Challenging Benchmarks for Diagrammatic Equivalence of Circuits in TPTP and SMT-LIB

Julie Cailler[1], Noé Delorme[1], and Sophie Tourret[1,2]

[1] University of Lorraine, CNRS, INRIA, LORIA, Nancy, France
[2] Max Planck Institute for Informatics, Saarbrücken, Germany

Abstract

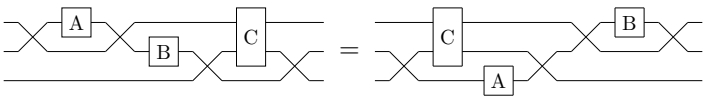
We introduce a new family of benchmarks for the problem of diagrammatic equivalence between circuits. Three variants of this problem are considered, ranging from basic to challenging, and benchmarks are generated for each variant. We provide first-order encodings in both TPTP and SMT-LIB formats, together with scripts that automatically generate benchmark instances, and evaluate these benchmarks on state-of-the-art automated theorem provers and SMT solvers.

1 Introduction

A recurrent complaint from benchmark repository maintainers and competition organizers is the lack of new contributions to their repositories.[1] It is already a time-consuming task to prepare an archive for reproducibility on an open repository such as Zenodo [16]. The extra effort of curating and documenting a selection of benchmarks for addition in a dedicated repository such as SMT-LIB or the TPTP library is important to the research community, but not well rewarded and thus not often undertaken. This paper goes against that trend. It describes a family of benchmarks for first-order automated theorem provers (ATPs) with arithmetic, as well as the tools to use to produce more of them.

The benchmarks and tools presented in this paper originate from an ongoing effort to produce certificates for proofs of diagrammatic equivalence between quantum circuits [6]. In recent years, graphical languages for quantum circuits have been extensively studied, leading to complete equational theories that enable circuits to be manipulated and rewritten while preserving their semantics [9–11, 14]. More broadly, quantum circuits are instances of the well-established framework of diagrammatic reasoning, which represents processes as string diagrams and captures their interactions in space and time through sequential and parallel composition [25]. These developments open the way to automated tasks such as circuit optimization, simplification, and formal verification.

In this paper, string diagrams are often referred to as *circuits*. Roughly speaking, circuits consist of generators—corresponding to elementary primitives—depicted as boxes connected by wires, and there are many diagrammatic representations of a given evolution. For instance, the two following circuits are equivalent.



Formally, circuits are expressed as terms freely generated from a set of generators under sequential and parallel composition. As a consequence, syntactically different terms may express the same circuit, depending on the order and structure in which compositions are formed.

[1] It is discussions on this very topic at the Dagstuhl seminar 25441 that motivated the existence of this paper.

Therefore, the intuitive notion of diagrammatic equivalence is fully captured by a collection of coherence equations [25], which can be used to rewrite equivalent terms. This naturally leads to the problem of diagrammatic equivalence: deciding, given two terms, whether they can be rewritten into one another using the coherence equations.

We provide tools for generating diagrammatically equivalent terms, which produced the benchmarks studied in this work. In the context of our original motivation—namely, the certification of quantum circuit equivalences—the modest performance of ATPs on these benchmarks, together with the lack of maturity of the associated certificate production techniques, highlights both the difficulty of the problem and the need for further progress in arithmetic and equational reasoning. For these reasons, we believe that the benchmarks themselves are of independent interest to the ATP community, as a challenge.

We consider three variants of the diagrammatic equivalence problem. The first one is the most general and corresponds exactly to the settings described above. In the second one, we consider circuits with no generators (i.e., no boxes) that allow the expression of any permutation. Finally, the last one is a restriction of the second one to a simpler configuration.

In this paper, we describe the three variants of the problems (Section 2) and explain the encoding built for them (Section 3). We also discuss the tools used to create the benchmarks (Section 4) and the evaluation of these benchmarks on state-of-the-art ATPs (Section 5). The tools and benchmarks used to run the experiments are available on Zenodo [7, 8].

2 Problem Description

Diagrammatic reasoning and its underlying notion of *circuits* is captured formally within the formalism of *symmetric monoidal categories* [25], and more precisely *PROPs*. In this section, we give a term-based definition of this structure and present the three variants of the diagrammatic equivalence problem that we consider in the rest of the paper.

2.1 Circuits and Their Graphical Representation

Given a set of generators Σ equipped with two functions $\mathrm{dom}, \mathrm{cod} : \Sigma \to \mathbb{N}$, we define circuits as terms of type $\mathbf{Circ}_\Sigma(n, m)$ for any $n, m \in \mathbb{N}$, as follows.

$$\frac{g \in \Sigma}{g \in \mathbf{Circ}_\Sigma(\mathrm{dom}(g), \mathrm{cod}(g))} \quad \frac{n \in \mathbb{N}}{\mathrm{id}_n \in \mathbf{Circ}_\Sigma(n, n)} \quad \frac{n, m \in \mathbb{N}}{\sigma_{n,m} \in \mathbf{Circ}_\Sigma(n+m, m+n)}$$

$$\frac{C_i \in \mathbf{Circ}_\Sigma(n_i, m_i) \quad m_1 = n_2}{C_1 \fatsemi C_2 \in \mathbf{Circ}_\Sigma(n_1, m_2)} \quad \frac{C_i \in \mathbf{Circ}_\Sigma(n_i, m_i)}{C_1 \otimes C_2 \in \mathbf{Circ}_\Sigma(n_1 + n_2, m_1 + m_2)}$$

Let $\mathbf{Circ}_\Sigma = \cup_{n,m \in \mathbb{N}} \mathbf{Circ}_\Sigma(n, m)$ be the collection of all circuits. We may write $C : n \to m$ when $C \in \mathbf{Circ}_\Sigma(n, m)$. Moreover, we extend dom and cod to circuits as follows: for any $C \in \mathbf{Circ}_\Sigma(n, m)$, let $\mathrm{dom}(C) \coloneqq n$ be the *domain* of C and $\mathrm{cod}(C) \coloneqq m$ be the *codomain* of C. Intuitively, $\mathrm{dom}(C)$ represents the number of inputs and $\mathrm{cod}(C)$ the number of outputs. For any $n \in \mathbb{N}$, the circuit id_n is the identity of size n. For any $n, m \in \mathbb{N}$, the circuit $\sigma_{n,m}$ swaps two registers of size n and m respectively. The circuit $C_1 \fatsemi C_2$ represents a sequential composition of C_1 and C_2. Note that this composition is only possible when $m_1 = n_2$. The circuit $C_1 \otimes C_2$ represents a parallel composition of C_1 and C_2. Note that neither operation is commutative.

Graphically, circuits are depicted as *string diagrams*, as presented in the introduction. This representation makes use of the two dimensions offered by drawings. The sequential composition is drawn horizontally and the parallel composition vertically. In that context, a circuit with n

inputs and m outputs is represented as a box with n input wires drawn on its left-hand side and m output wires drawn on its right-hand side. There are some special cases: the circuit id_0 is the empty circuit and is not represented, the circuit id_n is represented as n wires in parallel when $n \neq 0$, and the circuit $\sigma_{n,m}$ is represented as a swapping of n wires with m wires. We may draw dashed gray lines to identify (sub-)circuits when helpful. Several examples are available in Figure 1.

In order to precisely capture the intuitive notion of circuits, we need equivalence equations known as *coherence equations* to ensure that syntactically distinct terms that are represented by the same diagram are equal. The coherence equations are defined in Figure 2.

Let $\mathbf{Circ}_\Sigma \vdash \cdot = \cdot$ be the smallest congruence relation satisfying the coherence equations. That is, $\mathbf{Circ}_\Sigma \vdash C_1 = C_2$ means that C_1 can be rewritten into C_2 by applying the coherence equations to sub-circuits. This relation corresponds exactly to the intuitive notion of *diagrammatic equivalence* between circuits. This result is known as *coherence theorem* and relates equivalent circuits to isomorphic graphs [25].

2.2 Challenging Equivalence Checking Problems

This well-studied formalism of circuits offers an interesting equivalence checking problem. In this paper, we propose three variants of the problem together with their encoding in SMT-LIB and TPTP, yielding challenging benchmarks.

The first problem, called *Circuit Equivalence* (CE), is the most general and corresponds to the decision problem associated with the equivalence relation $\mathbf{Circ}_\Sigma \vdash \cdot = \cdot$.

Problem 1. *Given* $C_1, C_2 \in \mathbf{Circ}_\Sigma$*, decide whether* $\mathbf{Circ}_\Sigma \vdash C_1 = C_2$.

The second problem, namely *Permutation Circuit Equivalence* (PCE), is a particular instance of CE in which $\Sigma = \varnothing$. When there are no generators, circuits are solely made of wires and always have an identical number of inputs and outputs. In fact, any circuit of size n can be seen as a permutation, i.e., a bijective map from $\{1, \ldots, n\}$ to itself. The PCE problem is defined as follows.

Problem 2. *Given* $C_1, C_2 \in \mathbf{Circ}_\varnothing$*, decide whether* $\mathbf{Circ}_\varnothing \vdash C_1 = C_2$.

The third problem, referred to as *Simplified Permutation Circuit Equivalence* (SPCE), is a restricted version of PCE. It can express only concrete instances of PCE rather than families of problems parameterized by an arbitrary number of wires. More precisely, to express any permutation, we only need the two atomic circuits $\mathrm{id} \coloneqq \mathrm{id}_1$ and $\sigma \coloneqq \sigma_{1,1}$, together with the

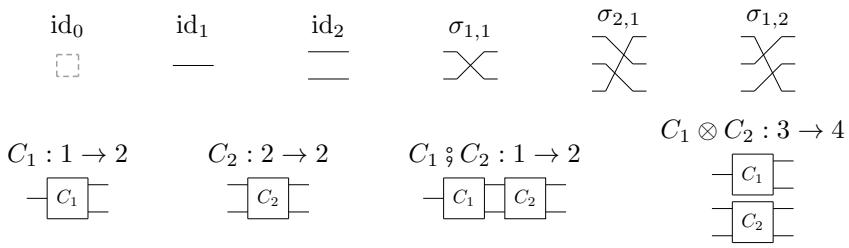


Figure 1: Examples of Circuits.

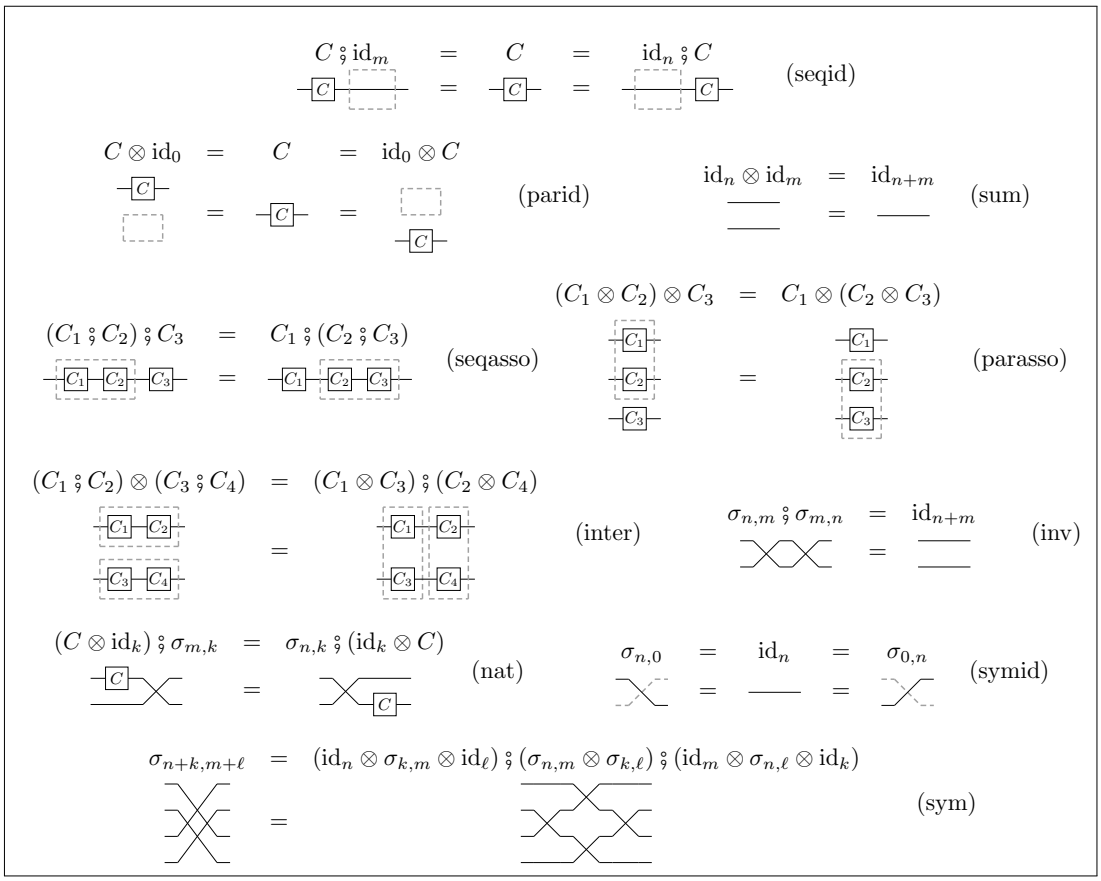


Figure 2: Coherence equations defined for any $C \in \mathbf{Circ}_\Sigma(n, m)$ and any $C_i \in \mathbf{Circ}_\Sigma(n_i, m_i)$ where Equation (seqasso) has to be satisfied whenever $m_1 = n_2$ and $m_2 = n_3$, and Equation (inter) has to be satisfied whenever $m_1 = n_2$ and $m_3 = n_4$. Notice that a single wire depicts arbitrarily many wires to simplify the drawings.

two compositions $\fatsemi$ and $\otimes$. The collections of *simplified permutation circuits* $\mathbf{Perm}(n)$ are defined as follows.

$$\frac{}{\mathrm{id} \in \mathbf{Perm}(1)} \qquad \frac{}{\sigma \in \mathbf{Perm}(2)}$$

$$\frac{C_i \in \mathbf{Perm}(n)}{C_1 \fatsemi C_2 \in \mathbf{Perm}(n)} \qquad \frac{C_i \in \mathbf{Perm}(n_i)}{C_1 \otimes C_2 \in \mathbf{Perm}(n_1 + n_2)}$$

Let $\mathbf{Perm} = \cup_{n \in \mathbb{N}} \mathbf{Perm}(n)$ be the collection of all simplified permutation circuits. In this representation, the coherence equations can also be simplified and are given in Figure 3. This new representation comes with a few noteworthy changes:

- The rules (seqasso), (parasso) and (inter) are unchanged.
- The (parid) rules have disappeared, since it is no longer possible to represent the empty circuit.

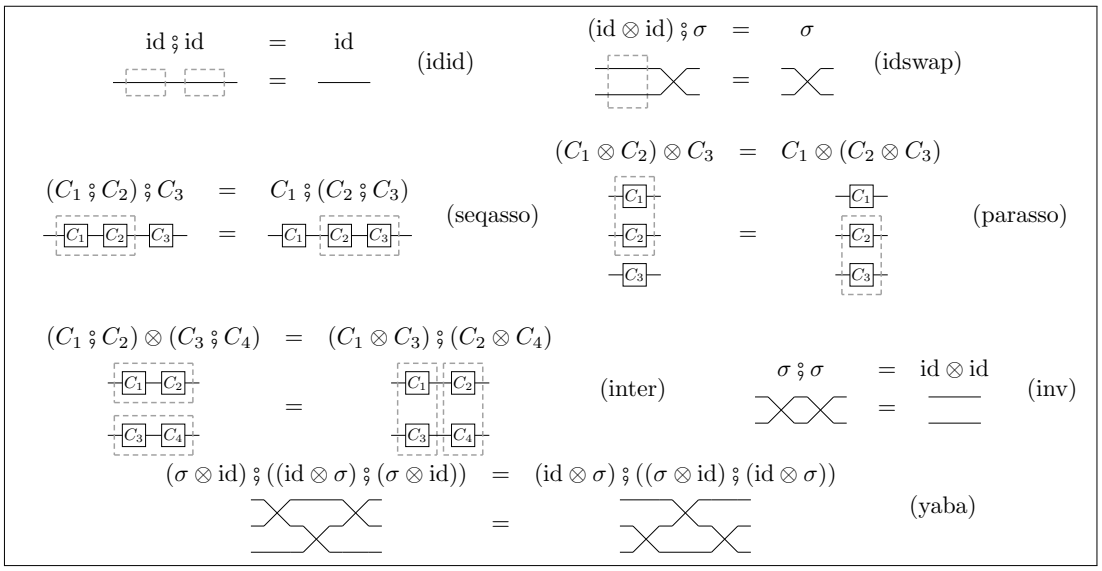


Figure 3: Simplified coherence equations defined for any $C_i \in \mathbf{Perm}(n_i)$ where Equation (seqasso) has to be satisfied whenever $n_1 = n_2 = n_3$, and Equation (inter) has to be satisfied whenever $n_1 = n_2$ and $n_3 = n_4$.

- The (seqid) rules need to be defined only for id and σ, becoming (idid) and (idswap). Notice that there is exactly one equation in (idid) and one equation in (idswap) rather than two in each. In the case of (idid), this is because the two equations in (seqid) coincide when $t = \mathsf{id}$. For (idswap), one side (e.g., $\sigma \fatsemi (\mathrm{id} \otimes \mathrm{id})$) can be derived from the other (e.g., $(\mathrm{id} \otimes \mathrm{id}) \fatsemi \sigma$) using (inv) and (seqasso).
- The (nat) rule is replaced by a slightly simpler rule called (yaba) after its proposers, Yang and Baxter [19]. Note that, due to the associativity of $\fatsemi$ and $\otimes$, this rule can be represented in several equivalent forms as an equation on terms.
- The (sum), (sym), and (symid) rules have disappeared, as identities and swaps now have fixed domains and codomains.

The SPCE problem is defined as follows.

Problem 3. *Given* $C_1, C_2 \in \mathbf{Perm}$*, decide whether* $\mathbf{Perm} \vdash C_1 = C_2$.

Remark. *To our knowledge, the complexity of the circuit equivalence problem has not been established in general. However, in the CE case, the problem can be embedded into graph isomorphism, placing it in the quasi-polynomial time class [1].*

3 Encodings

We present three encodings in first-order logic with equality for the problems introduced in the previous section, using linear arithmetic to represent constraints on rule application.

3.1 Encoding of Circuit Equivalence

Let Σ be a fixed set of symbols representing generators. We introduce the following syntax to encode the terms of the type system $\mathbf{Circ}_\Sigma$ as first-order terms:

$$t ::= \mathsf{id}(n) \mid \mathsf{swap}(n,m) \mid \mathsf{gen}(g,n,m) \mid \mathsf{seq}(t,t) \mid \mathsf{par}(t,t) \text{ for } n,m \in \mathbb{N}, g \in \Sigma.$$

The interpretation is straightforward: $\mathsf{id}(n)$ is id_n and $\mathsf{swap}(n,m)$ is $\sigma_{n,m}$; the term $\mathsf{gen}(g,n,m)$ represents the generator g with n input wires and m output wires; circuits represented by terms t_1 and t_2 are put in sequence using $\mathsf{seq}(t_1,t_2)$ and in parallel using $\mathsf{par}(t_1,t_2)$.

In the encoding, the functions dom and cod are computed as follows.

$$\begin{array}{rcl@{\qquad}rcl}
\mathsf{dom}(\mathsf{id}(n)) & = & n & \mathsf{cod}(\mathsf{id}(n)) & = & n \\
\mathsf{dom}(\mathsf{swap}(n,m)) & = & n+m & \mathsf{cod}(\mathsf{swap}(n,m)) & = & m+n \\
\mathsf{dom}(\mathsf{gen}(g,n,m)) & = & n & \mathsf{cod}(\mathsf{gen}(g,n,m)) & = & m \\
\mathsf{dom}(\mathsf{seq}(t_1,t_2)) & = & \mathsf{dom}(t_1) & \mathsf{cod}(\mathsf{seq}(t_1,t_2)) & = & \mathsf{cod}(t_2) \\
\mathsf{dom}(\mathsf{par}(t_1,t_2)) & = & \mathsf{dom}(t_1)+\mathsf{dom}(t_2) & \mathsf{cod}(\mathsf{par}(t_1,t_2)) & = & \mathsf{cod}(t_1)+\mathsf{cod}(t_2)
\end{array}$$

The main difference between this syntax and that of circuits is that we define $\mathsf{dom}(g)$ and $\mathsf{cod}(g)$ directly from terms instead of considering these functions already defined on generators. Thus, two generators with the same name but different numbers of in- or outgoing wires will be considered distinct.

In fact, terms are an over-approximation of circuits. Consider, for example, the term $\mathsf{seq}(\mathsf{id}(2),\mathsf{id}(3))$. It is syntactically correct but does not correspond to any well-typed circuit in $\mathbf{Circ}_\Sigma(2,3)$ since $\mathsf{cod}(\mathsf{id}(2)) = \mathrm{cod}(\mathrm{id}_2) = 2 \neq 3 = \mathrm{dom}(\mathrm{id}_3) = \mathsf{dom}(\mathsf{id}(3))$, due to the mismatch in the number of wires in id_2 and id_3. We call *proper* the terms that correspond to circuits. Using dom and cod, we can define proper terms by induction: $\mathsf{id}(n)$, $\mathsf{swap}(n,m)$ and $\mathsf{gen}(g,n,m)$ are proper for all $n,m \in \mathbb{N}$, $\mathsf{par}(t_1,t_2)$ is proper if t_1 and t_2 are proper, and $\mathsf{seq}(t_1,t_2)$ is proper if t_1 and t_2 are proper and if $\mathsf{cod}(t_1) = \mathsf{dom}(t_2)$. In that way, proper terms correspond exactly to circuits.

The coherence equations form an equivalence relation on circuits, and thus on proper terms. In our encoding, described in Figure 4, we use this relation as the equality relation between terms, but there is a catch. Some equations hold unconditionally in one direction, and with a condition in the other direction. By using them unguarded, one can equate proper terms with improper terms, leading to contradictions.

Example 1. *Let t be a term such that $\mathsf{dom}(t) = 2$. Using $(\text{seqid})_{e1}$ without guard, we can obtain $\mathsf{seq}(\mathsf{id}(3),t) = t$, and thus, $2 = \mathsf{dom}(t) = \mathsf{dom}(\mathsf{seq}(\mathsf{id}(3),t)) = 3$, which is a contradiction.*

We solve this issue by adding guards to ambiguous equations, ensuring that proper terms can only be equal to proper terms. For $(\text{seqid})_{e1}$, it is enough to replace the equations $\mathsf{seq}(\mathsf{id}(n),t) = t$ and $\mathsf{seq}(t,\mathsf{id}(n)) = t$ by the implications:

$$\begin{aligned}
\mathsf{dom}(t) = n &\longrightarrow \mathsf{seq}(\mathsf{id}(n),t) = t, \\
\mathsf{cod}(t) = n &\longrightarrow \mathsf{seq}(t,\mathsf{id}(n)) = t.
\end{aligned}$$

The only other equation that needs a guard is $(\text{inter})_{e1}$, which would be as follows without guard

$$\mathsf{par}(\mathsf{seq}(t,u),\mathsf{seq}(v,w)) = \mathsf{seq}(\mathsf{par}(t,v),\mathsf{par}(u,w)).$$

$$\begin{array}{rcl} \mathsf{dom}(t) = n & \longrightarrow & \mathsf{seq}(\mathsf{id}(n), t) = t \\ \mathsf{cod}(t) = n & \longrightarrow & \mathsf{seq}(t, \mathsf{id}(n)) = t \end{array} \quad (\text{seqid})_{e1} \qquad \begin{array}{rcl} \mathsf{par}(t, \mathsf{id}(0)) & = & t \\ \mathsf{par}(\mathsf{id}(0), t) & = & t \end{array} \quad (\text{parid})_{e1}$$

$$\mathsf{id}(n+m) = \mathsf{par}(\mathsf{id}(n), \mathsf{id}(m)) \qquad (\text{sum})_{e1}$$

$$\mathsf{seq}(\mathsf{seq}(t,u),v) = \mathsf{seq}(t, \mathsf{seq}(u,v)) \qquad (\text{seqasso})_{e1}$$

$$\mathsf{par}(\mathsf{par}(t,u),v) = \mathsf{par}(t, \mathsf{par}(u,v)) \qquad (\text{parasso})_{e1}$$

$$\mathsf{cod}(t) = \mathsf{dom}(u) \longrightarrow \mathsf{par}(\mathsf{seq}(t,u), \mathsf{seq}(v,w)) = \mathsf{seq}(\mathsf{par}(t,v), \mathsf{par}(u,w)) \qquad (\text{inter})_{e1}$$

$$\begin{array}{rcl} \mathsf{swap}(n,0) & = & \mathsf{id}(n) \\ \mathsf{swap}(0,n) & = & \mathsf{id}(n) \end{array} \quad (\text{symid})_{e1}$$

$$\mathsf{seq}(\mathsf{swap}(n,m), \mathsf{swap}(m,n)) = \mathsf{id}(n+m) \qquad (\text{inv})_{e1}$$

$$\mathsf{seq}(\mathsf{par}(u, \mathsf{id}(k)), \mathsf{swap}(\mathsf{cod}(u), k)) = \mathsf{seq}(\mathsf{swap}(\mathsf{dom}(u), k), \mathsf{par}(\mathsf{id}(k), u)) \qquad (\text{nat})_{e1}$$

$$\begin{array}{rcl} \mathsf{swap}(n+k, m+l) & = & \mathsf{seq}(\mathsf{seq}(\mathsf{par}(\mathsf{id}(n), \mathsf{par}(\mathsf{swap}(k,m), \mathsf{id}(l))), \\ & & \quad \mathsf{par}(\mathsf{swap}(n,m), \mathsf{swap}(k,l))), \\ & & \mathsf{par}(\mathsf{id}(m), \mathsf{par}(\mathsf{swap}(n,l), \mathsf{id}(k)))) \end{array} \quad (\text{sym})_{e1}$$

Figure 4: Encoding of the coherence equations of circuits.

Indeed, having a proper term on the right-hand side of interchange does not guarantee that the left-hand side is also proper (whereas a proper left-hand side ensures a proper right-hand side). To preserve properness from right to left, the term has to additionally verify $\mathsf{cod}(t) = \mathsf{dom}(u)$ and $\mathsf{cod}(v) = \mathsf{dom}(w)$. However, the latter can be omitted from the encoding because it follows from the former and from $\mathsf{seq}(\mathsf{par}(t,v), \mathsf{par}(u,w))$ being proper: properness implies $\mathsf{cod}(\mathsf{par}(t,v)) = \mathsf{dom}(\mathsf{par}(u,w))$, i.e. $\mathsf{cod}(t) + \mathsf{cod}(v) = \mathsf{dom}(u) + \mathsf{dom}(w)$, which together with the guard $\mathsf{cod}(t) = \mathsf{dom}(u)$ yields $\mathsf{cod}(v) = \mathsf{dom}(w)$. The other equations are encoded in a straightforward way.

3.2 Encoding of Permutation Circuit Equivalence

Next, we present two encodings, one for the PCE problem and one for the SPCE problem.

3.2.1 Straightforward Encoding of Permutation Circuits.

For permutation circuits, $\Sigma = \emptyset$, so the previous syntax can be simplified by removing gen altogether as follows:

$$t ::= \mathsf{id}(n) \mid \mathsf{swap}(n,m) \mid \mathsf{seq}(t,t) \mid \mathsf{par}(t,t) \text{ for } n, m \in \mathbb{N}.$$

This simplification does not affect the equations of Figure 4.

3.2.2 Simplified Encoding of Permutation Circuits.

To make this problem more accessible to theorem provers, we also designed an encoding of SPCE to reduce the amount of arithmetic reasoning required. The syntax is affected as follows:

$$t ::= \mathsf{id} \mid \mathsf{swap} \mid \mathsf{seq}(t,t) \mid \mathsf{par}(t,t).$$

$$\mathsf{seq}(\mathsf{id},\mathsf{id}) = \mathsf{id} \quad (\text{idid})_{e2} \qquad \mathsf{seq}(\mathsf{par}(\mathsf{id},\mathsf{id}),\mathsf{swap}) = \mathsf{swap} \quad (\text{idswap})_{e2}$$

$$\mathsf{seq}(\mathsf{seq}(t,u),v) = \mathsf{seq}(t,\mathsf{seq}(u,v)) \quad (\text{seqasso})_{e2}$$

$$\mathsf{par}(\mathsf{par}(t,u),v) = \mathsf{par}(t,\mathsf{par}(u,v)) \quad (\text{parasso})_{e2}$$

$$\mathsf{dom}(t) = \mathsf{dom}(u) \longrightarrow \mathsf{par}(\mathsf{seq}(t,u),\mathsf{seq}(v,w)) = \mathsf{seq}(\mathsf{par}(t,v),\mathsf{par}(u,w)) \quad (\text{inter})_{e2}$$

$$\mathsf{seq}(\mathsf{swap},\mathsf{swap}) = \mathsf{par}(\mathsf{id},\mathsf{id}) \quad (\text{inv})_{e2}$$

$$\mathsf{seq}(\mathsf{par}(\mathsf{swap},\mathsf{id}),\mathsf{seq}(\mathsf{par}(\mathsf{id},\mathsf{swap}),\mathsf{par}(\mathsf{swap},\mathsf{id}))) = \mathsf{seq}(\mathsf{par}(\mathsf{id},\mathsf{swap}),\mathsf{seq}(\mathsf{par}(\mathsf{swap},\mathsf{id}),\mathsf{par}(\mathsf{id},\mathsf{swap}))) \quad (\text{yaba})_{e2}$$

Figure 5: Encoding of the Coherence equations of Simplified Permutation Circuits.

The dom and cod functions are now such that $\mathsf{dom}(\mathsf{id}) = \mathsf{cod}(\mathsf{id}) = 1$ and $\mathsf{dom}(\mathsf{swap}) = \mathsf{cod}(\mathsf{swap}) = 2$. They are unchanged on seq and par. This new syntax gives rise to the simplified equations described in Figure 5, where the changes in the encoding mirror those from PCE to SPCE (i.e., from Figure 2 to Figure 3).

Remark. *Unfortunately, we are not completely rid of arithmetic constraints, since the guard from* $(\text{inter})_{e1}$ *is still used. We attempted to exploit term-ordering constraints to ensure that the* $(\text{inter})_{e2}$ *rule is always oriented left-to-right, removing the need for the guard. However, we determined that it was not possible to preserve this orientation over all equivalent terms, so this idea had to be abandoned.*

3.3 Encoding Files

For each problem variant, we provide theory files encoding the corresponding coherence equations in SMT-LIB [4] and TPTP [28] formats [8]. Each file formalizes the corresponding syntax of circuit terms and the coherence equations described in Section 3. Benchmark instances are produced as satisfiability problems by asserting the negation of an equality between two generated terms.

SMT-LIB. We provide the SMT-LIBv2 theory files: `encoding_CE.smt2`, `encoding_PCE.smt2`, and `encoding_SPCE.smt2`. All three files make use of the UFLIA logic, combining uninterpreted functions with linear integer arithmetic, the latter being required to express the guards on the (seqid) and (inter) rules.

TPTP. We also provide encoding files in TPTP: `encoding_CE.ax`, `encoding_PCE.ax`, and `encoding_SPCE.ax`. All three files make use of the `TFF` format, which supports typed first-order logic with arithmetic, again required to handle the guards on rules.

4 Benchmark Generators

In addition to the encoding files, we provide a collection of benchmark instances together with the tools used to generate them. The benchmarks target the three circuit equivalence problems:

(a) Circuit 1 (b) Circuit 2

Figure 6: Two Equivalent Circuits for the SPCE problem.

```
-
```

(a) id

```
\
/
```

(b) swap

```
\ - \ \ -
/ \ / / \
- / - - /
```

(c) Representation of 6a

```
- \ \ - - \
\ / / \ - /
/ - - / - -
```

(d) Representation of 6b

Figure 7: ASCII Representation of Terms for the SPCE Problem.

CE, PCE, and SPCE. For each problem class, we provide dedicated `Python` scripts that automatically generate equality problems between two syntactically distinct yet diagrammatically equivalent circuit terms. All generators follow a two-phase approach: first producing equivalent string diagrams, then translating them into terms. Note that a single diagram can have multiple encodings. Those files are available in Zenodo [8].

4.1 Generation of SPCE Instances

The script `main_SPCE.py` generates benchmarks for the SPCE problem. The benchmark generator operates on a graphical ASCII representation of circuits, which is first produced from the script parameters and then transformed into an equivalent one by applying rewrite rules.

In this setting, circuits are built exclusively from the generators id (represented by `-`) and swap (represented by `\` for the descending wire and `/` for the ascending wire), together with sequential and parallel compositions. Example SPCE circuits are presented in Figure 6, while their corresponding ASCII representations, as well as additional terms, are available in Figure 7.

Internally, circuits are represented as rectangular grids, where rows correspond to wires and columns correspond to successive layers of sequential compositions. The script is parameterized by the number of rows (`-r`) and columns (`-c`), which directly controls the size of the original circuit. The `-s` flag gives the number of rule applications. For instance, circuits 7c and 7d were generated by `main_SPCE.py -r=3 -c=5 -s 100`.

Starting from an initial number of rows r and columns c, the benchmark generator starts with an id-only grid of size $r \times c$ and then randomly adds binary swaps. Then, in order to produce an equivalent circuit, the benchmark generator selects a rectangular subpart of the grid and applies a sequence of transformations corresponding to a subset of the simplified coherence equations $(\text{idid})_{e2}$, $(\text{idswap})_{e2}$, $(\text{inv})_{e2}$, and $(\text{yaba})_{e2}$ of Figure 5. Note that the $(\text{idid})_{e2}$ rule can modify the number of columns of the circuit.

The remaining rules are handled by the translation into terms. In practice, the benchmark generator selects a split direction (vertical or horizontal), then checks if the cut is feasible (i.e., by not breaking a swap). If the split is allowed, a sequential (seq) or parallel (par) composition of the two sub-circuits thus obtained is generated.

(a) swap$(2,1)$ (b) par(id(1), id(1)) or id(2) (c) seq(swap$(2,1)$, swap$(1,2)$)

Figure 8: Representation of Elements of PCE.

(a) Left Pattern (b) Right Pattern (c) Left Pattern (d) Right Pattern

$(\text{inv})_{e1}$ Rule Patterns for swap$(2,1)$ $(\text{sym})_{e1}$ Rule Pattern for swap$(2,2)$

Figure 9: Two Instances of the $(\text{sym})_{e1}$ and $(\text{inv})_{e1}$ Rules of PCE.

4.2 Generation of PCE Instances

The PCE script `main_PCE.py` extends the previous script to handle permutation circuits with arbitrary-size swaps, as illustrated in Figure 8. It takes an additional parameter `-a` that controls the maximal size of a swap.

Unlike the previous script, this one starts by randomly inserting n-ary swaps, together with two other patterns: $(\text{sym})_{e1}$ and $(\text{inv})_{e1}$, of which the ASCII representation is available in Figure 9. These patterns can randomly arise with the insertion of swaps. However, to produce more realistic and diverse circuits, we chose to include dedicated functions to add them.

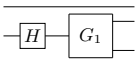
Next, as in the SPCE case, random equivalence-preserving rewrite rules are applied to generate an equivalent circuit. Three rules of Figure 4 are explicitly managed: sequential identity $(\text{seqid})_{e1}$, involution $(\text{inv})_{e1}$, and symmetry decomposition $(\text{sym})_{e1}$. The first one inserts or deletes columns of id, modifying the number of columns of the grid. The latter two detect specific graphical patterns and rewrite them according to the corresponding rule.

In practice, the benchmark generator selects a rule (identity insertion or deletion, $(\text{sym})_{e1}$ or $(\text{inv})_{e1}$ with a direction) and retrieves all applicable positions of the rule, then chooses one and applies it to the circuit. As an example, `main_PCE.py -r=5 -c=5 -a=4 -s=200` was used to generate the two equivalent circuits of Figure 10.

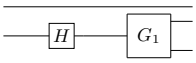
Moreover, an additional rule is added for id management: whenever a parallel composition of two id nodes is generated, the benchmark generator tries to rewrite it as another equivalent

(a) Circuit 1 (b) Circuit 2

Figure 10: Example Equivalent Circuits for PCE.



(a) Circuit 1



(b) Circuit 2

Figure 11: Two Equivalent Circuits 11a and 11b for CE with 2 inputs and 3 outputs.

decomposition. For instance, $\mathsf{par}(\mathsf{id}(2), \mathsf{id}(3))$ can be turned into $\mathsf{id}(5)$ or $\mathsf{par}(\mathsf{id}(1), \mathsf{id}(4))$, ensuring more randomness. The translation into terms is analogous to the SPCE case. Finally, the script also includes a validation step that ensures that every input wire exits at the same position in both circuits.

4.3 Generation of CE Instances

The CE script (`main_CE.py`) produces the most general benchmarks, allowing arbitrary generators. Unlike the previous benchmark generators, it is not ASCII-based; instead, it relies on an internal graph structure.

Such a graph consists of nodes, which are either identities ($\mathsf{id}(n)$) or generators ($\mathsf{gen}(n, m)$), both parameterized by their respective arities. Nodes maintain lists of predecessors and successors, as well as two lists corresponding to their input and output wires (including their positions and connected nodes). Finally, each node has a width, which corresponds to the number of columns it occupies.

To build a circuit, the script begins by introducing a set of generators, which are then combined into a directed planar acyclic graph. This graph is first organized into vertical layers using a topological sort, ensuring global wire connection consistency from inputs to outputs. Note that acyclicity and planarity are guaranteed by construction. Identity nodes are subsequently inserted as needed to fill the gaps and preserve wire alignment between layers. Two special nodes, `begin` and `end`, are added to collect wires that have no input or no output, respectively. Note that, in the scripts, generators with no input or no output are not allowed.

Moreover, in order to be closer to real-life circuits, we add three pre-defined generators, corresponding to Clifford generators [17]: the Hadamard generator H ($\mathsf{gen}(1, 1)$), the phase generator S ($\mathsf{gen}(1, 1)$), and the $CNOT$ generator ($\mathsf{gen}(2, 2)$). A dedicated option `--clifford` forces the script to only use those predefined generators.

This process generates the original circuit in its most compact form, where each node is assigned a row and a temporary column. Next, identity nodes are randomly inserted between existing nodes (corresponding to rule $(\text{seqid})_{e1}$ in Figure 4), and the column assignments of the nodes are updated accordingly.

An instance of a compact graph, together with one of its extended versions, is given in Figure 11, and its internal representation in Figure 12. This circuit was generated by `main_CE.py -r=2 -c=2`.

In this formalism, a circuit is parameterized by its number of inputs, outputs, and columns. A generator is represented as `Gen(name, dom, cod)`, while an identity node is written as `Id(n)`. The remaining information in all nodes is: `@([row of the first input wire; row of the first output wire], column, width) | input wires list | output wires list`.

From this underlying circuit graph, two equivalent terms are extracted by repeatedly merging adjacent nodes using either sequential or parallel composition. To this end, the algorithm first selects a node and a split direction (vertical or horizontal). It then attempts to merge the node according to the chosen direction: horizontally with its predecessor or successor, or vertically with the node above or below. If the merge is possible, a new node—either par or

```
Circuit(2,2,4):
  [0] Gen(begin, 0, 2)@([0;0], 0, 1) | pred: [-] | succ: [[2] Id(1), [3] Gen(H, 1, 1)]
  [2] Id(1)@([0;0], 1, 1) | pred: [[0] Gen(begin, 0, 2)] | succ: [[4] Id(1)]
  [3] Gen(H,1,1)@([1;1], 1, 1) | pred: [[0] Gen(begin, 0, 2)] | succ: [[5] Gen(Gen1, 1, 1)]
  [4] Id(1)@([0;0], 2, 1) | pred: [[2] Id(1)] | succ: [[1] Gen(end, 3, 0)]
  [5] Gen(Gen1, 1, 2)@([1;1], 2, 1) | pred: [[3] Gen(H, 1, 1)] | succ: [[1] Gen(end, 3, 0)]
  [1] Gen(end, 3, 0)@([0;0], 3, 1) | pred: [[4] Id(1), [5] Gen(Gen1, 1, 1)] | succ: [-]
```

(a) Compact form of the circuit

```
Circuit(2, 2, 6):
  [0] Gen(begin, 0, 2)@([0;0], 0, 1) | pred: [-] | succ: [[2] Id(1), [6] Id(1)]
  [2] Id(1)@([0;0], 1, 1) | pred: [[0] Gen(begin, 0, 2)] | succ: [[7] Id(1)]
  [6] Id(1)@([1;1], 1, 1) | pred: [[0] Gen(begin, 0, 2)] | succ: [[3] Gen(H, 1, 1)]
  [7] Id(1)@([0;0], 2, 1) | pred: [[2] Id(1)] | succ: [[8] Id(1)]
  [3] Gen(H, 1, 1)@([1;1], 2, 1) | pred: [[6] Id(1)] | succ: [[9] Id(1)]
  [8] Id(1)@([0;0], 3, 1) | pred: [[7] Id(1)] | succ: [[4] Id(1)]
  [9] Id(1)@([1;1], 3, 1) | pred: [[3] Gen(H, 1, 1)] | succ: [[5] Gen(Gen1, 1, 1)]
  [4] Id(1)@([0;0], 4, 1) | pred: [[8] Id(1)] | succ: [[1] Gen(end, 3, 0)]
  [5] Gen(Gen1, 1, 2)@([1;1], 4, 1) | pred: [[9] Id(1)] | succ: [[1] Gen(end, 3, 0)]
  [1] Gen(end, 3, 0)@([0;0], 5, 1) | pred: [[4] Id(1), [9] Id(1)] | succ: [-]
```

(b) Extended version of the circuit

Figure 12: Corresponding Representation for 11a and 11b.

seq, parameterized by the two chosen nodes—is created, then added to the graph, and the two merged nodes are removed. During parallel merges, the benchmark generator randomly inserts swap nodes and, as in the PCE generator, performs identity rewritings.

The resulting instances make use of the full set of coherence equations and constitute the most challenging benchmarks in our collection.

Overall, the three scripts generate circuits equivalent by construction, enabling the generation of large families of various circuit equivalence problems.

5 Evaluations

This section describes the experimental setup used to evaluate the benchmark families introduced in the previous sections. Our objective is to assess both the difficulty of the proposed circuit equivalence problems and the practical impact of the different encodings and tool configurations.

Tools. Beyond performance, our two main criteria when selecting tools for this work were (i) support for arithmetic reasoning, and (ii) the ability to produce proof certificates. The latter requirement is stronger than solving an instance and outputting a proof trace, as the certificate must be independently checkable. This criterion stems from the broader quantum circuit verification pipeline that motivated this work [6], but, although proof production was tested, a full evaluation of the verification pipeline goes beyond the scope of the current evaluation.

With respect to arithmetic reasoning, SMT solvers are generally well-suited, whereas not all first-order theorem provers provide dedicated decision procedures for arithmetic. Conversely,

due to their internal reasoning techniques, traditional theorem provers are often more suitable for producing detailed proofs.

We experimented with several automated reasoning tools, including egglog [29], Twee [26], E [23], Vampire [5], Z3 [13], and cvc5 [3]. We first excluded provers that do not support arithmetic reasoning, namely E and Twee. We then evaluated the proof-production capabilities of the remaining tools and identified two main pipelines, thus excluding egglog and Z3.

The first one relies on cvc5, which can generate proofs in formats such as LSFC [27], Eunoia [15], and Alethe [24]. The latter two formats can be translated and independently checked by proof assistants such as LambdaPi [12, 18] or Isabelle [21]. Although not all proof steps are currently supported, the implemented subset is sufficient for our encodings.

Regarding ATPs, Vampire can handle arithmetic, and it can generate SMT-style proofs from FOF encodings that are externally checkable, for instance by cvc5 [22]. Moreover, by providing problems directly in CNF to Vampire, we avoid its internal clausification phase, thereby enabling complete proof reconstruction in LambdaPi [20].

We therefore focus our evaluation on Vampire version 5.0.0 and cvc5 version 1.3.0, as both support arithmetic reasoning and produce independently checkable proof certificates.

Benchmarks design. We partition our benchmark suite into three categories corresponding to the problem classes CE, PCE, and SPCE.

For CE, we use a mixed setting in which Clifford generators are used with probability 50%, while the remaining generators are of the form $\mathsf{gen}(n, m)$, with $n, m \leq 3$, and are produced randomly. While our experimental choices are intended to reflect real-world circuits, these parameters can be adjusted in the benchmark generators to increase randomness.

For each (sub-)category, we generate pairs of equivalent circuit terms and combine them with the corresponding theory file from Section 3, resulting in complete solver inputs. The complete dataset used for our experiments is also available in Zenodo [7].

Instances are parameterized by two structural parameters: the number of input wires and the initial number of circuit columns. For SPCE and PCE, both parameters range over $\{5, 10, 15, 20\}$, while for CE the number of input wires ranges over $\{2, 3, 4, 5\}$. Since transformations can arbitrarily increase or decrease the number of circuit columns during instance generation, the initial column count alone is not sufficient to characterize instance difficulty. To obtain a more robust measure of circuit size, we therefore additionally record the number of sequential (seq) and parallel (par) composition operators occurring in the generated terms. The total number of such compositions ranges from 0 to 200 and is partitioned into eight intervals of equal width (i.e., $[0, 25]$, $[25, 50]$, ...).

To ensure representative instances within each interval, we restrict term sizes to lie within a ± 5 window around the interval midpoint. For example, instances in the interval $[0, 25]$ contain between 7 and 17 composition operators. For each interval, 100 problems were generated.

Experimental setup. All experiments ran on the Grid5000 [2] cluster using an Intel Xeon Gold 5220 CPU (Cascade Lake-SP, x86_64, 2.20 GHz, 18 cores, 96 GiB RAM). For each category, we recorded the number of successfully proved equivalences. A timeout of 30 seconds was imposed for each solver run. Vampire ran with the command line `vampire -t 30s`. cvc5 ran with the command line `cvc5 --tlimit 30000 --stats`. Results are available in Figures 13 to 15.

Results overview. First of all, cvc5 outperforms Vampire across all benchmark families. It solves more instances overall and scales better as both the number of wires and the term size

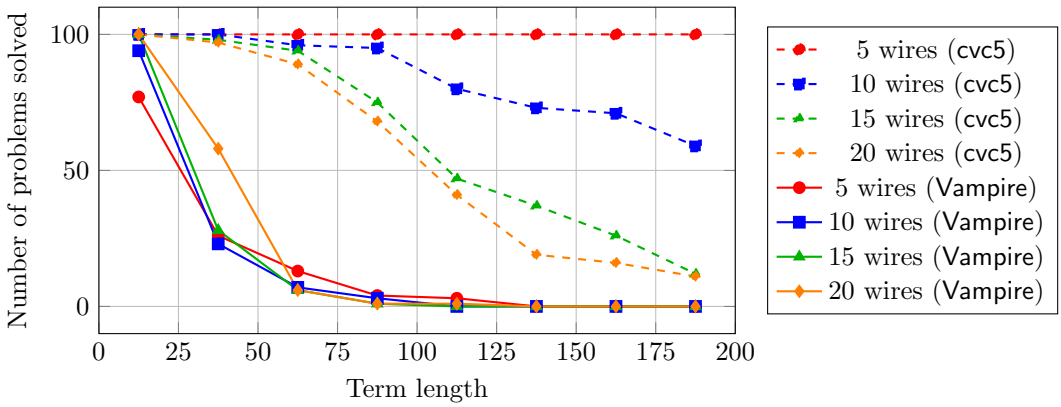


Figure 13: Results of cvc5 and Vampire on SPCE Benchmarks

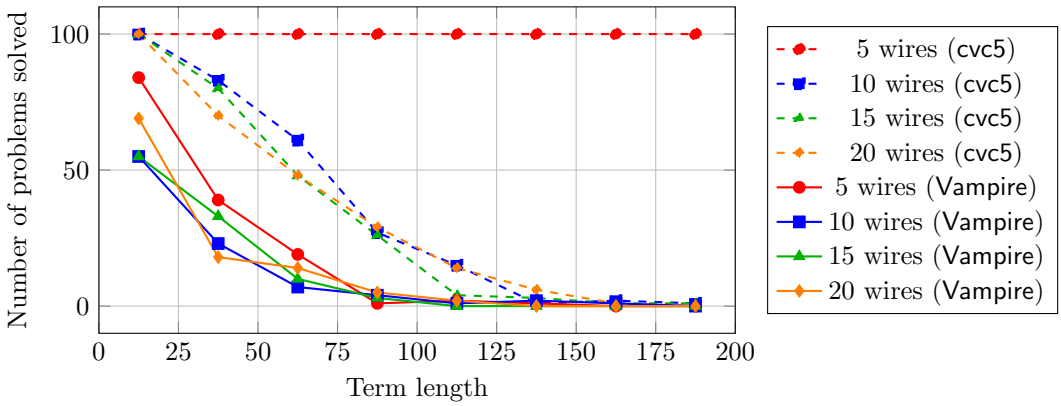


Figure 14: Results of cvc5 and Vampire on PCE Benchmarks

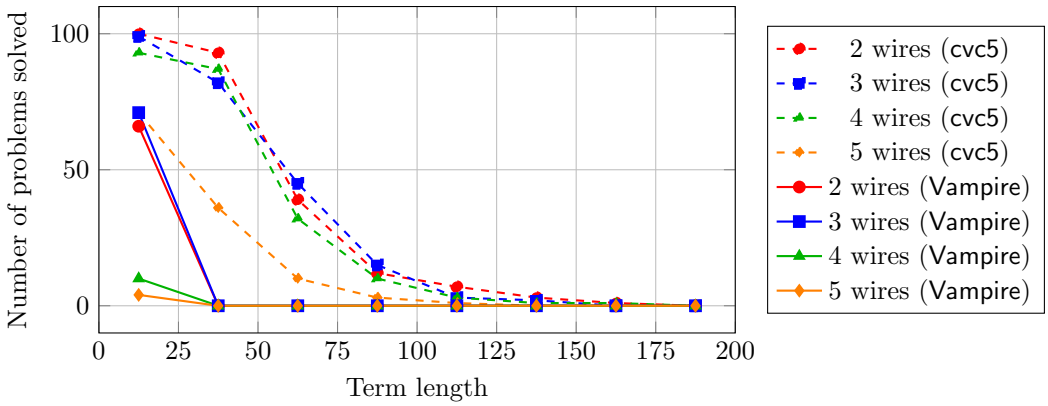


Figure 15: Results of cvc5 and Vampire on CE Benchmarks

increase. In particular, for SPCE and PCE with 5 wires, cvc5 solves all instances across the entire range of term sizes considered. This is not unexpected since SMT solvers are, by design, more suitable in general to reason with arithmetic.

Turning to structural parameters, the number of input wires and the term length have the strongest impact on the results. For cvc5 on SPCE, wire count is the dominant factor: the number of solved instances drops significantly as the number of wires increases beyond 10, regardless of term length (Figure 13). By contrast, for cvc5 on PCE and Vampire on SPCE and PCE, term length plays a comparably important role: performance degrades noticeably once term lengths exceed roughly 75–100 composition operators (Figures 13 and 14).

Finally, CE (Figure 15) is the most challenging benchmark category. We restrict experiments to instances with 2 to 5 input wires, as handling 5 wires is already challenging. Even within this restricted range, only small instances are solved. For instance, whereas cvc5 solves all SPCE and PCE instances with 5 wires, regardless of term length, it struggles on CE benchmarks once the term size exceeds roughly 100. For its part, Vampire was not able to solve any problem above the [0–25] interval, regardless of the number of wires.

To sum up, SPCE instances are largely solvable by cvc5 across a wide range of parameters, PCE instances show a performance drop as both wire count and term size increase, and CE instances are the hardest, with very few problems solved beyond small configurations.

6 Conclusion

In this work, we presented a new family of benchmarks for diagrammatic equivalence checking, motivated by the certification of quantum circuit equivalences. We introduced three variants of the problem—CE, PCE, and SPCE—and provided first-order encodings in both TPTP and SMT-LIB, together with parameterizable scripts to generate large collections of instances.

Our experiments show that diagrammatic equivalence remains challenging for current automated provers. SMT solvers such as cvc5 handle the guarded encodings more robustly than ATPs like Vampire, solving more instances across all benchmark families. Overall, the interaction between arithmetic and equational reasoning remains a central bottleneck, and both the number of input wires and the syntactic size of terms significantly impact performance, with their relative influence varying across solvers and benchmark families.

This work suggests several directions for future research. There are more ATPs with proof-production capacities than Vampire and cvc5, but most do not handle arithmetic. Thus, an arithmetic-free encoding that correctly captures the guards would be welcome. It could also be interesting to perform further experiments varying the benchmark generation parameters (for instance, by restricting the applicable rules, or by restricting the circuits to Clifford generators only). This would enable fine-grained analyses, e.g., investigating whether some coherence rules are potential bottlenecks or if a phase transition exists for this kind of problem. This in turn could provide a principled way to generate harder benchmarks.

We hope that this benchmark suite will serve as a stress test for existing solvers and that, along with the generation scripts, they will help with the development of new techniques for reasoning about string diagrams and even quantum circuits.

References

[1] Babai, L.: Graph isomorphism in quasipolynomial time. In: Proceedings of the forty-eighth annual ACM symposium on Theory of Computing. pp. 684–697 (2016)

[2] Balouek, D., Carpen Amarie, A., Charrier, G., Desprez, F., Jeannot, E., Jeanvoine, E., Lèbre, A., Margery, D., Niclausse, N., Nussbaum, L., Richard, O., Pérez, C., Quesnel, F., Rohr, C., Sarzyniec, L.: Adding virtualization capabilities to the Grid'5000 testbed. In: Ivanov, I.I., van Sinderen, M., Leymann, F., Shan, T. (eds.) Cloud Computing and Services Science, Communications in Computer and Information Science, vol. 367, pp. 3–20. Springer International Publishing (2013). https://doi.org/10.1007/978-3-319-04519-1_1

[3] Barbosa, H., Barrett, C.W., Brain, M., Kremer, G., Lachnitt, H., Mann, M., Mohamed, A., Mohamed, M., Niemetz, A., Nötzli, A., Ozdemir, A., Preiner, M., Reynolds, A., Sheng, Y., Tinelli, C., Zohar, Y.: cvc5: A versatile and industrial-strength SMT solver. In: Fisman, D., Rosu, G. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I. Lecture Notes in Computer Science, vol. 13243, pp. 415–442. Springer (2022). https://doi.org/10.1007/978-3-030-99524-9_24, https://doi.org/10.1007/978-3-030-99524-9_24

[4] Barrett, C., Stump, A., Tinelli, C.: The SMT-LIB Standard: Version 2.0. In: Gupta, A., Kroening, D. (eds.) Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, UK) (2010)

[5] Bártek, F., Bhayat, A., Coutelier, R., Hajdu, M., Hetzenberger, M., Hozzová, P., Kovács, L., Rath, J., Rawson, M., Reger, G., et al.: The vampire diary. In: International Conference on Computer Aided Verification. pp. 57–71. Springer (2025)

[6] Cailler, J., Delorme, N., Tourret, S., Perdrix, S.: Towards term-based verification of diagrammatic equivalence. arXiv preprint (2026), https://arxiv.org/abs/2602.11035

[7] Cailler, J., Delorme, N., Tourret, S.: Benchmarks for diagrammatic equivalence in tptp and smt-lib (feb 2026). https://doi.org/10.5281/zenodo.18623262

[8] Cailler, J., Delorme, N., Tourret, S.: Encoding files and problem generators for diagrammatic equivalence in tptp and smt-lib (feb 2026). https://doi.org/10.5281/zenodo.18593757

[9] Clément, A., Delorme, N., Perdrix, S.: Minimal equational theories for quantum circuits. In: Proceedings of the 39th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS) (2024). https://doi.org/10.1145/3661814.3662088

[10] Clément, A., Delorme, N., Perdrix, S., Vilmart, R.: Quantum circuit completeness: Extensions and simplifications. In: Proceedings of the 32nd EACSL Annual Conference on Computer Science Logic (CSL) (2024). https://doi.org/10.4230/LIPIcs.CSL.2024.20

[11] Clément, A., Heurtel, N., Mansfield, S., Perdrix, S., Valiron, B.: A complete equational theory for quantum circuits. In: Proceedings of the 38th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS) (2023). https://doi.org/10.1109/LICS56636.2023.10175801

[12] Coltellacci, A., Dowek, G., Merz, S.: Reconstruction of SMT proofs with lambdapi. In: 22nd International Workshop on Satisfiability Modulo Theories (SMT 2024). vol. 3725, pp. 13–23 (2024)

[13] De Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: International conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 337–340. Springer (2008)

[14] Delorme, N., Perdrix, S.: Diagrammatic reasoning with control as a constructor, applications to quantum circuits. In: Proceedings of the 29th International Conference on Foundations of Software Science and Computation Structures (FoSSaCS) (2026). https://doi.org/10.1007/978-3-032-22730-0_12

[15] Dunne, C., Burel, G.: Automatically translating proof systems for SMT solvers to the $\lambda\pi$-calculus (2025)

[16] European Organization For Nuclear Research, OpenAIRE: Zenodo (2013).

https://doi.org/10.25495/7GXK-RD71, https://www.zenodo.org/

[17] Gottesman, D.: Theory of fault-tolerant quantum computation. Physical Review A **57**(1), 127 (1998)

[18] Hondet, G., Blanqui, F.: The new rewriting engine of dedukti (system description). vol. 167, pp. 35:1–35:16. Schloss Dagstuhl – Leibniz-Zentrum für Informatik (2020). https://doi.org/10.4230/LIPICS.FSCD.2020.35, https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.FSCD.2020.35

[19] Joyal, A., Street, R.: Braided tensor categories. Advances in Mathematics **102**(1), 20–78 (1993). https://doi.org/10.1006/aima.1993.1055

[20] Komel, A.P., Rawson, M., Suda, M.: Case study: Verified vampire proofs in the lambdapi-calculus modulo. arXiv preprint arXiv:2503.15541 (2025)

[21] Lachnitt, H., Fleury, M., Barbosa, H., Jakpor, J., Andreotti, B., Reynolds, A., Schurr, H.J., Barrett, C., Tinelli, C.: Improving the SMT proof reconstruction pipeline in isabelle/hol. In: 16th International Conference on Interactive Theorem Proving (ITP 2025). pp. 26–1. Schloss Dagstuhl–Leibniz-Zentrum für Informatik (2025)

[22] Rawson, M., Voronkov, A., Schoisswohl, J., Komel, A.P.: Ground truth: Checking vampire proofs via satisfiability modulo theories. In: International Conference on Automated Deduction. pp. 136–149 (2025)

[23] Schulz, S., Cruanes, S., Vukmirovic, P.: Faster, higher, stronger: E 2.3. In: International Conference on Automated Deduction. Lecture Notes in Computer Science, vol. 11716, pp. 495–507. Springer (2019)

[24] Schurr, H.J., Fleury, M., Barbosa, H., Fontaine, P.: Alethe: Towards a generic SMT proof format (extended abstract). Electronic Proceedings in Theoretical Computer Science **336**, 49–54 (jul 2021). https://doi.org/10.4204/eptcs.336.6, http://dx.doi.org/10.4204/EPTCS.336.6

[25] Selinger, P.: A Survey of Graphical Languages for Monoidal Categories, pp. 289–355. Springer Berlin Heidelberg, Berlin, Heidelberg (2011). https://doi.org/10.1007/978-3-642-12821-9

[26] Smallbone, N.: Twee: An equational theorem prover. In: International Conference on Automated Deduction. pp. 602–613 (2021)

[27] Stump, A., Oe, D., Reynolds, A., Hadarean, L., Tinelli, C.: SMT proof checking using a logical framework. Formal Methods in System Design **42**(1), 91–118 (2013)

[28] Sutcliffe, G.: The Logic Languages of the TPTP World. Logic Journal of the IGPL (2022). https://doi.org/10.1093/jigpal/jzac068

[29] Zhang, Y., Wang, Y.R., Flatt, O., Cao, D., Zucker, P., Rosenthal, E., Tatlock, Z., Willsey, M.: Better together: Unifying datalog and equality saturation. Proceedings of the ACM on Programming Languages **7**(PLDI), 468–492 (2023)