Improving Equality Saturation for EDA via Semantic E-Graphs

SIJIE KONG*, University of California, Santa Barbara, USA
JINGTAO XIA*, University of California, Santa Barbara, USA
DANIEL RUELAS-PETRISKO, University of Washington, USA
ZACHARY D. SISCO, The Chinese University of Hong Kong, Shenzhen, China
JONATHAN BALKIND, University of California, Santa Barbara, USA
GUS HENRY SMITH, Southmountain Research, USA

Equality saturation (eqsat) is a program optimization technique that uses syntax-based term rewriting to simultaneously explore many possible optimizations of a program, storing equivalent programs efficiently in a data structure called an e-graph. By exploring optimizations simultaneously, eqsat mitigates the *phase ordering problem,* where the order of optimizations significantly affects quality of results. Eqsat is especially promising for Electronics Design Automation (EDA), whose tools suffer from phase ordering. Previous eqsat-for-EDA efforts have focused on single tool stages; while they demonstrate significant benefits *within* a stage, they do not address phase ordering *between* stages. When we investigated the reason for their limited scope, we found that previous works struggle to implement an efficient hardware representation useful in both high-level (e.g. arithmetic optimization) and low-level (e.g. logic synthesis) tasks. The root issue is that such a representation must maintain equivalences between the high- and low-level portions of the language. While these equalities are conceptually simple—e.g., two high-level bitvectors are equal if they contain the same low-level bits—maintaining them using syntax-based rewrites alone proves inefficient in modern eqsat engines. In response, this paper makes two contributions. First, we introduce *semantic e-graphs,* an enhancement to e-graphs that improves performance of a narrow but highly useful class of semantics-based equalities. Second, we present Nextmap, a new eqsat-based hardware optimization engine whose representation uses semantic e-graphs to efficiently bridge high- and low-level hardware expressions. As a result, Nextmap simultaneously runs more EDA stages than previous eqsat-based works, more effectively mitigating phase ordering and reaching previously inaccessible optimizations. Compared with open-source and commercial tools, Nextmap provides competitive quality of results on a range of designs.

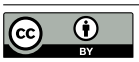
CCS Concepts: • **Hardware → Technology-mapping**; **Combinational synthesis**; **Reconfigurable logic and FPGAs**; **Hardware description languages and compilation**.

Additional Key Words and Phrases: equality saturation, e-graphs, hardware design, technology mapping

ACM Reference Format:
Sijie Kong, Jingtao Xia, Daniel Ruelas-Petrisko, Zachary D. Sisco, Jonathan Balkind, and Gus Henry Smith. 2026. Improving Equality Saturation for EDA via Semantic E-Graphs. *Proc. ACM Program. Lang.* 10, PLDI, Article 221 (June 2026), 24 pages. https://doi.org/10.1145/3808299

*Co-first author.

Authors' Contact Information: Sijie Kong, University of California, Santa Barbara, USA, sijie_kong@ucsb.edu; Jingtao Xia, University of California, Santa Barbara, USA, jingtaoxia@ucsb.edu; Daniel Ruelas-Petrisko, University of Washington, Seattle, USA, petrisko.bsg@gmail.com; Zachary D. Sisco, The Chinese University of Hong Kong, Shenzhen, China, zsisco@cuhk.edu.cn; Jonathan Balkind, University of California, Santa Barbara, USA, jbalkind@ucsb.edu; Gus Henry Smith, Southmountain Research, Seattle, USA, gus@southmountain.ai.



ACM 2475-1421/2026/6-ART221
https://doi.org/10.1145/3808299

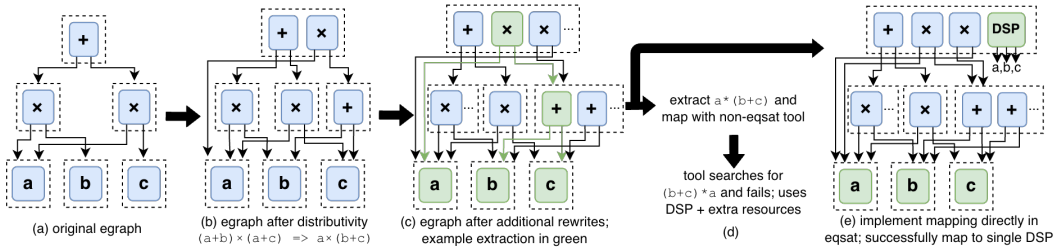


Fig. 1. Running optimization and mapping stages simultaneously leads to higher QoR. (a)–(c): An initial hardware design is optimized via ROVER-style rewrites [9]. (d): If mapping stage is not implemented in eqsat, we must extract a single design. Non-eqsat EDA tools are brittle to syntax, leading to lower quality of results (QoR). (e): By implementing mapping in eqsat and running it simultaneously with optimization, we keep all previously-seen designs and increase ease of mapping, leading to higher QoR.

1 Introduction

Equality saturation (eqsat) [19, 27, 31, 36] is a term rewriting technique that uses an efficient data structure called an *e-graph* to simultaneously capture every version of a program produced during rewriting. Figure 1 (a) shows how an e-graph extends a program's graph structure by grouping equivalent *e-nodes* (solid boxes) into *e-classes* (dotted boxes). Unlike traditional rewriting, which destructively replaces matched nodes, eqsat rewrites non-destructively add new e-nodes to the corresponding e-class, as shown in fig. 1 (b) and (c). Arrows from e-nodes point to e-classes, not e-nodes, since any e-node in a class can serve as its representative. After rewriting, a single program is *extracted* from the e-graph by choosing one e-node per e-class, reducing it to a standard program graph (fig. 1 (c)). Modern eqsat engines include `egg` [31] and `egglog` [36], and eqsat has been applied to floating point optimization [21], CAD simplification [18], database query optimization [30], tensor program optimization [33], SIMD mapping [29], and many more [37].

Eqsat is especially enticing for Electronics Design Automation[1] (EDA), where the phase ordering problem is pervasive. While "EDA" spans the entire tool stack from design to silicon, we focus on the *synthesis* portion, in which a design is lowered from a high-level representation into low-level, backend-specific hardware primitives. Synthesis tools suffer from phase ordering: their optimization stages must be carefully ordered to produce high quality of results (QoR). For example, in the open-source synthesis tool Yosys [32], the script for one hardware backend invokes over fifty separate passes which developers must order precisely.

While recent eqsat-for-EDA projects have shown significant gains by applying eqsat *within* single stages, each tool has also conspicuously limited itself to one single stage. ROVER [9], E-Syn [5], and E-Morphic [4] all target specific optimization stages. ROVER [9] targets arithmetic and structural optimization, such as restructuring multipliers and adders. E-Syn [5] and E-Morphic [4] apply eqsat to Boolean optimization. Meanwhile, EqMap [14] and Churchroad [24] target *technology mapping*, in which a design is converted into backend-specific primitives. Both target FPGAs—reconfigurable hardware platforms which implement hardware designs by connecting together small, pre-fabricated primitives. EqMap [14] maps Boolean expressions to Look-up Tables (LUTs), while Churchroad [24] maps arithmetic expressions to small, specialized arithmetic primitives called Digital Signal Processors (DSPs). None of these tools run optimization and mapping simultaneously.

By limiting their scope to single stages, these tools fail to address phase ordering *between* stages. Consider the example in fig. 1: we optimize a hardware design with the eqsat-powered ROVER,

[1]An all-encompassing descriptor for the tools which compile hardware design code into actual hardware.

then map it to an FPGA using a non-eqsat technology mapper. Although ROVER eliminates phase ordering *within* the arithmetic optimization stage, when it hands off to the non-eqsat technology mapper, it must extract a single design. Suppose ROVER extracts `a*(b+c)`. Technology mapping is notoriously syntax-dependent [25]; if the mapper only recognizes `(b+c)*a`, it will fail to map this pattern efficiently. The gains from using eqsat for optimization are lost when we must leave eqsat for mapping. This is *problem 1:* existing eqsat-based approaches still suffer from phase ordering between EDA stages.

If instead we bring technology mapping into eqsat by combining ROVER's optimization rewrites with Churchroad's mapping rewrites, the mapper sees *all* equivalent expressions in the e-graph, not just the one we extracted. It may match on `(b+c)*a`, `a*(b+c)`, or any other variation present in the e-graph, more easily finding a mapping to a single DSP and yielding higher QoR. This is our solution to problem 1: run more EDA stages simultaneously.

If there is such an obvious benefit to running stages simultaneously, then why have previous eqsat-based tools limited their scope? When we investigated, we found that previous tools struggle to implement a hardware representation which can bridge between the *word-level* representations of tools like ROVER and Churchroad and the *bit-level* representations of tools like E-Syn, E-Morphic, and EqMap. ROVER and Churchroad view hardware designs at the *word* level, in which operators like multiplication and addition operate over bitvectors representing multi-bit numbers. E-Syn, E-Morphic, and EqMap, on the other hand, view designs at the *bit* level, e.g. optimizing and mapping the single-bit and, or, and xor gates which implement multi-bit addition or multiplication.

It is straightforward to *represent* both the word and bit levels. Consider a simple example of a three-bit bitvector s, which is the result of adding the two-bit bitvectors a and b. At the word level, our representation can express the bitvectors as $s = a + b$; at the bit level, it can express each bit as and (&) and xor ($\oplus$) gates, e.g. $s[1] = (a[1] \oplus b[1]) \oplus (a[0] \ \ \& \ \ b[0])$.

Representation alone is not enough, though: to be useful, the two levels must be *bridged* by equivalences, and maintaining these equivalences efficiently is difficult with syntax-based equality alone. Syntactically, bit and word levels are connected via extraction and concatenation operators. For example, $s[1:0] = \{s[1], s[0]\}$ connects the lower two bits of our word- and bit-level expressions. Imitating Verilog syntax, $s[1:0]$ selects from bitvector s from index 1 (the MSB) down to index 0 (the LSB) inclusive, and $\{s[1], s[0]\}$ concatenates the two bits $s[1]$ and $s[0]$. But the number of such equivalences grows exponentially with bitvector size: 3-bit s can be written as $\{s[2], s[1], s[0]\}$, $\{s[2:1], s[0]\}$, $\{s[2], s[1:0]\}$, or $\{s[2:0]\}$, while 4-bit bitvectors have 8 representations, and so on. This is *problem 2:* bridging between word and bit levels via syntactic equality is unwieldy, as the number of equivalences to maintain grows exponentially.

However, we do not need to rely only on *syntactic* equality when attempting to bridge bit and word levels—we can also use *semantic* equality. Despite the fact that $s[1:0]$ and $\{s[1], s[0]\}$, are not syntactically identical, it is plain to see that both expressions share the same underlying bits, and are thus equivalent. This is our solution to problem 2: instead of enumerating all possible syntactic equalities, determine bitvector equivalence by the equality of their underlying bits.

This is not the end of our problems, however. Computing the underlying bits of each expression is straightforward using *e-class analyses* in `egg` and `egglog`, which attach analysis data to e-classes. But once we have the bits, we must maintain the following invariant for all bitvector expressions e_1 and e_2:

$$\text{the bits of } e_1 \text{ equal the bits of } e_2 \quad \Longleftrightarrow \quad e_1 = e_2 \tag{1}$$

As section 2 details, maintaining this invariant is inefficient in current eqsat engines. In the forward direction, we must detect and merge all pairs (e_1, e_2) whose underlying bits are identical (*bitvector extensionality*). In the backward direction, when e_1 and e_2 are found equal, we must propagate that

equality to each pair of underlying bits (*injectivity of the bitvector constructor*). As section 2 will show, though both directions are expressible with existing eqsat engines, they are not performant. This is *problem 3:* maintaining our semantic equality invariant is inefficient in existing eqsat engines.

Detecting expression equivalence via domain-specific semantic information is a broadly useful pattern. In fact, the idea has already proven useful in the SMT domain, most notably through CC(X) [8], which combines congruence closure with a solvable theory using semantic values as class representatives. Previous work [38] has hypothesized the utility of bringing the same pattern to eqsat, for domains like strings, sets, multisets, and polynomials. Given the general utility of this pattern, we believe eqsat theory and engine implementation should be adjusted to support it.

In response to problem 3, we introduce *semantic e-graphs* (SEGs)—a modification of e-graphs that efficiently supports equality checking based on *semantic* rather than *structural* equality. Semantic e-graphs enhance the performance of eq. (1) for a broad category of domains.

On top of semantic e-graphs, we build Nextmap,[2] a new eqsat-powered EDA framework whose representation efficiently bridges bit- and word-level expressions. This allows Nextmap to run optimization and mapping at multiple levels of granularity simultaneously, further mitigating phase ordering in EDA. Nextmap delivers better QoR than previous eqsat-based and open-source tools, and competitive QoR against commercial tools.

In summary, the core contributions of this work are:

(1) *Semantic e-graphs,* an e-graph variant supporting efficient semantics-based equality checking
(2) *Nextmap,* a new eqsat-based framework for EDA tasks leveraging semantic e-graphs

The rest of this paper proceeds as follows. Section 2 walks through an extended example demonstrating the shortcomings of existing eqsat engines for EDA and how semantic e-graphs address them. Section 3 formalizes the semantic e-graph. Section 4 instantiates semantic e-graphs for hardware, and section 5 presents the Nextmap system. Section 6 evaluates our approach. Section 7 discusses related work.

2 Overview: Addressing egglog Inefficiencies with Semantic E-Graphs

In this section, we walk through concrete examples of the issues encountered when implementing an efficient hardware representation in `egglog`, and show how semantic e-graphs address them.

In `egglog`, we first define the language to optimize and transform. Imagine a simple hardware design language with nodes `(Var name bw)` for a variable bitvector of bitwidth `bw`, `(BV val bw)` for a constant bitvector, `(And e1 e2)` (and others) for operators, `(Extract hi lo e)` for extraction from index `hi` down to `lo` inclusive, and `(Concat e1 e2)` for concatenation.

To discover equalities and implement optimizations, we use the `(rewrite lhs rhs)` command, which finds expressions matching `lhs` and inserts the corresponding `rhs` into the matched expression's e-class. Recall 3-bit `s` from section 1; to discover that `(Concat (Extract 2 1 s) (Extract 0 0 s))` equals `(Concat (Extract 2 2 s) (Extract 1 0 s))`, we can write rewrites such as `(rewrite s (Concat (Extract n-1 i) (Extract i-1 0)))`, where `n` is the length of `s`. This rewrite, combined with other rewrites such as the associativity of `Concat` and distributivity of `Concat` and `Extract`, will enumerate all equivalent forms of these expressions. But as noted in section 1, the number of forms grows exponentially with bitwidth `n`, making syntax-based rewrites alone unwieldy.

We need not rely on syntax alone, however. For both expressions above, the underlying bits are $[s_2, s_1, s_0]$, where s_i is the ith bit of `s`. `egglog` supports attaching semantic information to expressions via *e-class analyses* using the `(rule conds analysis)` command, which adds the e-class analysis data `analysis` when the conditions in `conds` are met. We implement an analysis

[2] Nextmap is open source at https://github.com/UCSBarchlab/nextmap.

(`HasBits e` $[b_{n-1}, \ldots, b_0]$) stating that expression `e` has underlying bits $[b_{n-1}, \ldots, b_0]$, where each b_i is a reference to an e-class representing an individual bit. As with bitvectors, individual bits can be variables or constant 0/1. The analysis is defined by the following rules:

if `(Var name bw)` *then* `(HasBits (Var name bw)` $[b_{\texttt{bw}-1}, \ldots, b_0]$`)`
where b_i *is a fresh symbolic bit*

if `(BV val bw)` *then* `(HasBits (BV val bw) (to-bits val bw))`

if `(Concat a b)`, `(HasBits a` $a_N \ldots a_0$`)`, `(HasBits b` $b_M \ldots b_0$`)` *then* `(HasBits (Concat a b)` $[a_N \ldots a_0 b_M \ldots b_0]$`)`

if `(Extract h l e)`, `(HasBits e` $e_N \ldots e_0$`)` *then* `(HasBits (Extract h l e)` $[e_h \ldots e_l]$`)`

(where `to-bits` converts a value into a vector of concrete 0s and 1s in the e-graph.)

Instead of discovering equality syntactically, we can use `HasBits` to write a general rule about bitvector equality (restated from section 1):

$$\text{the bits of } e_1 \text{ equal the bits of } e_2 \quad \Longleftrightarrow \quad e_1 = e_2 \tag{1}$$

where equality, in both cases, is captured by e-node equivalence. The forward direction (bitvector extensionality) ensures that bitvectors with the same bits are automatically *unified* (their e-classes merged), with no syntax-based rewrites needed. The backward direction (injectivity of the bitvector constructor) ensures that bitvectors found equal by other means (e.g. other rewrites) have their underlying bits unified pairwise. This is exactly what we mean by bridging between word and bit levels: the forward direction propagates bit-level equality to the word level, and the backward direction propagates word-level equality to the bit level.

Unfortunately, implementing eq. (1) in `egglog` reveals inefficiencies in both directions. The semantic e-graph addresses both.

The forward direction can be implemented in `egglog` as:

if `(HasBits e1 b)`, `(HasBits e2 b)` *then* `(union e1 e2)`

where `union` merges the e-classes of two expressions. This rule is inefficient: for n expressions with identical bits b, it matches $O(n^2)$ times when $O(n)$ matches will suffice. For example, three expressions e_1, e_2, e_3 sharing bits b produce nine matches (e_1, e_1), (e_1, e_2), ..., (e_3, e_3), when only two are needed to transitively union them, e.g. (e_1, e_2) and (e_2, e_3).

The semantic e-graph's insight is that no rule is needed here, because e-graphs already maintain a similar property defining equality between expressions:[3]

$$\text{the e-class id of } e_1 \text{ equals the e-class id of } e_2 \quad \Longleftrightarrow \quad e_1 = e_2$$

Swapping e-class ids for bits makes the invariants identical. We reuse the same machinery, replacing opaque e-class ids (EIDs; often unsigned integers) with richer *semantic* EIDs (SEIDs)—here, the vector of bits computed by the rules above. With bits as the e-class id, every expression with the same computed bits is *automatically* in the same e-class.

The backward direction states that if two expressions are found equal, their bits should be unified. It can be implemented as

if `(HasBits e b1)`, `(HasBits e b2)` *then* `(union` b_{1_i} b_{2_i}`)` *for each pair* (b_{1_i}, b_{2_i})

This rule also faces the $O(n^2)$ matching issue for n equivalent expressions. Moreover, implementing the "for each pair" part is inefficient in `egglog`, requiring iterative rules that mimic loops.

Unlike the forward direction, this inefficiency has no one-size-fits-all solution. The union procedure depends on the structure of the thing being unioned—in this case, our vector of bits. In our case, we union the bits pairwise, but if our semantic identifier had a different structure—e.g. a tree—we would need a different union procedure.

[3]To be precise, this is $\equiv_{\text{node}}$ in the egg paper [31], which defines equality over e-nodes.

The semantic e-graph's solution is to let the user define a custom decomposition routine that tells the engine exactly which expressions to union. For example, suppose expressions e_1 and e_2 are found equal by some rewrite, and we have (`HasBits` e_1 $[a_2, a_1, a_0]$) and (`HasBits` e_2 $[b_2, b_1, b_0]$). The decomposition routine returns the pairs (a_2, b_2), (a_1, b_1), (a_0, b_0), and the engine unions each pair directly—no rules or pattern matching required. Other SEID structures could return other patterns. By allowing user-defined decomposition, semantic e-graphs sidestep the need to implement merge routines as rules.

In summary, semantic e-graphs have two key insights for efficiently identifying semantic, not just syntactic, equalities. First, replace opaque e-class ids with semantically rich SEIDs while reusing the same id-tracking machinery. Second, allow users to define custom merge routines on structured types like bitvectors. In the next section, we formalize semantic e-graphs.

3 Semantic E-Graphs

An *e-graph* [19, 27, 31, 36] is a data structure for compactly representing many equivalent expressions. It organizes expressions into *e-classes*, each of which contains a set of *e-nodes* representing equivalent terms. E-graphs maintain *congruence closure*: if two e-nodes have the same function symbol and their corresponding children belong to the same e-classes, then the two e-nodes are equivalent and are merged into the same e-class. In standard e-graphs, each e-class is identified by an opaque identifier, which serves as a representative for all expressions in the class.

Semantic e-graphs (SEGs) extend standard e-graphs by replacing opaque e-class identifiers with *semantic identifiers* drawn from a user-defined semantic domain. A semantic identifier gives a domain-specific abstraction of an e-class: for example, 1 + 3 and 2 + 2 can both be assigned the identifier 5, so they are recognized as equivalent and merged automatically.

In this section, we formalize semantic e-graphs and describe the maintenance algorithm needed to support semantic identifiers.

3.1 Semantic Domains

Semantic e-graphs represent well-typed terms from a simply typed, first-order, monomorphic term language. Each function symbol is assigned a fixed type signature, and every well-formed term has a unique type τ. We maintain the invariant that each e-class contains only e-nodes of a single type; we call this the type of the e-class. We omit type annotations when they are clear from context.

For each type τ appearing in e-nodes, we associate two semantic domains:

- a *concrete semantic domain* $\mathbb{C}_\tau$: fully interpreted values;
- a *symbolic semantic domain* $\mathbb{S}_\tau$: opaque placeholders used otherwise.

For simplicity, we write the combined semantic domain as $\mathbb{D}_\tau = \mathbb{C}_\tau \cup \mathbb{S}_\tau$. The two domains are disjoint by construction. Equality on $\mathbb{S}_\tau$ is identity: two symbolic values are equal iff they are the same placeholder. Equality on $\mathbb{C}_\tau$ is user-defined and assumed decidable; it must be sound with respect to the intended semantics, but need not be complete.

We distinguish a class of *composite types* κ (e.g., tuples and vectors). Composite domains are equipped with a fixed set of *domain constructors*, which define the internal structure of values in $\mathbb{C}_\kappa$. These are distinct from function symbols in the term language: semantic constructors $[\![f]\!]$ compute values, whereas domain constructors provide structured representations that support decomposition. For such types, the concrete domain $\mathbb{C}_\kappa$ consists of structured values of the form $K(v_1, \ldots, v_n)$, where K is a domain constructor of $\mathbb{C}_\kappa$ with component types $\tau_1, \ldots, \tau_n$ and each $v_i \in \mathbb{D}_{\tau_i}$. We assume that for each such constructor K, there exist projections π_i such that for all i, $\pi_i(K(v_1, \ldots, v_n)) = v_i$. In particular, this implies injectivity:

$$K(v_1, \ldots, v_n) = K(w_1, \ldots, w_n) \implies v_i = w_i \quad \forall i.$$

This injectivity property applies only to $\mathbb{C}_\kappa$; the symbolic domain $\mathbb{S}_\kappa$ contains opaque placeholders with no such structure.

Values of all other non-composite types are treated as opaque; no decomposition property is assumed.

Definition 3.1 (Precision Order)**.** For each type τ, we define a relation $\sqsubseteq_\tau$ on $\mathbb{D}_\tau$ capturing the precision of values: symbolic values are less precise than concrete values, and two distinct symbolic values are minimal elements and incomparable with each other.

If τ is not a composite type, then $\sqsubseteq_\tau$ is defined by

$$x \sqsubseteq_\tau y \iff (x = y) \ \vee \ (x \in \mathbb{S}_\tau \wedge y \in \mathbb{C}_\tau).$$

If κ is a composite type, then $\sqsubseteq_\kappa$ is defined by

$$x \sqsubseteq_\kappa y \iff (x = y) \vee (x \in \mathbb{S}_\kappa \wedge y \in \mathbb{C}_\kappa) \vee \Big(x = K(v_1, \ldots, v_n),\ y = K(w_1, \ldots, w_n),\ \forall i,\ v_i \sqsubseteq_{\tau_i} w_i\Big),$$

where K is a domain constructor of $\mathbb{C}_\kappa$ as defined above. Two concrete composite values are comparable only when they share the same constructor and their components are comparable pointwise.

Lemma 3.1. *Each $\sqsubseteq_\tau$ is a partial order.*

3.2 Semantic E-Graph Structure

Figure 2 presents the syntax of semantic e-graphs. We extend the standard e-graph syntax [31] by associating each e-class with a *semantic identifier* drawn from the appropriate domain $\mathbb{D}_\tau$.

types	$\tau,\ \kappa$	semantic domains	$\mathbb{C}_\tau, \mathbb{S}_\tau, \mathbb{D}_\tau$
function symbols	$f, g : \tau_1 \times \cdots \times \tau_m \longrightarrow \tau$	semantic identifier	$\delta \in \mathbb{D}_\tau$
terms	$t ::= f \mid f(t_1, \ldots, t_m)$		$m \geq 1$
e-nodes	$n ::= f \mid f(\delta_1, \ldots, \delta_m)$		$m \geq 1$
e-classes	$c ::= \{n_1, \ldots, n_k\} \triangleright \delta$		$k \geq 1$

Fig. 2. Syntax and metavariables for semantic e-graphs. Each e-node refers to child e-classes $c_i \triangleright \delta_i$, and each e-class is annotated with a semantic identifier $\delta \in \mathbb{D}_\tau$.

Formally, an e-class is a pair (N, δ) consisting of its e-node set and semantic identifier, but we write $\{n_1, \ldots, n_k\} \triangleright \delta$ for brevity. We also write $c \triangleright \delta$ to indicate that semantic identifier δ is associated with e-class c. We write $[\![n]\!]$ for the semantic identifier of the e-class containing n.

To construct and maintain a semantic e-graph, we require a family of semantic operations associated with the term language.

Definition 3.2 (Semantic Constructor)**.** For each function symbol $f : \tau_1 \times \cdots \times \tau_m \to \tau$, a *semantic constructor* for f is a function $[\![f]\!] : \mathbb{D}_{\tau_1} \times \cdots \times \mathbb{D}_{\tau_m} \to \mathbb{D}_\tau$ with the following behavior:

- *Semantic case.* If $[\![f]\!]$ can compute a concrete result from its inputs, it returns a value in $\mathbb{C}_\tau$.
- *Fallback case.* Otherwise, $[\![f]\!]$ returns a symbolic element of $\mathbb{S}_\tau$.

In the semantic case, $[\![f]\!]$ exposes equalities that would otherwise require explicit rewrite rules or e-class analysis; for a composite type κ, the result is a value in $\mathbb{C}_\kappa$ whose components may themselves be symbolic. In the fallback case, $[\![f]\!]$ returns an opaque identifier analogous to the e-class IDs in a classical e-graph. In the formalization, we model this symbolically as the uninterpreted term $f(\delta_1, \ldots, \delta_m)$ itself, avoiding any need for effectful fresh name generation.

Example 3.1 (Arithmetic Constant Folding). Consider the e-nodes $n_1 = +(c_1 \triangleright 1,\ c_4 \triangleright 4)$ and $n_2 = +(c_2 \triangleright 2,\ c_3 \triangleright 3)$, where each c_i is a constant e-class with semantic identifier in $\mathbb{C}_{\mathsf{Int}}$. Since constants interpret to themselves, the constructor evaluates $[\![n_1]\!] = [\![+]\!](1, 4) = 5$ and $[\![n_2]\!] = [\![+]\!](2, 3) = 5$. Both identifiers lie in $\mathbb{C}_{\mathsf{Int}}$, so the two e-nodes are immediately assigned the same semantic identifier 5. Thus, n_1 and n_2 are inserted into the same e-class without requiring any rewrite rules or e-analysis.

Now consider the e-node $n = +(c_1 \triangleright v_1,\ c_2 \triangleright 4)$, where $v_1 \in \mathbb{S}_{\mathsf{Int}}$ is not concretely interpretable. Because the semantic constructor cannot evaluate + on these inputs, it falls back and produces a symbolic identifier $v_3 \in \mathbb{S}_{\mathsf{Int}}$, yielding a new e-class $c_3 \triangleright v_3$. This matches the behavior of a traditional e-graph: the result is treated as an uninterpreted value.

Assumption 3.1 (Constructor Precision Monotonicity)**.** For every function symbol $f : \tau_1 \times \cdots \times \tau_m \to \tau$ and for all $x_i, y_i \in \mathbb{D}_{\tau_i}$, if $x_i \sqsubseteq_{\tau_i} y_i$ for all i, then

$$[\![f]\!](x_1, \ldots, x_m) \sqsubseteq_\tau [\![f]\!](y_1, \ldots, y_m).$$

This states that making inputs more precise cannot make the result less precise. In particular, once $[\![f]\!]$ produces a concrete result, further refinement of the arguments must preserve concreteness.

Lemma 3.2 (Ascending-Chain Condition)**.** *For each type τ, $(\mathbb{D}_\tau, \sqsubseteq_\tau)$ satisfies the ascending-chain condition.*

Proof. For non-composite types, every strict refinement goes from a symbolic value to a concrete value, and concrete values are maximal, so all chains are finite. For a composite type κ, the claim follows by structural induction: every value in $\mathbb{C}_\kappa$ is generated by a finite, acyclic constructor tree, and each component domain satisfies the ascending-chain condition by hypothesis. □

Equality between two concrete composite values $x, y \in \mathbb{C}_\kappa$ is decomposed pointwise via the projections π_i:

$$\Delta_\kappa(x, y) = \{(\pi_1(x), \pi_1(y)),\ \ldots,\ (\pi_n(x), \pi_n(y))\}.$$

Each pair $(\pi_i(x), \pi_i(y))$ should be recursively merged, rather than merging the two composite identifiers directly.

Example 3.2 (Vector Decomposition). Let $\mathbb{D}_{\mathrm{Vec}_3}$ denote the domain of length-3 vectors with projections π_1, π_2, π_3. Consider two identifiers in $\mathbb{C}_{\mathrm{Vec}_3}$: $\delta_1 = [\delta_{1,1},\ \delta_{1,2},\ \delta_{1,3}]$ and $\delta_2 = [\delta_{2,1},\ \delta_{2,2},\ \delta_{2,3}]$. Decomposition via projections yields:

$$\Delta_{\mathrm{Vec}_3}(\delta_1, \delta_2) = \{(\delta_{1,1}, \delta_{2,1}),\ (\delta_{1,2}, \delta_{2,2}),\ (\delta_{1,3}, \delta_{2,3})\}.$$

Each pair $(\pi_i(\delta_1), \pi_i(\delta_2))$ represents a required equality between the corresponding components. Thus, equality of δ_1 and δ_2 implies equality of all their aligned components.

Intuitively, semantic construction produces semantic values during insertion, while semantic decomposition exposes additional equalities through structured values that must be maintained by the e-graph. The partial order over semantic domains provides a notion of refinement. Together, these mechanisms allow the e-graph to reason about both syntactic and interpreted equality within a single incremental framework.

Definition 3.3 (Semantic E-Graph)**.** A *semantic e-graph* is a set of typed e-classes where each e-class is identified by a semantic identifier from $\mathbb{D}_\tau$, rather than an opaque id. It is equipped with a semantic constructor $[\![f]\!]$ for each function symbol, which computes the semantic identifier of a parent node from those of its children. For composite types, equality between identifiers is decomposed componentwise via the projections π_i.

3.3 Semantic E-Graph Operations

Semantic identifiers are not merely auxiliary metadata: they are first-class participants in equivalence maintenance. One could imagine storing semantic values in an external map indexed by opaque e-class ids, but such a map would be passive — it could not drive merging on its own. In our design, two e-classes with the same semantic identifier are merged directly, and composite identifiers propagate equality to their components through rebuild. The two primary mutating operations are `sadd` and `smerge`, both relying on a deferred *rebuild* phase that restores congruence closure and semantic consistency.

For each type τ, we maintain a union-find structure W_τ over $\mathbb{D}_\tau$ to represent equalities among semantic identifiers. We write $\texttt{find}_\tau$ and $\texttt{union}_\tau$ for its operations: $\texttt{find}_\tau(\delta)$ returns the current canonical representative of δ, and $\texttt{union}_\tau(\delta_1, \delta_2)$ merges the equivalence classes containing δ_1 and δ_2.

Insertion. Let τ be a result type, and let $n = f(\delta_1, \ldots, \delta_m)$ be an e-node of type τ. The operation $\texttt{sadd}_\tau(n)$ proceeds as follows:

(1) Canonicalize each child: let $\delta_i' = \texttt{find}_{\tau_i}(\delta_i)$ and form $n' = f(\delta_1', \ldots, \delta_m')$. If n' already exists in some e-class c, return c.
(2) Otherwise, compute $v = [\![f]\!](\delta_1', \ldots, \delta_m') \in \mathbb{D}_\tau$.
(3) Create a new e-class containing n' and annotate it with v.

Thus, insertion performs semantic interpretation eagerly, while deferring any induced merging to rebuild.

Merge. For each type τ, the operation $\texttt{smerge}_\tau(\delta_1, \delta_2)$ with $\delta_1, \delta_2 \in \mathbb{D}_\tau$ requests that the two identifiers be made equal.

If $\tau = \kappa$ is a composite type and $\delta_1, \delta_2 \in \mathbb{C}_\kappa$, then semantic equality is propagated structurally via the projections π_i. Instead of directly invoking $\texttt{union}_\kappa$, we recursively invoke $\texttt{smerge}_{\tau_i}(\pi_i(\delta_1), \pi_i(\delta_2))$ for each component i. After the components are merged, the updates are propagated to parents during rebuild.

Otherwise, that is, when τ is not a composite type, or when at least one of δ_1, δ_2 is symbolic, $\texttt{smerge}_\tau$ simply invokes $\texttt{union}_\tau(\delta_1, \delta_2)$ in W_τ. No decomposition occurs; if one identifier is concrete and the other symbolic, $\texttt{union}_\tau$ records the concrete value as canonical.

Example 3.3 (Complex Numbers). Assume Complex is a composite type with domain constructor $\mathsf{cmplx} : \mathsf{Real} \times \mathsf{Real} \to \mathsf{Complex}$ and projections $\pi_1(\mathsf{cmplx}(a, b)) = a$, $\pi_2(\mathsf{cmplx}(a, b)) = b$. Consider two e-classes $c_1 \triangleright \mathsf{cmplx}(a_1, b_1)$ and $c_2 \triangleright \mathsf{cmplx}(a_2, b_2)$.

Invoking $\texttt{smerge}_{\mathsf{Complex}}\big(\mathsf{cmplx}(a_1, b_1), \mathsf{cmplx}(a_2, b_2)\big)$ applies projections to yield the component obligations $\texttt{smerge}_{\mathsf{Real}}(a_1, a_2)$ and $\texttt{smerge}_{\mathsf{Real}}(b_1, b_2)$. Rebuild then propagates the canonical representatives a^* and b^* upward, merging the two original e-classes with canonical identifier $\mathsf{cmplx}(a^*, b^*)$.

This example illustrates the two directions of propagation: projection-based decomposition propagates equality downward to components, while rebuild propagates canonical identifiers upward to composite parents.

Rebuild. As in many equality-saturation engines, semantic e-graphs use deferred rebuild for efficiency. During rebuild, every occurrence of a semantic identifier is rewritten to its current canonical representative $\texttt{find}_\tau(\delta)$, including occurrences inside composite identifiers. Any parent e-node affected by such changes is then re-canonicalized and rehashed, which may trigger congruence merges exactly as in a standard e-graph. In addition, its semantic identifier is recomputed using the corresponding semantic constructor. By assumption 3.1, such recomputation can only yield a more

precise identifier. If the recomputed identifier is already associated with another e-class, $\texttt{union}_\tau$ merges the two e-classes in W_τ.

Rebuild repeats this process until no union-find structure changes and no parent requires reprocessing. At that point, the semantic e-graph has reached a fixpoint: there are no pending merge obligations and no further semantic refinement is possible. In practice, we maintain a hash map from semantic identifiers to e-classes to detect congruence merges in $O(1)$ expected time. This relies on hash functions consistent with domain equality for concrete values.

Lemma 3.3 (Canonical Representative)**.** *After every* `sadd`*,* `smerge`*, and rebuild step, the canonical representative of each equivalence class in W_τ is the maximally precise element known for that class with respect to $\sqsubseteq_\tau$.*

Proof. For non-composite types, if the class contains a concrete value, it is the unique most precise element: merging two distinct concrete values would violate the soundness of the semantic relation R_{semantic} and therefore does not occur. If all elements are symbolic, they are incomparable, and any one may serve as canonical representative, matching the behavior of a traditional e-graph. For composite types, two concrete values are never merged directly via $\texttt{union}_\kappa$. Instead, `smerge` decomposes them via projections, recursively merging each component. During rebuild, the semantic constructor recomputes the composite representative from the canonical representatives of its components. Since both original values share the same domain constructor K, the reconstructed composite identifiers from both sides are identical after component merges, naturally yielding a single canonical representative. By induction, each component's canonical representative is maximally precise, so the reconstructed composite is also maximally precise. □

3.4 Rewriting

Rewriting in a semantic e-graph follows the same overall structure as in a classical e-graph. Rewrite rules are statically type-checked and required to be type-preserving: the left- and right-hand sides must have the same type. In a simply-typed, monomorphic setting, each function symbol has a fixed type signature, so the type of any e-node is determined by its function symbol alone, and no runtime type checks are needed.

Given a rewrite rule $\ell \to r$ where both sides have type τ, e-matching is purely syntactic and unchanged from a classical e-graph: `ematch` traverses the e-graph structure, matching e-nodes by function symbol and binding each pattern variable to a matched e-class. No semantic information is consulted during matching.

For each match, `ematch` produces a substitution σ mapping each pattern variable to the semantic identifier of the matched e-class. Applying σ to the right-hand side r yields $r[\sigma]$, a term where each pattern variable is replaced by a semantic identifier. This term is then inserted via `sadd`, which recursively processes each e-node bottom-up and invokes the semantic constructor at each step; semantic interpretation occurs entirely at this stage. The resulting e-class is then equated with the matched e-class c via $\texttt{smerge}_\tau\big(\delta_c,\ \texttt{sadd}_\tau(r[\sigma])\big)$, where δ_c is the semantic identifier of c. Semantic interpretation thus occurs entirely within `sadd` and `smerge`, not during matching.

A semantic e-graph is *saturated* when no rewrite produces a new e-node and rebuild reaches a fixpoint.

3.5 Properties

The semantic e-graph derives equalities from two user-defined sources: the *syntactic relation* $R_{\text{syntactic}}$, induced by rewrite rules, and the *semantic relation* R_{semantic}, induced by semantic construction and semantic decomposition. More concretely, R_{semantic} includes (i) equalities discovered by `sadd`, when two terms are assigned the same semantic identifier by a constructor $[\![f]\!]$, and (ii)

equalities generated by `smerge`, when merging concrete composite identifiers triggers recursive merges via projections π_i.

Theorem 3.1 (Algorithmic Soundness)**.** *All equalities maintained by the semantic e-graph are contained in the least congruence containing* $R_{\text{syntactic}} \cup R_{\text{semantic}}$.

Proof. We argue by induction over the execution trace. Each maintained equality is either introduced by a rewrite, `sadd`, or `smerge`, or inherited from previously maintained equalities through rebuild.

- *Rewrite.* Each rewrite step introduces an equality in $R_{\text{syntactic}}$ by definition.
- `sadd`. When two terms are assigned the same semantic identifier by a constructor $[\![f]\!]$, the resulting equality is in R_{semantic} by definition.
- `smerge`. When merging two concrete composite identifiers, decomposition via projections π_i generates component equalities. We define R_{semantic} to include equalities induced by semantic decomposition, so these component equalities are in R_{semantic}.
- *Rebuild.* Rebuild performs only congruence closure: it propagates canonical identifiers upward and merges parents with identical structure, introducing no new source of equality beyond closure.

The first three cases contribute edges in $R_{\text{syntactic}} \cup R_{\text{semantic}}$ by definition, while rebuild contributes only congruence closure. Therefore all maintained equalities lie in the least congruence containing $R_{\text{syntactic}} \cup R_{\text{semantic}}$. □

Corollary 3.1 (Semantic Soundness)**.** *Let* $\equiv \subseteq \mathcal{T} \times \mathcal{T}$ *be a ground-truth congruence on well-typed terms, where* $\mathcal{T}$ *is the set of well-typed terms. If* $R_{\text{syntactic}} \subseteq \equiv$ *and* $R_{\text{semantic}} \subseteq \equiv$*, then all equalities maintained by the semantic e-graph are sound with respect to* $\equiv$.

Proof. By theorem 3.1, all maintained equalities lie in the least congruence containing $R_{\text{syntactic}} \cup R_{\text{semantic}}$. Since $\equiv$ is itself a congruence and $R_{\text{syntactic}} \subseteq \equiv$ and $R_{\text{semantic}} \subseteq \equiv$, the least such congruence is contained in $\equiv$. □

Lemma 3.4 (Monotone Refinement)**.** *During saturation, semantic identifiers evolve monotonically: for every merge of two e-classes of type* τ *with identifiers* $\delta_1, \delta_2 \in \mathbb{D}_\tau$*, the resulting canonical identifier* δ^* *satisfies* $\delta_1 \sqsubseteq_\tau \delta^*$ *and* $\delta_2 \sqsubseteq_\tau \delta^*$.

Proof. For non-composite types, the property follows directly from lemma 3.3: W_τ maintains the most precise canonical representative, so the merged class inherits a representative at least as precise as either input.

For composite types, it follows by structural induction. Suppose $\delta_1 = K(v_1, \ldots, v_n)$ and $\delta_2 = K(w_1, \ldots, w_n)$, where K is the domain constructor of a composite type κ. Then `smerge` does not merge δ_1 and δ_2 directly; instead, it generates the component obligations $\{(\pi_i(\delta_1), \pi_i(\delta_2))\}_{i=1}^n$. By induction, each recursive merge yields a refined representative u_i such that $v_i \sqsubseteq_{\tau_i} u_i$ and $w_i \sqsubseteq_{\tau_i} u_i$. Therefore the reconstructed composite identifier $\delta^* = K(u_1, \ldots, u_n)$ satisfies $\delta_1 \sqsubseteq_\kappa \delta^*$ and $\delta_2 \sqsubseteq_\kappa \delta^*$ by the definition of the pointwise order on composite values.

The same invariant is preserved during rebuild. If an e-node $n = f(\delta_1, \ldots, \delta_m)$ is reinterpreted after its children have been refined to $\delta_1', \ldots, \delta_m'$ with $\delta_i \sqsubseteq_{\tau_i} \delta_i'$ for all i, then assumption 3.1 gives $[\![f]\!](\delta_1, \ldots, \delta_m) \sqsubseteq_\tau [\![f]\!](\delta_1', \ldots, \delta_m')$. Hence rebuilding after child merges can only preserve or refine the parent's identifier. □

Theorem 3.2 (Termination)**.** *Rebuild terminates.*

Proof. Every reprocessing step corresponds to strict progress in a well-founded measure. A parent e-node is re-enqueued only when one of the following occurs:

- its e-class participates in a merge; or
- one of its child identifiers is replaced by a strictly more precise canonical identifier.

The first kind of event can occur only finitely many times, because each merge strictly decreases the number of e-classes. The second kind is also finite: by lemma 3.4, identifiers evolve monotonically, so the sequence of identifiers assigned to an e-class forms an ascending chain in $\sqsubseteq_\tau$; by lemma 3.2, this chain is finite, so each e-class can be strictly refined only finitely many times.

Every refinement is attached to an already existing e-class, and under lemma 3.2, each e-class can be refined only finitely many times.

Therefore rebuild must eventually reach a fixpoint at which no further merges occur in any union-find structure W_τ and no e-class identifier is further refined. □

4 Semantic E-Graphs for Hardware

In this section, we instantiate the semantic e-graph framework for hardware by defining concrete e-class types and semantic domains for Register-Transfer Level (RTL) designs. We then demonstrate how these semantic domains enable forms of hardware reasoning that are difficult to achieve with traditional e-graphs.

4.1 Hardware Basics

We formalize hardware designs at the Register-Transfer Level[4] (RTL) using a small set of primitive constructs: *wires*, *wire vectors*[5], *tuples*, and *cells*.

Wires. A *wire* represents a single-bit signal and is drawn from a set $\mathrm{W} = \{w_0, w_1, \ldots\}$.

Wire Vectors. A *wire vector* represents a multi-bit signal and is treated as an uninterpreted syntactic constructor at the structural level. Each vector has an associated *width* $n \in \mathbb{N}$ and is denoted $v \in \mathrm{Vec}_n$, corresponding to an ordered sequence of n wires.

Signals and Tuples. We define the set of *signals* as $S = \mathrm{W} \cup \bigcup_n \mathrm{Vec}_n$, capturing both wires and wire vectors. A tuple is an ordered collection of signals, $t = (x_1, x_2, \ldots, x_k)$ with $x_i \in S$, used to represent multi-port cell interfaces.

Cells. A *cell* encapsulates a primitive hardware operation and is written $c = (I_c, O_c, f)$, where I_c and O_c are tuples of input and output signals, and f is the cell's functional symbol. All hardware components—including single-bit and multi-bit logic gates, arithmetic units (e.g., adders), stateful elements (e.g., registers), structural operators such as concat or extract, and entire modules or black-box IP blocks—may be uniformly represented as cells.

4.2 RTL in Semantic E-Graph

We instantiate the general semantic e-graph framework for RTL hardware using three e-class types: Wire, WireVec, and Register. For simplicity, we omit tuples here; they are only a technical device for cells with multiple output ports and do not affect the core semantic mechanism.

Wire. A Wire e-class represents a single-bit signal. Its semantic domain is $\mathbb{D}_{\mathsf{Wire}} = \mathbb{C}_{\mathsf{Wire}} \cup \mathbb{S}_{\mathsf{Wire}}$, where $\mathbb{C}_{\mathsf{Wire}} = \{0, 1\}$ and $\mathbb{S}_{\mathsf{Wire}}$ consists of symbolic wire identifiers drawn from W. Thus a wire's semantic identifier is either a concrete Boolean value or a symbolic wire (in the implementation, an unsigned integer). There is no decomposer for Wire.

[4]In RTL, designs are represented very simply as networks of registers (state elements) and combinational logic.
[5]Wires and wire vectors are analogous to the bits and bitvectors referenced in sections 1 and 2; in this section, we adopt more hardware-centric terminology.

WireVec. A WireVec e-class represents a bit-vector. Its semantic domain is $\mathbb{D}_{\mathsf{WireVec}} = \mathbb{C}_{\mathsf{WireVec}} \cup \mathbb{S}_{\mathsf{WireVec}}$, where $\mathbb{C}_{\mathsf{WireVec}} = \bigcup_{n\geq 0} \mathsf{Vec}_n(\mathbb{D}_{\mathsf{Wire}})$ and $\mathbb{S}_{\mathsf{WireVec}}$ consists of symbolic vector identifiers drawn from $\bigcup_n \mathsf{Vec}_n$. Thus a vector semantic identifier is either a concrete vector of wire semantic identifiers or a symbolic vector. Unlike Wire, WireVec is composite. For concrete identifiers $\delta_1 = [x_0, \ldots, x_{n-1}]$ and $\delta_2 = [y_0, \ldots, y_{m-1}]$, decomposition proceeds via the projections $\pi_i([x_0, \ldots, x_{n-1}]) = x_i$, generating component obligations $\{(x_i, y_i)\}_{i=0}^{n-1}$, where $n = m$ is required since merging only occurs between same-width vectors. Equality of two concrete vectors is thus reduced to equality of aligned lanes.

Register. A Register e-class represents a stateful element (e.g., a D flip-flop). It is composite with domain constructor $\mathsf{reg} : \mathbb{D}_{\mathsf{Wire}} \to \mathbb{D}_{\mathsf{Register}}$ and projection $\pi_1(\mathsf{reg}(w)) = w$, so merging two register identifiers propagates equality to their stored values. Registers introduce sequential feedback, which is handled specially during extraction; see section 5.

Semantic Constructors. Each RTL cell f is equipped with a semantic constructor $[\![f]\!] : \mathbb{D}_{\tau_1} \times \cdots \times \mathbb{D}_{\tau_m} \to \mathbb{D}_{\tau}$ for its result type τ. The constructor may return a concrete value, a symbolic identifier, or a partially-concrete composite value depending on how much information is available from the inputs.

For Wire, a gate can often evaluate even when not all inputs are concrete. For AND, the semantic constructor can short-circuit: $[\![\mathsf{AND}]\!](0, x) = 0$ and $[\![\mathsf{AND}]\!](1, 1) = 1$, while $[\![\mathsf{AND}]\!](1, x)$ remains symbolic for an unknown wire x. Thus $\mathsf{AND}(0, w)$ already evaluates to the concrete result 0, whereas $\mathsf{AND}(1, w)$ remains symbolic until the second input is refined further. This also illustrates monotonicity: refining an argument never invalidates a previously computed result. So if an AND node has enough semantic information to determine a concrete value, sadd reuses or creates the corresponding concrete wire identifier; otherwise it keeps the result symbolic.

For WireVec, word-level cells may interpret concrete vectors directly. For example, if $\delta_a = [a_0, \ldots, a_{n-1}]$ and $\delta_b = [b_0, \ldots, b_{m-1}]$ are concrete vectors, then $[\![\mathsf{ADD}]\!](\delta_a, \delta_b)$ evaluates to a concrete output vector of the declared output width. Operationally, this means that sadd on a word-level adder may immediately reuse or create a concrete WireVec identifier when both operands are concrete.

This choice makes the semantic type of each hardware signal explicit: Wire is a primitive semantic type, WireVec and Register are composite. In practice, RTLIL designs are translated into this representation by normalizing word-level arithmetic and sequential elements into a consistent form; see section 5 for details.

4.3 Better Hardware Reasoning with Semantic E-Graphs

Semantic e-graphs enable forms of hardware reasoning that are difficult to achieve with traditional e-graphs. In hardware, the same operator or value often admits multiple natural representations, and exploiting these relationships is essential for effective optimization. Semantic e-graphs make these relationships explicit through semantic constructors, allowing them to unify equivalent structures across levels of abstraction and to discover additional congruences without requiring bespoke rewrite rules.

The following two examples illustrate these distinct advantages. The first shows how the semantic e-graph simultaneously maintains word-level and bit-level views of an operator, enabling cross-level optimization. The second shows how the semantic e-graph can automatically recover structural equivalences, such as extract–concat roundtrips, that traditional e-graphs would need carefully crafted rewrite rules to express.

Example 4.1 (Two–Bit Adder Synthesis). We illustrate how the semantic e-graph aids hardware reasoning using a simple 2-bit adder synthesis example, shown in fig. 3 (left). Given input vectors $a \triangleright [a_0, a_1]$ and $b \triangleright [b_0, b_1]$, the adder cell $\mathsf{adder}(a \triangleright [a_0, a_1],\ b \triangleright [b_0, b_1]) \triangleright [s_0, s_1]$ can be rewritten into its bit-level implementation by a synthesis rewrite rule:

$$\mathsf{adder}(a, b) \longrightarrow \big[\mathsf{xor}(a_0, b_0),\ \mathsf{xor}(\mathsf{and}(a_0, b_0),\ \mathsf{xor}(a_1, b_1))\big].$$

Such rules can be automatically generated by a generic adder implementation. Under this rewrite, the semantic e-graph merges $s_0 \equiv \mathsf{xor}(a_0, b_0)$ and $s_1 \equiv \mathsf{xor}(\mathsf{and}(a_0, b_0),\ \mathsf{xor}(a_1, b_1))$, thus, each output bit of the high-level adder is placed in the same e-class as the corresponding bit of its synthesized implementation.

Because both representations are present in the semantic e-graph, the adder is now available simultaneously at word level and at bit level. This enables bit-level optimizations that exploit shared substructure. For example, consider a three-bit adder $\mathsf{adder}(a' \triangleright [a_0, a_1, a_2],\ b' \triangleright [b_0, b_1, b_2]) \triangleright [s'_0, s'_1, s'_2]$. Its least-significant bit also rewrites to $\mathsf{xor}(a_0, b_0)$. Since s_0 and s'_0 both merge with the same $\mathsf{xor}(a_0, b_0)$ e-node, the semantic e-graph unifies them automatically. Thus common subexpressions, such as low-order sum bits across adders of different widths, are not duplicated but instead shared through the semantic structure of the e-graph.

This example shows how the semantic e-graph fully exploits multi-level hardware representations, enabling cross-level optimizations that a single-level e-graph cannot achieve.

Example 4.2 (Extract and Concat). Here, we demonstrate the example originally presented in section 2. This example is shown visually in fig. 3 (right). Consider a wire vector in e-class c with $c \triangleright \delta = [a_0, a_1, a_2, a_3]$. We create two extract e-nodes $\mathsf{extr}(v, 0, 1), \mathsf{extr}(v, 2, 3)$, where the extract bounds $(0, 1)$ and $(2, 3)$ are themselves e-nodes in the semantic e-graph but are shown directly in the operator for readability.

By the semantic constructor of extr,

$$c_0 \triangleright \delta_0 = [\![\mathsf{extr}]\!](v, 0, 1) = [a_0, a_1], \qquad c_1 \triangleright \delta_1 = [\![\mathsf{extr}]\!](v, 2, 3) = [a_2, a_3].$$

Thus each extract extracts the corresponding contiguous subvector of v.

Next, consider the concat e-node $\mathsf{concat}(e_0, e_1)$. Its semantic constructor yields

$$c_2 \triangleright \delta_2 = [\![\mathsf{concat}]\!](\delta_0, \delta_1) = [a_0, a_1] \parallel [a_2, a_3] = [a_0, a_1, a_2, a_3].$$

Hence $\delta_2 = \delta$, and the semantic e-graph merges c_2 with the original e-class c. Without any explicit rewrite rule, the semantic e-graph discovers that slicing a vector and concatenating the extracts reconstructs the original value.

This example illustrates how semantic identifiers enable the semantic e-graph to recover such congruences automatically, capabilities that would otherwise require intricate hand-written rewrite rules in traditional e-graphs.

These examples demonstrate how semantic e-graphs enable reasoning across multiple abstraction levels and automatically recover equivalences that would otherwise require extensive rewrite rules.

5 Nextmap

We now present Nextmap, a technology-mapping and optimization framework that leverages semantic e-graphs to unify synthesis, optimization, and technology mapping.

Conventional mappers operate at a single abstraction level, typically at the bit level or the word level, thereby missing optimization opportunities that span across these layers. Nextmap encodes both the syntactic structure and semantic meaning of hardware components, from individual wires to composite arithmetic cells, within a single SEG, enabling cross-level optimization. This section

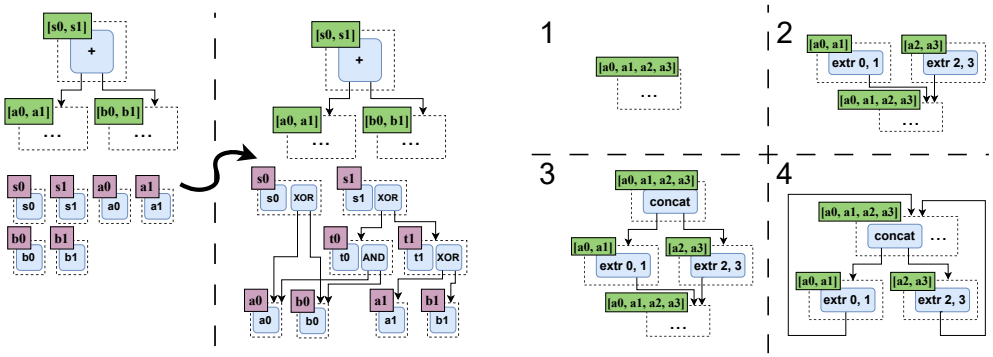


Fig. 3. Left: Synthesizing an adder. Right: Semantics-based automated unification of nested extracts and concat.

describes Nextmap's workflow, rewrite strategy, technology mapping, extraction, and database backend.

5.1 High-Level Workflow

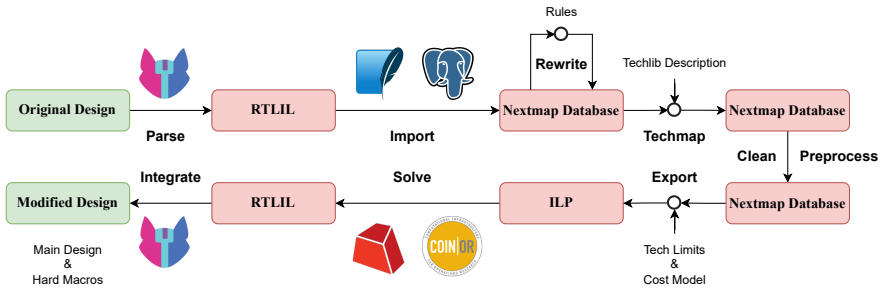


Fig. 4. Overview of Nextmap's Workflow.

As illustrated in fig. 4, Nextmap accepts a design in Yosys's RTLIL format and translates it into our database-backed SEG representation.

After import, users may apply rewrite rules—either the built-in set or user-defined extensions—to saturate the SEG with technology-independent transformations.

Following these rewrites, users provide a technology library (techlib). The library may be specified in JSON for simple cells or in Python for complex or parameterized cells. Based on this specification, Nextmap performs technology mapping via rewrite rules. Users may optionally supply technology limits and cost models that capture extraction constraints and optimization objectives.

Nextmap then formulates the extraction task as an ILP instance and delegates the optimization to an external solver, such as Gurobi or the open-source CBC solver. After receiving the solver's result, Nextmap interprets the solution and generates the corresponding RTLIL. The final RTLIL includes both the technology-mapped portion—expressed as black-box instances of the target hard macros—and any remaining unmapped abstract cells, which can be handled by downstream synthesis flows.

5.2 Technology-Independent Rewrites

Nextmap applies a collection of technology-independent rewrites to normalize and optimize hardware designs before technology mapping. These rewrites fall into several built-in rulesets, each targeting a specific class of structural or arithmetic transformations.

Table 1. A selection of representative technology-independent rewrite rules in Nextmap.

Ruleset	Representative Rule	Description
Arithmetic	Commutative	`(Add ?a ?b)` → `(Add ?b ?a)`
	Associative	`(Add (Add ?a ?b) ?c)` → `(Add ?a (Add ?b ?c))` (when safe)
	Width-Reduction	Removes redundant leading zeros in unsigned arithmetic.
	Karatsuba	Splits a wide multiply into three narrower multiplies plus adders/subtractors.
	Partial-Product-Split	Splits a 16×32 multiply into two smaller multiplies plus adders.
Retiming	Forward	Pushes a register forward across combinational logic.
	Backward	Pulls a register backward across combinational logic.
Bitblast	Adder	Expands an adder into a bitwise ripple-carry chain.
	Comparator	Expands `<, <=, ==` into bitwise compare chains.
	Word-Mux	Expands word-level mux into per-bit selects.
Logic	Double-Negation	`(Not (Not ?x))` → `?x`
	Idempotence	`(And ?x ?x)` → `?x`
	Absorption	`(And ?x (Or ?x ?y))` → `?x`
	De Morgan	`(Not (And ?x ?y))` → `(Or (Not ?x) (Not ?y))`

Table 1 lists several built-in rulesets along with representative rules from each. In addition to traditional arithmetic rules such as commutativity and associativity, which operate primarily at the word level, Nextmap includes mixed word–bit-level rules. For example, width-reduction shrinks operand bitwidths when high bits are constant, and Karatsuba decomposes a wide multiplication into three narrower multiplications.

To improve scalability, Nextmap employs several heuristics: Rewrites are applied in stages to limit the working set size at each step. Scheduling rewrites is a common strategy well supported in tools like `egglog`. Certain rewrites, such as retiming, are applied only to a register and its surrounding arithmetic logic rather than all gates, reducing the search space and preventing e-graph explosion.

5.3 Technology Mapping

After saturation, Nextmap performs techmap by pattern matching against the input techlib.

DSP Mapping. For DSP blocks, Nextmap uses multiplication as the anchor for matching, since it is the computational core around which DSP datapaths are organized. Starting from each candidate multiplier, the mapper incrementally explores the connected pre- and post-processing logic to determine whether the surrounding structure can be absorbed into a DSP instance, rather than relying on a large enumerated rule set. This basis-driven recursive exploration generalizes across DSP blocks whose internal datapaths are multiplier-centered; details and a DSP48E1-based case study are provided in the appendix.

ML Accelerator Blocks. For regular structures, such as MAC-based processing elements in machine learning accelerators, Nextmap performs hierarchical matching. Individual elements are identified first, then combined into larger arrays (e.g., 2×2, 4×4 meshes). This recursive, reduction-based strategy efficiently captures both regular and slightly irregular layouts without excessive computation.

5.4 Extraction

Extraction selects a single, type-consistent hardware implementation from the saturated SEG using an integer linear program (ILP); see the appendix for details. Unlike traditional e-graphs, SEG distinguishes between *combinational* and *sequential* loops, ensuring only the latter can be preserved. In the ILP formula, Sequential elements are modeled explicitly to permit valid feedback through registers, while combinational cycles are disallowed by bounded propagation delay variables (critical path length). This unified formulation yields precise, technology-aware extractions directly from the SEG.

5.5 Database Backend

While `egglog` offers a logic-programming interface over an e-graph database, SEG is implemented atop SQLite for simplicity and extensibility, not for performance. SQLite serves as a lightweight yet efficient relational backend with native support for declarative queries, aggregation, and grouping—capabilities essential for implementing SEG's grouped rewrites and rebuild procedures. Moreover, explicit control over indexing enables efficient container manipulation, and the standard SQL interface facilitates seamless integration with external tools. While we choose SQLite for our main backend, as we demonstrate in section 6.4, similar functionality could also be implemented in `egglog`, though with somewhat greater effort. We also have preliminary support for PostgreSQL as the backend, which is faster in some cases.

6 Evaluation

Here, we evaluate Nextmap and the semantic e-graph, addressing the following research questions:

- RQ1 How does Nextmap's quality of results (QoR) compare to existing tools?
- RQ2 Can we attribute Nextmap's QoR improvements to the simultaneous optimization and mapping enabled by the semantic e-graph?
- RQ3 Does the semantic e-graph enable us to build a representation general enough to implement a broad set of EDA tasks?
- RQ4 Does the semantic e-graph enable a more efficient representation for EDA tasks?

Our core motivation is to demonstrate the true promise of eqsat for EDA, by demonstrating that simultaneous optimization and mapping leads to better QoR. RQ1 and RQ2 evaluate whether we achieve this goal. First, RQ1 asks how Nextmap's QoR compares to existing EDA tools. We answer this in section 6.1 via a straightforward comparison of Nextmap against two existing EDA tools on a wide range of benchmarks. Finding that Nextmap achieves higher QoR, RQ2 asks, *why?* Is it truly due to simultaneous optimization and mapping? Section 6.2's ablation study shows that, when we artificially disable simultaneous optimization and mapping in Nextmap, its QoR declines steeply.

Underlying Nextmap is the core conceptual contribution of this paper: the semantic e-graph. We developed the semantic e-graph motivated by the need for an *efficient* and *expressive* eqsat-friendly representation for hardware; RQ3 and RQ4 ask whether we satisfy this need. RQ3 asks whether Nextmap's representation achieves an adequate level of *expressivity*. In section 6.3 we evaluate RQ3 by demonstrating that Nextmap is expressive enough to cover a range of EDA tasks covered by previous eqsat-for-hardware tools. RQ4, on the other hand, asks whether Nextmap's SEG-enabled representation is more efficient than other possible eqsat-friendly representations. In section 6.4 we conduct various experiments demonstrating the efficiency of the SEG-based representation.

All experiments were executed on a server running Ubuntu 22.04.5 LTS (Jammy) with Linux kernel 5.15.0-153-generic. The system features two Intel® Xeon® E5-2420 processors (24 logical cores) and 31 GiB of RAM. The software environment is based on Python 3.10.12. Unless otherwise stated, our default hardware target is the Xilinx UltraScale+ family of FPGAs.

Table 2. Resource comparison across three tools (Nextmap, Yosys, Proprietary) on small but challenging hardware design benchmarks. A tool's columns are shaded for a row when no other tool uses fewer resources on any reported metric. Multiple groups may be shaded when tools tie. CARRY4 is a specialized logic element which implements a carry-chain, useful for adders and comparison operations. FF is short for flip-flop (register). Nextmap is quick (<5sec) in each case, and so tool timing is not included.

Name	Description	Nextmap				Yosys				Proprietary			
		DSP	CARRY4	FF	LUT	DSP	CARRY4	FF	LUT	DSP	CARRY4	FF	LUT
bad_multiplier	16-bit truncated multiplier	1	0	0	0	1 (0%)	0 (0%)	64 (+∞)	0 (0%)	1 (0%)	0 (0%)	0 (0%)	0 (0%)
complex_multiplier	$(a + bi) \times (c + di)$	3	0	64	0	4 (+33%)	16 (+∞)	128 (+100%)	64 (+∞)	4 (+33%)	0 (0%)	64 (0%)	0 (0%)
dot_product	$a \times b + c \times d$	2	0	32	0	2 (0%)	8 (+∞)	64 (+100%)	32 (+∞)	2 (0%)	0 (0%)	32 (0%)	0 (0%)
multiplier_with_rst	16-bit multiplier w/ sync reset	1	0	0	0	1 (0%)	0 (0%)	64 (+∞)	0 (0%)	1 (0%)	0 (0%)	0 (0%)	0 (0%)
redundant_adders	Three adders w/ shared inputs	0	8	0	32	0 (0%)	14 (+75%)	0 (0%)	32 (0%)	0 (0%)	14 (+75%)	0 (0%)	57 (+78%)
signed_mac	$a \times b + c$	1	0	32	0	1 (0%)	8 (+∞)	64 (+100%)	32 (+∞)	1 (0%)	0 (0%)	32 (0%)	0 (0%)
signed_reg	Sign-extension and delay	0	0	32	0	0 (0%)	0 (0%)	32 (0%)	0 (0%)	0 (0%)	0 (0%)	32 (0%)	0 (0%)
square_diff	$(a - b)^2$	1	0	64	0	1 (0%)	5 (+∞)	83 (+30%)	16 (+∞)	1 (0%)	5 (+∞)	32 (-50%)	16 (+∞)
unsigned_mac	$a \times b + c$	1	0	32	0	1 (0%)	8 (+∞)	64 (+100%)	32 (+∞)	1 (0%)	0 (0%)	32 (0%)	0 (0%)
wide_multiplier	16–32-bit multiplier	2	0	17	0	2 (0%)	4 (+∞)	64 (+276%)	15 (+∞)	2 (0%)	0 (0%)	17 (0%)	0 (0%)

6.1 Nextmap's Quality of Results vs. Existing Tools

To answer RQ1, we compare Nextmap against existing EDA tools: the open-source hardware synthesis tool Yosys [32] and a proprietary hardware synthesis tool. We ran all three tools on two sets of benchmarks: one set of small but challenging designs, and one set of large, real-world designs. The small benchmarks were primarily adapted from Xilinx's user guide examples, and capture edge cases for which proprietary tools are optimized, but that remain difficult for open-source flows. The large benchmarks contain fewer edge cases, but are more realistic in scale. For the large benchmarks, SA(N,W) denotes a systolic array of size $N \times N$ with W-bit weights used for matrix multiplication; FIR(T,W) denotes a finite impulse response filter with T taps and W-bit coefficients; FFT(P,W) denotes a pipelined radix-2^2 single-path delay feedback FFT of P points with W-bit data. All three families use fixed-point arithmetic. `nerv` is a simple RISC-V CPU design from YosysHQ. The benchmarks range from 1,082 wires (FIR(16,8)) to 345,238 wires (FFT(1024,32)) before optimization.

For these experiments, all three tools used the same limit for the number of DSPs available. The cost model employed by Nextmap was intentionally simple: the cost of logic cells, adders, and registers scales linearly with bit-width, while the cost of multipliers scales quadratically. We equipped Nextmap with a rich rewrite rule set consisting of arithmetic, logic, retiming, and DSP techmapping. As described in section 5, Nextmap only explicitly technology maps to DSPs. The rest of the design is optimized by our arithmetic, logic, and retiming rewrites, but LUT/MUX/flip-flop mapping is left to Yosys.

Results are summarized in table 2 for the small designs and table 3 for the large designs. Nextmap largely dominates Yosys, i.e., uses equal or fewer resources in all categories, while using fewer resources in at least one category. Although Yosys provides basic passes for mapping multipliers to DSP slices and packing registers into DSPs, it lacks two critical features: (1) retiming transformations that can move registers across combinational logic *in conjunction with DSP techmap*, and (2) comprehensive techmap rules for DSP slices. As a result, Yosys often consumes more logic cells and registers than either Nextmap or the commercial tool.

Nextmap is also competitive relative to the commercial tool, often tying its results and occasionally dominating its results. Commercial tools, while equipped with rich vendor-specific techmap rules, are comparatively rigid and often limited to a fixed set of recognizable patterns [25]. They are less effective at identifying behaviorally equivalent but syntactically different forms of computation. These results demonstrate that, despite millions of dollars and many engineer-hours of investment, there are still advancements to be made over commercial tools.

Table 3. Resource comparison on larger benchmarks, in the same format as table 2.

Name	Nextmap					Yosys					Proprietary				
	DSP	CARRY4	FF	LUTx	MUXFx	DSP	CARRY4	FF	LUTx	MUXFx	DSP	CARRY4	FF	LUTx	MUXFx
SA(4,8)	16	0	128	0	0	16 (0%)	64 (+∞)	448 (+250%)	400 (+∞)	0 (0%)	0 (-100%)	256 (+∞)	448 (+250%)	1648 (+∞)	0 (0%)
SA(4,16)	16	0	256	0	0	16 (0%)	128 (+∞)	896 (+250%)	864 (+∞)	0 (0%)	16 (0%)	0 (0%)	16 (-94%)	0 (0%)	0 (0%)
SA(4,32)	48	594	1568	3568	288	64 (+33%)	720 (+21%)	1792 (+14%)	3848 (+8%)	0 (-100%)	64 (+33%)	448 (-25%)	1280 (-18%)	1776 (-50%)	0 (-100%)
SA(8,32)	192	2426	7200	14432	1152	256 (+33%)	2880 (+19%)	7680 (+7%)	15232 (+6%)	0 (-100%)	240 (+25%)	2904 (+20%)	6656 (-8%)	13456 (-7%)	0 (-100%)
SA(16,32)	768	9629	30752	57488	4608	1024 (+33%)	11520 (+20%)	31744 (+3%)	60928 (+6%)	0 (-100%)	840 (+9%)	19956 (+107%)	29793 (-3%)	101464 (+76%)	0 (-100%)
FIR(16,8)	16	16	105	285	88	16 (0%)	4 (-75%)	240 (+129%)	1011 (+255%)	799 (+808%)	0 (-100%)	255 (+1494%)	240 (+129%)	1537 (+439%)	0 (-100%)
FIR(16,16)	16	32	225	620	184	16 (0%)	8 (-75%)	496 (+120%)	2154 (+247%)	1713 (+831%)	16 (0%)	0 (-100%)	210 (-7%)	0 (-100%)	0 (-100%)
FIR(16,32)	48	384	465	8009	4796	64 (+33%)	464 (+21%)	1008 (+117%)	6176 (-23%)	3576 (-25%)	64 (+33%)	304 (-21%)	434 (-7%)	1697 (-79%)	0 (-100%)
FIR(32,32)	96	752	961	16045	9563	128 (+33%)	912 (+21%)	2016 (+110%)	14024 (-13%)	8880 (-7%)	128 (+33%)	624 (-17%)	930 (-3%)	3457 (-78%)	0 (-100%)
FIR(64,32)	192	1488	1953	35485	23085	256 (+33%)	1808 (+22%)	4032 (+106%)	29801 (-16%)	19641 (-15%)	256 (+33%)	1264 (-15%)	1922 (-2%)	6977 (-80%)	0 (-100%)
FFT(64,16)	6	153	425	958	0	8 (+33%)	185 (+21%)	478 (+12%)	1084 (+13%)	0 (0%)	8 (+33%)	180 (+18%)	553 (+30%)	1165 (+22%)	0 (0%)
FFT(128,16)	9	174	503	1204	35	12 (+33%)	222 (+28%)	571 (+14%)	1395 (+16%)	37 (+6%)	12 (+33%)	200 (+15%)	663 (+32%)	1467 (+22%)	29 (-17%)
FFT(1024,32)	37	982	1482	10410	1227	65 (+76%)	1042 (+6%)	1482 (0%)	10343 (-1%)	2838 (+131%)	64 (+73%)	728 (-26%)	1921 (+30%)	8765 (-16%)	806 (-34%)
nerv	0	895	4183	15132	6919	0 (0%)	904 (+1%)	4163 (> −1%)	14028 (-7%)	7229 (+4%)	0 (0%)	620 (-31%)	2290 (-45%)	8255 (-45%)	63 (-99%)

Table 4. Ablation study: Nextmap vs. phase-ordered Nextmap, in the same format as table 2.

Name	Nextmap					phase-ordered Nextmap				
	DSP	CARRY4	FF	LUTx	MUXFx	DSP	CARRY4	FF	LUTx	MUXFx
SA(4,8)	16	0	128	0	0	16 (0%)	64 (+∞)	320 (+150%)	304 (+∞)	0 (0%)
SA(4,16)	16	0	256	0	0	16 (0%)	128 (+∞)	640 (+150%)	640 (+∞)	0 (0%)
SA(4,32)	48	594	1568	3568	288	48 (0%)	634 (+7%)	1346 (-14%)	4240 (+19%)	288 (0%)
SA(8,32)	192	2426	7200	14432	1152	192 (0%)	2531 (+4%)	5999 (-17%)	16944 (+17%)	1152 (0%)
SA(16,32)	768	9629	30752	57488	4608	768 (0%)	10134 (+5%)	18126 (-41%)	67808 (+18%)	4608 (0%)
FIR(16,8)	16	16	105	285	88	16 (0%)	32 (+100%)	225 (+114%)	1002 (+252%)	775 (+781%)
FIR(16,16)	16	32	225	620	184	16 (0%)	8 (-75%)	465 (+107%)	2165 (+249%)	1709 (+829%)
FIR(16,32)	48	384	465	8009	4796	64 (+33%)	465 (+21%)	1009 (+117%)	6218 (-22%)	3587 (-25%)
FIR(32,32)	96	752	961	16045	9563	128 (+33%)	913 (+21%)	2017 (+110%)	14077 (-12%)	8903 (-7%)
FIR(64,32)	192	1488	1953	35485	23085	256 (+33%)	1809 (+22%)	4033 (+107%)	29805 (-16%)	19598 (-15%)
FFT(64,16)	6	153	425	958	0	6 (0%)	209 (+37%)	478 (+12%)	1143 (+19%)	0 (0%)
FFT(128,16)	9	174	503	1204	35	9 (0%)	258 (+48%)	571 (+14%)	1479 (+23%)	34 (-3%)
FFT(1024,32)	37	982	1482	10410	1227	49 (+32%)	1022 (+4%)	1482 (0%)	10428 (+<1%)	1638 (+33%)

Nextmap's runtimes on the larger benchmarks range from 0.3 s to 1938 s. Nextmap runtime includes the time spent techmapping with Yosys. Most complete in under a minute: SA(4,8): 1.4 s; SA(4,16): 1.2 s; SA(4,32): 5.9 s; FIR(16,8): 0.3 s; FIR(16,16): 0.4 s; FIR(16,32): 5.7 s; FIR(32,32): 14.3 s; FIR(64,32): 41.2 s; FFT(64,16): 1.5 s; FFT(128,16): 2.4 s; FFT(1024,32): 33.0 s; `nerv`: 8.2 s. Only the two largest systolic arrays take substantially longer: SA(8,32) at 145 s and SA(16,32) at 1938 s. In all cases, the vast majority of time is spent in the ILP solver. Efficient extraction from e-graphs is a well-known issue [3, 12, 13, 34], and though Nextmap contains many optimizations in its ILP formulation (see the appendix), there is still room for improvement in many cases. However, in the vast majority of cases, Nextmap's runtime is more than adequate.

The user-provided technology limits subtly influence the final mapping. In a case study (see the appendix), we observed that Nextmap explores a wider range of valid implementations under these limits than the proprietary tool, yielding better results on most resource-constrained targets.

6.2 Nextmap's Quality of Results vs. "Phase-Ordered" Nextmap

To answer RQ2 and more effectively demonstrate the benefit of simultaneous optimization, we conducted an ablation study in which we compared Nextmap with a "phase-ordered" version of Nextmap. Both tools are equipped with the same rewrite rules: synthesis, optimization and techmapping. However, "phase-ordered" Nextmap separates out the rewrites by phase and runs each phase separately, extracting a single representative from the e-graph after each stage, and repopulating a fresh e-graph using just that representative in the next stage.

The results are shown in table 4. None of the results from the "phase-ordered" Nextmap beat plain Nextmap, while plain Nextmap dominates "phase-ordered" Nextmap in a number of cases. This clearly demonstrates that *simultaneous* optimization and mapping is key to Nextmap's quality of results. Though we provide reasonable cost models for every intermediate extraction in phase-ordered Nextmap, they cannot perfectly reflect the optimal extraction choice at each point. Thus, some useful information will always be lost in extraction when moving between phases.

Table 5. Nextmap vs. E-Syn on logic synthesis over the EPFL Combinational Benchmarks.

Name	Wires	Nextmap			E-Syn		
		Area (μm^2)	Delay (ps)	Time (s)	Area (μm^2)	Delay (ps)	Time (s)
dec	671	351.55	69.63	336	351.55	69.63	399
cavlc	2303	484.52	107.09	41	473.79	114.69	1389
adder	6024	1144.94	1964.39	15	1102.25	2088.60	683
max	14542	2483.50	1854.88	408	2534.35	1775.79	35459

6.3 Expressivity of Nextmap

To answer RQ3, this experiment examines whether Nextmap's representation, enabled by the semantic e-graph, is general enough to implement a range of EDA tasks. To do so, we demonstrate that it is possible to reimplement the eqsat functionality of a variety of recent eqsat-for-hardware works (E-Syn [5] for bit-level optimization, ROVER [9] for arithmetic optimization, and Churchroad [24] for DSP techmapping), plus a new task: mapping to hardware blocks specialized for machine learning. In all cases, we successfully reimplement the eqsat contributions of prior work using *only* the core Nextmap infrastructure described in this paper.

Bit-Level Optimization. We benchmark Nextmap against E-Syn [5] for AIG-based synthesis using equivalent rewrites and ILP-based extraction. Table 5 shows four representative examples of varying size. Nextmap achieves comparable area and performance results to E-Syn, with consistently shorter runtime, demonstrating native support for bit-level optimization. Note that the hardware target in these experiments is the ASAP7 ASIC PDK [7].

Arithmetic Optimization. ROVER [9] focuses on RTL-level arithmetic and datapath optimizations. As ROVER itself is closed-source, we cannot directly benchmark against its implementation. However, based on the published descriptions, we implemented its core bitvector arithmetic and logic rewrites in our system. For ROVER's open-source benchmarks, our evaluation includes equivalent or larger designs. We did not implement specialized rule sets such as constant expansion, and thus do not include MCM (multiple constant multiplication) benchmarks in our evaluation.

DSP Mapping. Churchroad operates at the RTL level, focusing on arithmetic-to-DSP mappings with hardcoded rules in `egglog`, producing feasible but not optimized designs. Using equivalent DSP techmap rules, Nextmap matches the mapped outputs of every test case in the Churchroad repository, and additionally enables algebraic and structural optimizations on other designs.

Large Accelerator Mapping. We evaluated Nextmap on structured, repetitive computations representative of the systolic-array-based machine learning processors in Achronix and other FPGAs. Using the recursive pattern-matching approach described in section 5.3, Nextmap efficiently recognized and mapped large systolic arrays onto various target architectures. For example, a 16×16 matrix-multiplication array was successfully decomposed and mapped onto 16 4×4 MAC meshes in about 40 seconds, maintaining structural consistency across all processing elements. This demonstrates the true range of Nextmap. Not only can it implement the lowest, bit-level optimization; it can also capture mapping tasks larger than what most EDA tools generally handle.

6.4 Efficiency of the Semantic E-Graph

To answer RQ4 about the efficiency of semantic e-graphs, we implement two simplified versions of Nextmap directly within `egglog`, one with semantic e-graphs and one without (`egglog-vanilla` and `egglog-spec`, respectively). Both versions implement the same set of optimization and mapping rewrites. `egglog-vanilla` is the naive implementation described in section 2 which falls victim to the $O(n^2)$ `egglog` inefficiencies. `egglog-spec` (specialized) implements the semantic e-graph's

Table 6. egglog-vanilla vs. egglog-spec.

Name	Description	Wire Count	Cell Count	egglog-vanilla		egglog-spec		Nextmap	
				Build	Rewrite	Build	Rewrite	Build	Rewrite
alu_w32	32-bit ALU	1079	69	3.77 s	0.03 s	0.04 s	0.03 s	0.04 s	0.03 s
multi_alu_n4_w32	Four 32-bit ALUs	4732	276	34.78 s	0.12 s	0.15 s	0.04 s	0.10 s	0.03 s
multi_alu_n8_w32	Eight 32-bit ALUs	9464	552	199.95 s	0.56 s	0.24 s	0.04 s	0.20 s	0.03 s
multi_alu_n16_w32	Sixteen 32-bit ALUs	18928	1104	>600 s (timeout)	N.A.	0.63 s	0.07 s	0.51 s	0.07 s
aes_sbox	256-bit AES S-box	11768	2548	>600 s (timeout)	N.A.	0.82 s	0.01 s	0.73 s	0.01 s

efficient upward and downward propagation of equalities using custom egglog primitives. Given the same ruleset and design, we measure how long it takes each tool to saturate the e-graph. The results are in table 6, with Nextmap included for reference. egglog-spec clearly dominates egglog-vanilla. As the primary difference between these two implementations is the use of the semantic e-graph, this data clearly demonstrates how the semantic e-graph contributes to the efficiency of our representation.

7 Related Work

Prior eqsat-for-EDA tools each target a single stage: E-Syn [5] and E-Morphic [4] apply e-graphs to logic synthesis, ROVER [9], SEER [6], and ESFO [20] optimize word-level datapaths at RTL, HLS, and FIRRTL levels, and EqMap [14] and Churchroad [24, 26] perform technology mapping for LUTs and DSPs. Each achieves gains within its stage but does not cross stage boundaries. ESFO [20] comes closest, partially bridging word- and bit-level optimization within FIRRTL, but does not unify optimization with mapping. Nextmap demonstrates true simultaneous optimization and mapping.

Similar ideas have been explored in the SMT and automated reasoning literature. CC(X) [8] combines congruence closure with a solvable theory X, using semantic values as class representatives rather than canonized terms. Other work [2] centralizes equality reasoning in a shared equality-graph structure across theory reasoners. Semantic e-graphs are closest in spirit to CC(X). Both semantic e-graphs and CC(X) enrich equality maintenance with domain-specific semantic information, but semantic e-graphs operate in equality saturation rather than SMT.

Semantic e-graphs also relate to normalization by evaluation (NBE) [1, 10], where evaluation produces canonical representatives for equality checking. More broadly, partial evaluation [11, 15] specializes programs by evaluating as much as possible statically; semantic constructors in SEGs perform analogous evaluation during construction.

Several works extend traditional e-graphs for efficiency or expressiveness. Colored e-graphs [23] support multiple alternative assumptions in a single e-graph. Dis/Equality graphs [35] efficiently handle disequalities. Slotted e-graphs [22] and related work [28] add first-class support for bindings. Guided eqsat [16] uses program sketches to scale e-graph rewriting to complex transformations.

8 Conclusion

In this paper, we present Nextmap: a new eqsat-based framework for EDA tasks which pushes beyond the limitations of previous eqsat-for-EDA tools by implementing truly simultaneous optimization and mapping. This is made possible by our core insight: *semantic e-graphs,* an enhancement to traditional e-graphs which more efficiently implement semantics-based equality detection. This enables us to avoid the problems of e-graph blowup suffered by purely syntax-driven equality detection which have especially plagued previous eqsat-powered EDA tools. Via simultaneous optimization and mapping, Nextmap reaches previously unreachable optimizations and demonstrates competitive results when compared to open-source and commercial tools.

Acknowledgments

AI tools were used to proofread, edit, and refine the wording of this paper. AI programming tools were also used during development. The authors thank Yihong Zhang for his assistance with eqsat details, Sam Coward for his guidance on the eqsat-for-EDA landscape, Susan Clay for her paper feedback, engineers at Achronix for their feedback and discussions, and our anonymous reviewers and shepherd for their helpful feedback and suggestions. This material is based upon work supported by the National Science Foundation under award nos. 2237379 and 2450426.

Data Availability Statement

The software artifact supporting this work is freely available on Zenodo [17]. It contains the Nextmap artifact used for this paper, including the implementation and supporting materials needed to reproduce the results in our evaluation.

References

[1] Ulrich Berger and Helmut Schwichtenberg. 1991. An inverse of the evaluation functional for typed lambda-calculus. (1991). doi:10.1109/lics.1991.151645

[2] François Bobot, Stéphane Graham-Lengrand, Bruno Marre, and Guillaume Bury. 2018. Centralizing equality reasoning in MCSAT. In *16th International Workshop on Satisfiability Modulo Theories (SMT 2018)*.

[3] Yaohui Cai, Kaixin Yang, Chenhui Deng, Cunxi Yu, and Zhiru Zhang. 2025. Smoothe: Differentiable e-graph extraction. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*. 1020–1034. doi:10.1145/3669940.3707262

[4] Chen Chen, Guangyu Hu, Cunxi Yu, Yuzhe Ma, and Hongce Zhang. 2025. E-morphic: Scalable Equality Saturation for Structural Exploration in Logic Synthesis. *arXiv preprint arXiv:2504.11574* (2025). doi:10.1109/dac63849.2025.11133110

[5] Chen Chen, Guangyu Hu, Dongsheng Zuo, Cunxi Yu, Yuzhe Ma, and Hongce Zhang. 2024. E-syn: E-graph rewriting with technology-aware cost functions for logic synthesis. In *Proceedings of the 61st ACM/IEEE Design Automation Conference*. 1–6. doi:10.1145/3649329.3656246

[6] Jianyi Cheng, Samuel Coward, Lorenzo Chelini, Rafael Barbalho, and Theo Drane. 2024. Seer: Super-optimization explorer for high-level synthesis using e-graph rewriting. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 1029–1044. doi:10.1145/3620665.3640392

[7] Lawrence T Clark, Vinay Vashishtha, Lucian Shifren, Aditya Gujja, Saurabh Sinha, Brian Cline, Chandarasekaran Ramamurthy, and Greg Yeric. 2016. ASAP7: A 7-nm finFET predictive process design kit. *Microelectronics Journal* 53 (2016), 105–115. doi:10.1016/j.mejo.2016.04.006

[8] Sylvain Conchon, Evelyne Contejean, Johannes Kanig, and Stéphane Lescuyer. 2008. CC (X): Semantic combination of congruence closure with solvable theories. *Electronic Notes in Theoretical Computer Science* 198, 2 (2008), 51–69. doi:10.1016/j.entcs.2008.04.080

[9] Samuel Coward, Theo Drane, and George A Constantinides. 2024. ROVER: RTL optimization via verified E-graph rewriting. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 43, 12 (2024), 4687–4700. doi:10.1109/tcad.2024.3410154

[10] Peter Dybjer and Andrzej Filinski. 2000. Normalization and partial evaluation. In *International Summer School on Applied Semantics*. Springer, 137–192. doi:10.1007/3-540-45699-6_4

[11] Yoshihiko Futamura. 1971. Partial evaluation of computation process-an approach to a compilercompiler. *Systems, computers, controls* 2, 5 (1971), 45–50. doi:10.1023/a:1010043619517

[12] Amir Kafshdar Goharshady, Chun Kit Lam, and Lionel Parreaux. 2024. Fast and optimal extraction for sparse equality graphs. *Proceedings of the ACM on Programming Languages* 8, OOPSLA2 (2024), 2551–2577. doi:10.1145/3689801

[13] Mike He, Haichen Dong, Sharad Malik, and Aarti Gupta. 2023. Improving term extraction with acyclic constraints. In *EGRAPHS 2023 workshop*.

[14] Matthew Hofmann, Berk Gokmen, and Zhiru Zhang. [n. d.]. EqMap: FPGA LUT Remapping using E-Graphs. ([n. d.]). doi:10.1109/iccad66269.2025.11240672

[15] Neil D Jones, Carsten K Gomard, and Peter Sestoft. 1993. *Partial evaluation and automatic program generation*. Peter Sestoft.

[16] Thomas Koehler, Andrés Goens, Siddharth Bhat, Tobias Grosser, Phil Trinder, and Michel Steuwer. 2024. Guided equality saturation. *Proceedings of the ACM on Programming Languages* 8, POPL (2024), 1727–1758. doi:10.1145/3632900

[17] Sijie Kong. 2026. Nextmap Artifact for PLDI 2026. ACM. doi:10.5281/zenodo.19080665

[18] Chandrakana Nandi, Max Willsey, Adam Anderson, James R Wilcox, Eva Darulova, Dan Grossman, and Zachary Tatlock. 2020. Synthesizing structured CAD models with equality saturation and inverse transformations. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 31–44. doi:10.1145/3385412.3386012

[19] Charles Gregory Nelson. 1980. *Techniques for program verification.* Stanford University.

[20] Yan Pi, Hongji Zou, Tun Li, Wanxia Qu, and Hai Wan. 2023. ESFO: Equality Saturation for FIRRTL Optimization. In *Proceedings of the Great Lakes Symposium on VLSI 2023*. 581–586. doi:10.1145/3583781.3590239

[21] Brett Saiki, Oliver Flatt, Chandrakana Nandi, Pavel Panchekha, and Zachary Tatlock. 2021. Combining precision tuning and rewriting. In *2021 IEEE 28th Symposium on Computer Arithmetic (ARITH)*. IEEE, 1–8. doi:10.1109/arith51176.2021.00013

[22] Rudi Schneider, Marcus Rossel, Amir Shaikhha, Andrés Goens, Thomas Kœhler, and Michel Steuwer. 2025. Slotted E-Graphs: First-Class Support for (Bound) Variables in E-Graphs. *Proceedings of the ACM on Programming Languages* 9, PLDI (2025), 1888–1910. doi:10.1145/3729326

[23] Eytan Singher and Shachar Itzhaky. 2023. Colored E-Graph: Equality Reasoning with Conditions. *arXiv preprint arXiv:2305.19203* (2023). doi:10.48550/arXiv.2305.19203

[24] Gus Henry Smith, Colin Knizek, Daniel Petrisko, Zachary Tatlock, Jonathan Balkind, Gilbert Louis Bernstein, Haobin Ni, and Chandrakana Nandi. 2024. Scaling Program Synthesis Based Technology Mapping with Equality Saturation. *arXiv preprint arXiv:2411.11036* (2024). doi:10.48550/arXiv.2411.11036

[25] Gus Henry Smith, Benjamin Kushigian, Vishal Canumalla, Andrew Cheung, Steven Lyubomirsky, Sorawee Porncharoenwase, René Just, Gilbert Louis Bernstein, and Zachary Tatlock. 2024. Fpga technology mapping using sketch-guided program synthesis. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 416–432. doi:10.1145/3620665.3640387

[26] Gus Henry Smith, Zachary D Sisco, Thanawat Techaumnuaiwit, Jingtao Xia, Vishal Canumalla, Andrew Cheung, Zachary Tatlock, Chandrakana Nandi, and Jonathan Balkind. 2024. There and back again: A netlist's tale with much egraphin'. *arXiv preprint arXiv:2404.00786* (2024). doi:10.48550/arXiv.2404.00786

[27] Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. 2009. Equality saturation: a new approach to optimization. In *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 264–276. doi:10.2168/lmcs-7(1:10)2011

[28] Aleksei Tiurin, Dan R Ghica, and Nick Hu. 2025. E-Graphs With Bindings. *arXiv preprint arXiv:2505.00807* (2025). doi:10.48550/arXiv.2505.00807

[29] Alexa VanHattum, Rachit Nigam, Vincent T Lee, James Bornholt, and Adrian Sampson. 2021. Vectorization for digital signal processors via equality saturation. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 874–886. doi:10.1145/3445814.3446707

[30] Yisu Remy Wang, Shana Hutchison, Jonathan Leang, Bill Howe, and Dan Suciu. 2020. SPORES: sum-product optimization via relational equality saturation for large scale linear algebra. *arXiv preprint arXiv:2002.07951* (2020). doi:10.14778/3407790.3407799

[31] Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. 2021. Egg: Fast and extensible equality saturation. *Proceedings of the ACM on Programming Languages* 5, POPL (2021), 1–29. doi:10.1145/3434304

[32] Clifford Wolf, Johann Glaser, and Johannes Kepler. 2013. Yosys-a free Verilog synthesis suite. In *Proceedings of the 21st Austrian Workshop on Microelectronics (Austrochip)*, Vol. 97.

[33] Yichen Yang, Phitchaya Phothilimthana, Yisu Wang, Max Willsey, Sudip Roy, and Jacques Pienaar. 2021. Equality saturation for tensor graph superoptimization. *Proceedings of Machine Learning and Systems* 3 (2021), 255–268. doi:10.48550/arXiv.2101.01332

[34] Jiaqi Yin, Zhan Song, Chen Chen, Yaohui Cai, Zhiru Zhang, and Cunxi Yu. 2025. e-boost: Boosted E-Graph Extraction with Adaptive Heuristics and Exact Solving. *arXiv preprint arXiv:2508.13020* (2025). doi:10.1109/iccad66269.2025.11240719

[35] George Zakhour, Pascal Weisenburger, Jahrim Gabriele Cesario, and Guido Salvaneschi. 2025. Dis/Equality Graphs. *Proceedings of the ACM on Programming Languages* 9, POPL (2025), 2282–2305. doi:10.1145/3704913

[36] Yihong Zhang, Yisu Remy Wang, Oliver Flatt, David Cao, Philip Zucker, Eli Rosenthal, Zachary Tatlock, and Max Willsey. 2023. Better together: Unifying datalog and equality saturation. *Proceedings of the ACM on Programming Languages* 7, PLDI (2023), 468–492. doi:10.1145/3591239

[37] Philip Zucker. 2025. Awesome E-Graphs. https://github.com/philzook58/awesome-egraphs. A curated list of resources on e-graphs, equality saturation, and applications.

[38] Philip Zucker. 2025. Omelets Need Onions: E-graphs Modulo Theories via Bottom-up E-matching. *arXiv preprint arXiv:2504.14340* (2025). doi:10.48550/arXiv.2504.14340

Received 2025-11-13; accepted 2026-04-03