cvc5-1.2.0 from [5] and use the same core miner as the related work SMART [33]. We use Z3-4.15.4.0 from [25] as the underlying solver for calculating the mUS and simplifying the assertions. We use Cocotb-1.8.0 [29] and ebmc-5.6 [1] as the hardware simulator and formal assertion verifier, respectively.

All experiments are conducted on a dual-socket server equipped with two Intel Xeon Gold 6438Y+ processors, providing 128 logical CPUs (2 sockets $\times$ 32 cores $\times$ 2 threads). The machine has 125 GiB of main memory and runs a 64-bit Linux operating system on x86_64 architecture. In each experiment, we limit execution to 32 threads to ensure consistent and fair comparison across different configurations. Each tool is run with a 12-hour (43,200s) timeout; if a tool does not terminate within this budget, its runtime is recorded as 43,200s.

We compare MAPminer to SMART [33] and HARM [12]. HARM is a template-based miner that matches patterns on simulation traces. All three tools are evaluated using the same simulation testbench for each benchmark. We run HARM using 10,000-cycle random input traces with a larger random step depth than MAPminer and SMART, which in practice gives HARM a more favorable setup in the comparison. HARM is parallelized across 32 threads. Within MAPminer, we conduct an ablation study under different configurations to analyze the contribution of each component, in particular the use of mUS and the minimizer. We consider four variants: mUS+Minimizer (default: uses mUS and the minimizer after every iteration of the loop), mUS only (uses mUS to select variables but no minimization), Random+mUS (mixes mUS selected variables with randomly selected variables), and Full Random (does not use mUS at all).

To evaluate MAPminer, we employ benchmarks from prior work alongside two widely used hardware benchmark suites. From GoldMine [14], we select behavioral Verilog designs derived from the Ibex RISC-V CPU [28]—including `arb2`, `controller`, `decoder`, and `multdiv_slow`—which model representative microarchitectural components featuring non-trivial control logic and stateful behavior. Additionally, we incorporate the ISCAS'85 and ISCAS'89 benchmark suites [2], comprising purely combinational and sequential circuits, respectively. ISCAS'85 includes circuits ranging from simple logic (e.g., `c17` with 6 gates) to complex arithmetic units (e.g., `c6288`, a 16$\times$16 multiplier with over 2,400 gates). ISCAS'89 provides sequential benchmarks of varying complexity, from small finite state machines such as `s27` (3 flip-flops) to large-scale industrial designs such as `s38584` and `s35932`, which contain over 20,000 gates and more than 1,700 flip-flops. Together, these benchmarks provide comprehensive coverage of both behavioral and gate-level hardware designs across a wide range of circuit scales.

Table 1: Summary of results comparing MAPminer and related work. We report the average size of the generated assertion set $|\Phi|$, VC-rate, MD-rate, and average runtime over the 38 benchmarks common to all three tools. Each tool is run with a 12-hour timeout.

Category (#)	Metric	MAPminer	SMART	HARM
Structural (34)	avg. $\|\Phi\|$	3,957	1,319	60,887
	avg. VC (%)	100.0	100.0	15.28
	avg. MD (%)	76.94	59.57	17.77
	avg. Time (s)	12,089	2,076	303
Behavioral (4)	avg. $\|\Phi\|$	21	13	3,114
	avg. VC (%)	100.0	100.0	18.13
	avg. MD (%)	21.96	19.31	29.70
	avg. Time (s)	126	51	1,994
All (38)	avg. $\|\Phi\|$	3,543	1,181	54,806
	avg. VC (%)	100.0	100.0	15.58
	avg. MD (%)	71.16	55.33	19.03
	avg. Time (s)	10,830	1,863	481

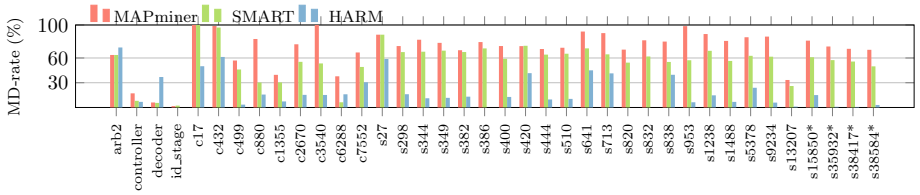


Fig. 4: Comparison of MD-rates between MAPminer, SMART [33], and HARM [12] on the benchmarks.

5.2 Results

VC-rate. All assertions generated by MAPminer and SMART are further validated using formal verification, and so both achieve a 100% VC-rate. HARM frequently fails to produce verified assertions.

MD-rate. The mutation detection results are summarized in Table 1. Compared with the state-of-the-art tool SMART, MAPminer achieves higher MD-rates on most benchmarks. Overall, MAPminer improves the average MD-rate from 55.33% to 71.16%, corresponding to an absolute gain of 15.83 percentage points.

Assertion Size. Table 1 reports the average number of generated assertions $|\Phi|$ across the common benchmark set. Overall, MAPminer generates a moderate number of assertions compared with existing approaches. On average, MAPminer produces 3,543 assertions, which is larger than SMART (1,181) but more than an order of magnitude smaller than HARM (54,806). Although it generates more assertions than SMART, MAPminer achieves substantially higher

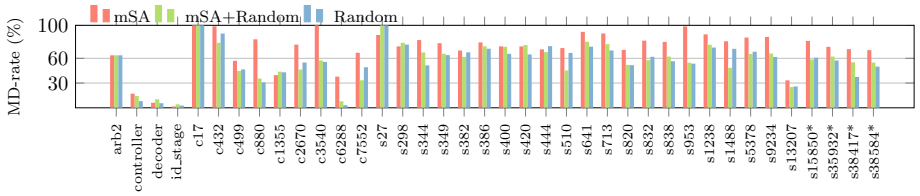


Fig. 5: Comparison of MD-rates between mUS, mUS+Random, and Random.

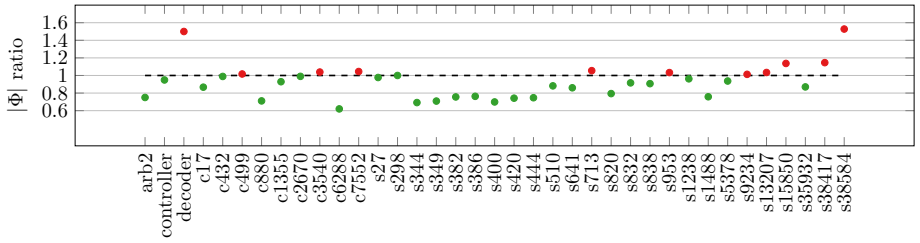


Fig. 6: Ratio of assertion set size between mUS+Minimizer and mUS ($\frac{|\Phi_{mUS+Minimizer}|}{|\Phi_{mUS}|}$). Green points are benchmarks where the $|\Phi|$ is reduced by the minimizer. *We omit the outlier benchmark `ibex_id_stage` (ratio = 8.25)

MD-rates, indicating that the additional assertions contribute to improved verification coverage rather than redundancy.

Impact of Variable Selection. Figure 5 compares MAPminer with different variable selection strategies. Overall, mUS-based selection improves mutation detection rates across the benchmark set. The benefit becomes increasingly pronounced on larger benchmarks, where random or partially random selection fails to scale, leading to substantially higher MD-rates. On smaller benchmarks, fully random selection can occasionally achieve comparable results due to the limited variable space.

Impact of Minimization. We evaluate the impact of assertion minimization. Figure 6 reports the ratio between the final number of assertions generated with minimization enabled (mUS+Min) and without minimization (mUS).

For most benchmarks, this ratio is below one, indicating that enabling minimization leads to a smaller final assertion set. This shows that a substantial fraction of mined assertions are redundant and can be effectively eliminated by the minimizer. For some larger benchmarks and timeout-prone cases, the ratio exceeds one. In these cases, minimization removes redundancy earlier during the mining process, which allows the miner to make further progress within the same time budget. As a result, more non-redundant assertions can be discovered overall, leading to a larger final assertion set compared to disabling minimization.

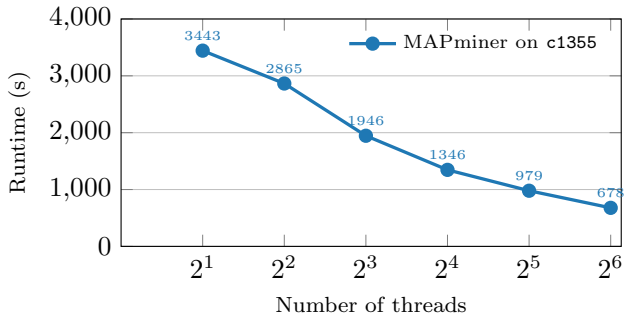


Fig. 7: Impact of thread count on runtime for benchmark `c1355`. Runtime decreases steadily as the number of threads increases.

Overall, this ratio-based comparison shows that minimization is effective in most cases, either by reducing the final assertion size or by enabling additional useful assertions to be mined within the same resource constraints.

Impact of Parallelism. We evaluate the effect of thread count on benchmark `c1355` (588 variables). Figure 7 shows that runtime decreases from 3,443s with 2 threads to 678s with 64 threads, demonstrating effective utilization of parallel resources. Increasing the number of threads primarily reduces wall-clock runtime; we did not observe a meaningful change in MD-rate or assertion size across these configurations. This confirms that the parallelism in MAPminer provides a direct speedup without affecting the quality of the mined specifications.

Runtime Analysis. Table 1 reports the average runtime across the common benchmark set. HARM is the fastest tool on average (481s), as its template-based pattern matching avoids expensive synthesis queries, but it produces the lowest-quality assertions. SMART averages 1,863s and MAPminer averages 10,830s. Both SMART and MAPminer reach the 12-hour timeout on some of the largest benchmarks. Runtime differences are strongly benchmark-dependent: on `c6288`, MAPminer finishes in 2,288s while SMART takes 23,670s, whereas on `s641`, SMART finishes in 66s while MAPminer takes 2,097s. MAPminer can take longer because it continues to discover new meaningful assertions via mUS-guided variable selection, whereas SMART and HARM may terminate earlier once their search spaces are exhausted. The ablation study (Figure 5) confirms that the MD-rate improvement of MAPminer over SMART is attributable to mUS-based variable selection rather than longer runtime alone, since the Full Random variant, which is run under the same time budget, achieves substantially lower MD-rates.

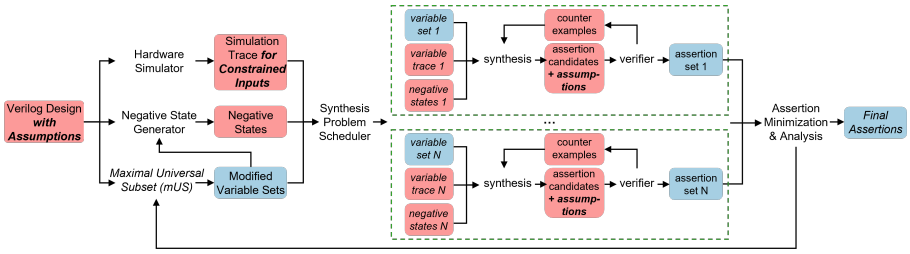
6 Case Studies

```
assume(|attacker_hitmap);
if (choice == 2'b00) begin
  assume(hitmap1 ==
      attacker_hitmap
  && hitmap2 ==
      attacker_hitmap);
  ...
end
```

(a) User-defined assumptions for a cache design with the Not Recently Used (NRU) eviction policy [13].

```
assert(
  ... &&
  (!(hitmap2 <= c2_metadata
      ) || c1_rst) &&
  (!(-((-attacker_hitmap) |
       (!attacker_hitmap)))
      || attacker_hitmap) &&
  ...);
```

(b) Verified invariants of `hitmap1` and `attacker_hitmap` generated by MAPminer under assumptions in Figure 8a.



(c) Assumptions handled by MAPminer without needing to change the toolflow. The components encoded with assumptions are ***highlighted***.

Fig. 8: External environment-guided synthesis for producing specifications with constrained inputs.

In this section, we show how the proposed work can be applied to verification tasks for realistic hardware designs. Here we take the artifact published by [13] as two case studies for evaluation. The artifact contains two hardware implementations of cache components for improved memory isolation, which use replacement policies of Not Recently Used (NRU) and Pseudo-Least Recently Used (PLRU), respectively. Verifying the properties of such hardware components is crucial, especially since they are designed for memory protection, where any design flaws could lead to serious system vulnerabilities.

In the rest of this section, we compare the assertions automatically generated by MAPminer with the manually implemented assertions [13] for evaluation. We first show how MAPminer handles external environment constraints in assumptions for producing more precise assertions, as human experts do in practice. Second, we show how MAPminer could help identify steps towards verifying an end-to-end property of a realistic hardware design.

6.1 Handling Assumptions

In the previous sections, MAPminer checks whether a property always holds for a hardware design under all the possible states of its input signals. However, not

all possible input states occur in practice because most hardware designs do not operate independently. A hardware component interacts with its external environment only under restricted input conditions. Missing such external environment constraints could lead to false negative verification results and state-space explosion during model checking.

In practice, these external environments for the hardware designs are described in the form of assumptions, which constrain the behaviors of input signals, such as timing constraints and value ranges. Here we provide an example of how MAPminer handles assumptions for generating more precise specifications.

Figure 8a shows part of the assumptions used for describing the external inputs to a cache to perform the NRU policy. The assumptions describe the scenario in which the attacker starts to work on two different machines. Figure 8b shows an assertion automatically generated by MAPminer that describes the behavior of these two signal components. We observed that these hardware states always hold under the assumption in Figure 8a but cannot be verified for any input states. This indicates that MAPminer could produce more precise specifications when input constraints for the hardware design are provided.

Supporting these assumptions in MAPminer is straightforward. Figure 8c highlights the refined components based on the assumptions in the MAPminer toolflow. In the figure, the assumptions are provided by the users in the Verilog design, restricting the states of their external inputs. These input constraints are applied to the test inputs for hardware simulation and ensure that they always hold in the simulation traces. The assumptions are also encoded into the synthesis conjecture given to the synthesis-based assertion miner, so that it generates assertions under the given assumptions. The toolflow of MAPminer does not need any changes and naturally supports the integration of assumptions. The assumptions are integrated as metadata in the components.

6.2 Generating Auxiliary Invariants for Verification

While Section 5 shows that MAPminer produces meaningful assertions, here we evaluate the utility of MAPminer for realistic verification tasks. In practice, the designer inputs end-to-end properties of the design that describe systematic hardware states. These properties often have complex data and timing dependencies, and cannot be directly verified by the SMT solver. As a result, verifying these properties requires significant manual effort by human experts to write and verify complex intermediate properties, which can then be used as auxiliary invariants to ensure the verification query for the end-to-end property is provable [13].

In this case study, we show that MAPminer can produce assertions comparable to manually written auxiliary invariants, enabling automated verification of the given end-to-end properties. Specifically, although MAPminer generates a different set of assertions from the manually implemented ones, both sets enable verification of the same end-to-end property.

Table 2 evaluates various approaches to verifying the end-to-end properties for both cache designs. We compare MAPminer with three baselines: manual k-

Table 2: Comparison of different approaches in run-time breakdown and proof boundness. MAPminer achieves the **best** results in both the end-to-end time and unbounded proof. k-Ind = k-induction. BMC = Bounded Model Checking. PDR = Property-Directed Reachability.

Benchmark	Approach	Mining (s)	Proof (s)	End-to-end (s)	Unbounded
PLRU	k-Ind (MAPminer)	139	2	**141**	✓
	k-Ind (manual)	days	5	days	✓
	BMC ($d = 20$)	-	8743	8743	✗
	PDR	-	1450	1450	✓
NRU	k-Ind (MAPminer)	129	2	**131**	✓
	k-Ind (manual)	days	3	days	✓
	BMC ($d = 20$)	-	10756	10756	✗
	PDR	-	Timeout	-	✗

induction as used in the prior work [13], Bounded Model Checking (BMC) [7] and Property-Directed Reachability (PDR) [16]. To ensure fairness, we use the same k-induction depth as the prior work. We show run-time for both specification mining and verification, and whether the proofs are bounded.

In the table, MAPminer completes the same unbounded security proof as the manual efforts, accelerating development time from days to seconds. BMC requires significantly longer runtimes and only establishes bounded guarantees, and PDR leads to a timeout of one day when proving the property for the NRU policy. This indicates that MAPminer can produce helpful invariants, automating certain verification efforts in verifying end-to-end properties for realistic hardware designs.

7 Related Work

Partitioning of search to allow parallelization is indispensable for many search tasks whose search is exorbitantly large. There are a number of related fields that have explored such methods and ideas. The work most closely related to MAPminer is HARM [12], which also parallelizes specification mining. Unlike MAPminer, HARM assigns different combinations of pattern matching and user hints to different cores. However, its pattern-matching approach limits the invariants that can be generated, leading to lower mutation detection rates compared to MAPminer. Other mining approaches applied to software, such as Daikon [10], are broadly statistical. Although such approaches can be parallelized, ensuring that each thread performs meaningful work remains challenging. Many existing specification mining tools are sequential and rely on pattern matching or static analysis, as in GoldMine [14] and ARTMine [20]. Parallel methods have also been explored in invariant synthesis. H-Houdini [9] makes use of Machine-Learning-Inspired Synthesis to decompose inductiveness checks into many small

parallelizable checks whose results can soundly compose into a global invariant. Prior work in invariant synthesis has also leveraged invalidating candidates to improve sampling [11].

There are a number of methods used in SAT solving to partition the search space. Many tools make use of *divide-and-conquer*, whereby the space is split a-priori. There are a number of techniques to overcome the difficulty of determining these splits; propagation-rate [26], partition-trees [17], symbolic partitioning of execution traces [21] and variable-level partitioning [34], all utilizing data from preprocessing formulas. *Cube-and-conquer* is a specific type of divide-and-conquer; formulas are broken down into *cubes*, with each cube fixing a subset of the variables. A separate solver for each cube is then tasked with trying to fill the free variables [6,15]. This same idea has also been used in program synthesis [22].

Lazy clause exchange [3] in portfolio solving allows clauses to be passed between partitions when they are required; similarly selective clause sharing shares 'good-looking' clauses between solvers [4,30]. Both are methods to better harness parallelism in SAT solving by sharing useful information.

8 Conclusions

We presented MAPminer, a parallel synthesis-based framework for automated specification mining in hardware designs. By combining mUS-guided variable selection with assertion minimization, MAPminer scales synthesis-based mining to realistic Verilog designs while preserving formal correctness guarantees. Compared with prior work, MAPminer achieves higher verification coverage with a moderate number of assertions, balancing assertion quality and quantity. Case studies on security-oriented cache designs demonstrate that MAPminer can automatically generate auxiliary invariants enabling unbounded verification of realistic end-to-end properties. These results highlight the practical value of MAPminer in reducing manual effort for formal hardware verification. Future work includes extending MAPminer to richer temporal specifications and tighter integration with industrial verification flows.

Acknowledgments. This work is supported by a Royal Academy of Engineering Research Fellowship RF2223-22-262 and the EPSRC Centre for Doctoral Training in Machine Learning Systems (EP/Y03516X/1). It is partially based upon work supported by the Defense Advanced Research Projects Agency (DARPA) under Agreement No. HR00112590131.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

Data-Availability Statement The artifact supporting this paper, including source code, benchmarks, and instructions for reproducing the experimental results, is available on Zenodo at `https://doi.org/10.5281/zenodo.20393223`.

References

1. EBMC. https://github.com/diffblue/hw-cbmc/
2. Circuit netlist benchmarks. https://sportlab.usc.edu/~msabrishami/benchmarks.html (2025)
3. Audemard, G., Simon, L.: Lazy clause exchange policy for parallel SAT solvers. In: International Conference on Theory and Applications of Satisfiability Testing. pp. 197–205. Springer (2014). https://doi.org/10.1007/978-3-319-09284-3_15
4. Balyo, T., Sanders, P., Sinz, C.: HordeSat: A massively parallel portfolio SAT solver. In: Theory and Applications of Satisfiability Testing – SAT 2015. pp. 156–172. Springer (2015). https://doi.org/10.1007/978-3-319-24318-4_12
5. Barbosa, H., Barrett, C.W., Brain, M., Kremer, G., Lachnitt, H., Mann, M., Mohamed, A., Mohamed, M., Niemetz, A., Nötzli, A., Ozdemir, A., Preiner, M., Reynolds, A., Sheng, Y., Tinelli, C., Zohar, Y.: cvc5: A versatile and industrial-strength SMT solver. In: Fisman, D., Rosu, G. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I. Lecture Notes in Computer Science, vol. 13243, pp. 415–442. Springer (2022). https://doi.org/10.1007/978-3-030-99524-9_24
6. Biere, A., et al.: Lingeling, Plingeling and Treengeling entering the SAT competition 2013. Proceedings of SAT competition **2013**, 1 (2013)
7. Clarke, E., Biere, A., Raimi, R., Zhu, Y.: Bounded model checking using satisfiability solving. Formal methods in system design **19**(1), 7–34 (2001). https://doi.org/10.1023/A:1011276507260
8. Dillig, I., Dillig, T., McMillan, K.L., Aiken, A.: Minimum satisfying assignments for SMT. In: International Conference on Computer Aided Verification. pp. 394–409. Springer (2012). https://doi.org/10.1007/978-3-642-31424-7_30
9. Dinesh, S., Zhu, Y., Fletcher, C.W.: H-Houdini: Scalable invariant learning. In: Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1. pp. 603–618 (2025). https://doi.org/10.1145/3669940.3707263
10. Ernst, M.D., Perkins, J.H., Guo, P.J., McCamant, S., Pacheco, C., Tschantz, M.S., Xiao, C.: The Daikon system for dynamic detection of likely invariants. Science of computer programming **69**(1-3), 35–45 (2007). https://doi.org/10.1016/j.scico.2007.01.015
11. Fedyukovich, G., Bodík, R.: Accelerating syntax-guided invariant synthesis. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 251–269. Springer (2018). https://doi.org/10.1007/978-3-319-89960-2_14
12. Germiniani, S., Pravadelli, G.: Harm: a hint-based assertion miner. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **41**(11), 4277–4288 (2022). https://doi.org/10.1109/TCAD.2022.3197525
13. Godbole, A., Ye, L., Manerkar, Y.A., Seshia, S.A.: Modelling and verification of security-oriented resource partitioning schemes. In: FMCAD. pp. 268–273 (2023). https://doi.org/10.34727/2023/isbn.978-3-85448-060-0_35
14. Hertz, S., Sheridan, D., Vasudevan, S.: Mining hardware assertions with guidance from static analysis. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **32**(6), 952–965 (2013). https://doi.org/10.1109/TCAD.2013.2241176

15. Heule, M.J.H., Kullmann, O., Wieringa, S., Biere, A.: Cube and conquer: Guiding CDCL SAT solvers by lookaheads. In: Hardware and Software: Verification and Testing – HVC 2011. pp. 50–65. Springer (2012). https://doi.org/10.1007/978-3-642-34188-5_8
16. Hoder, K., Bjørner, N.: Generalized property directed reachability. In: International Conference on Theory and Applications of Satisfiability Testing. pp. 157–171. Springer (2012). https://doi.org/10.1007/978-3-642-31612-8_13
17. Hyvärinen, A.E., Junttila, T., Niemelä, I.: Partitioning SAT instances for distributed solving. In: International Conference on Logic for Programming Artificial Intelligence and Reasoning. pp. 372–386. Springer (2010). https://doi.org/10.1007/978-3-642-16242-8_27
18. IEEE: Combining dynamic slicing and mutation operators for ESL correction (2012). https://doi.org/10.1109/ETS.2012.6233020
19. Ignatiev, A., Previti, A., Marques-Silva, J.: On finding minimum satisfying assignments. In: International Conference on Principles and Practice of Constraint Programming. pp. 287–297. Springer (2016). https://doi.org/10.1007/978-3-319-44953-1_19
20. Iman, M.R.H., Jervan, G., Ghasempouri, T.: Artmine: Automatic association rule mining with temporal behavior for hardware verification. In: 2024 Design, Automation & Test in Europe Conference & Exhibition (DATE). pp. 1–6. IEEE (2024). https://doi.org/10.23919/DATE58400.2024.10546742
21. Inverso, O., Trubiani, C.: Parallel and distributed bounded model checking of multi-threaded programs. In: Proceedings of the 25th ACM SIGPLAN symposium on principles and practice of parallel programming. pp. 202–216 (2020). https://doi.org/10.1145/3332466.3374529
22. Jeon, J., Qiu, X., Solar-Lezama, A., Foster, J.S.: Adaptive concretization for parallel program synthesis. In: Computer Aided Verification – 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18–24, 2015, Proceedings, Part II. pp. 377–394. Springer (2015). https://doi.org/10.1007/978-3-319-21668-3_22
23. Jha, S., Seshia, S.A.: A theory of formal synthesis via inductive learning. Acta Inf. **54**(7), 693–726 (Nov 2017). https://doi.org/10.1007/s00236-017-0294-5, https://doi.org/10.1007/s00236-017-0294-5
24. Kande, R., Pearce, H., Tan, B., Dolan-Gavitt, B., Thakur, S., Karri, R., Rajendran, J.: LLM-assisted generation of hardware assertions. arXiv preprint arXiv:2306.14027 (2023). https://doi.org/10.48550/arXiv.2306.14027
25. de Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: Tools and Algorithms for the Construction and Analysis of Systems – TACAS 2008. pp. 337–340. Springer (2008). https://doi.org/10.1007/978-3-540-78800-3_24
26. Nejati, S., Newsham, Z., Scott, J., Liang, J.H., Gebotys, C.H., Poupart, P., Ganesh, V.: A propagation rate based splitting heuristic for divide-and-conquer solvers. In: Theory and Applications of Satisfiability Testing – SAT 2017. pp. 251–260. Springer (2017). https://doi.org/10.1007/978-3-319-66263-3_16
27. Orenes-Vera, M., Martonosi, M., Wentzlaff, D.: Using LLMs to facilitate formal verification of RTL. arXiv preprint arXiv:2309.09437 (2023). https://doi.org/10.48550/arXiv.2309.09437
28. Raveendran, R., Bhuinya, S.: Customization of ibex risc-v processor core. Customization of Ibex RISC-V Processor Core (2021)
29. Rosser, B.J.: Cocotb: a python-based digital logic verification framework. In: Micro-electronics Section seminar. CERN, Geneva, Switzerland (2018)

30. Schreiber, D., Sanders, P.: MallobSat: Scalable SAT solving by clause sharing. Journal of Artificial Intelligence Research **80**, 1437–1495 (2024). https://doi.org/10.1613/jair.1.15827
31. Sun, C., Hahn, C., Trippel, C.: Towards improving verification productivity with circuit-aware translation of natural language to systemverilog assertions. In: First International Workshop on Deep Learning-aided Verification (2023)
32. Yan, Z., Fang, W., Li, M., Li, M., Liu, S., Xie, Z., Zhang, H.: AssertLLM: Generating hardware verification assertions from design specifications via multi-LLMs. In: Proceedings of the 30th Asia and South Pacific Design Automation Conference. pp. 614–621. ASPDAC '25, Association for Computing Machinery, New York, NY, USA (2025). https://doi.org/10.1145/3658617.3697756
33. Ye, L., Li, Y., Frankel, G., Cheng, J., Polgreen, E.: Unlocking hardware verification with oracle guided synthesis. In: Proceedings of the 25th Conference on Formal Methods in Computer-Aided Design. pp. 235–245. TU Wien Academic Press (2025). https://doi.org/10.34727/2025/isbn.978-3-85448-084-6_30
34. Zhao, M., Cai, S., Qian, Y.: Distributed SMT solving based on dynamic variable-level partitioning. In: Computer Aided Verification – 36th International Conference, CAV 2024, Montreal, QC, Canada, July 24–27, 2024, Proceedings, Part I. pp. 68–88. Springer (2024). https://doi.org/10.1007/978-3-031-65627-9_4