Nondeterminism subject to output commitment in combinatorial filters

Yulin Zhang[1] and Dylan A. Shell[2] ⋆

[1] The University of Texas at Austin, Austin TX 78712, USA
[2] Texas A&M University, College Station TX 77843, USA
yulin@cs.utexas.edu   dshell@tamu.edu

Abstract. We study a class of filters—discrete finite-state transition systems employed as incremental stream transducers—that have application to robotics: e.g., to model combinatorial estimators and also as concise encodings of feedback plans/policies. The present paper examines their minimization problem under some new assumptions. Compared to strictly deterministic filters, allowing nondeterminism supplies opportunities for compression via re-use of states. But this paper suggests that the classic automata-theoretic concept of nondeterminism, though it affords said opportunities for reduction in state complexity, is problematic in many robotics settings. Instead, we argue for a new constrained type of nondeterminism that preserves input–output behavior for circumstances when, as for robots, causation forbids 'rewinding' of the world. We identify problem instances where compression under this constrained form of nondeterminism results in improvements over all deterministic filters. In this new setting, we examine computational complexity questions for the problem of reducing the state complexity of some given input filter. A hardness result for general deterministic input filters is presented, as well as for checking specific, narrower requirements, and some special cases. These results show that this class of nondeterminism gives problems of the same complexity class as classical nondeterminism, and the narrower questions help give a more nuanced understanding of the source of this complexity.

Keywords: Discrete filters · State space reduction · Nondeterminism

1   Introduction

Going right back to Rabin and Scott's seminal paper [4], classic automata theory has considered a particular type of nondeterminism that declares a string accepted if *some* tracing of it reaches a final state. Under this definition, as that paper first established, such nondeterministic finite automata express the same languages as their deterministic brethren, *viz.* precisely the regular ones. But it is well known that nondeterminism can, nevertheless, confer practical benefits: nondeterminism permits expression of some languages with far greater concision.

These facts have bearing on a line of robotics research looking at finite-state encodings of combinatorial estimators, discrete feedback plans and stateful policies. Employing the terminology of Tovar et al. [11], we shall refer to discrete transition systems that process a stream of observations as 'filters'. In their paper, they demonstrate how ascertaining the amount of information required to answer particular queries may involve surprising subtlety. Among their illustrative examples of filters, their most elegant instances track all that is needed via curiously few states. More generally, as representations, discrete filters can help direct attention to considerations of *minimalism* [1], drawing into sharp focus questions of necessity rather than of mere sufficiency. Beside elucidating interesting structure —a consideration of obvious scientific value— they also have practical application in building simpler, cheaper devices.

A series of papers has explored filter compression: proposing algorithms for reducing the number of states needed [13,6,12], and examining the hardness of achieving the minimal filter under differing assumptions [3,8,14,9]. An initial suggestion that the problem might simply be identical to automata minimization was shown to be false [3]. And thoughts that filter minimization could be accomplished by quotienting under a bisimulation relation turns out also to be false [5] and, indeed, no equivalence relation will do [13]. Even though automata theory suggests that nondeterminism may afford opportunities for added compression, no algorithm for compressing filters currently exploits nondeterminism.

Insofar as nondeterminism does figure in prior work on filters, there are two forms. The first is *tracing nondeterminism* wherein any vertex may have multiple departing arcs that match some symbol being processed. This type of nondeterminism corresponds with the concept in classical automata, and the informal intuitions that are usual there, apply here as well. So, when there are two edges that match, we might speak of 'taking both'; or, upon reaching such a juncture, we might pick one but later change our mind and rewind to choose another. The imaginary processes that these two narrations provide as interpretations, despite being distinct, agree in terms of the language they characterize.

In filters there is a second form, *output nondeterminism*, where any vertex may bear multiple outputs. On arriving at a vertex with several outputs, any of these may be selected. Both types of filter nondeterminism were first explored together in [7], which investigated their relationship under a model that examines how sensor imperfections lead to loss of functionality. Looking specifically at minimization of filters, output nondeterminism is formulated and studied in our paper at the previous WAFR [13]. Recently, [14] added tracing nondeterminism to the minimization picture, showing that tracing nondeterminism does allow further compression. (In other words, the 'opportunities' mentioned in the earlier paragraph *do* exist in filters.)

The present paper highlights why tracing nondeterminism may not always be appropriate in robotics applications. It then defines a new class of nondeterminism on the basis of this observation. This class represents a *juste milieu*: permitting choices in tracing while still encoding deterministic input–output behavior. We then see whether this class affords additional compression over the

strictly deterministic case (spoiler: it does). Thereafter, Sections 4–5 examine the relation of this class, in terms of hardness of minimization, to others and draws connections with results from automata theory.

2   Basic definitions

Having reached the limits of informal talk, some definitions follow next.

Definition 1 (procrustean filter [7]). *A* procrustean filter*, p-filter or* filter *for short, is a 6-tuple* $\mathcal{F} = (V, V_0, Y, \tau, C, c)$ *in which* V *is a non-empty finite set of states,* V_0 *is the set of initial states,* Y *is the set of observations,* $\tau : V \times V \to \wp(Y)$ *is the transition function,* C *is the set of outputs, and* $c : V \to \wp(C) \setminus \{\varnothing\}$ *is the output function. (Here,* $\wp(A)$ *denotes the powerset of set* A*.)*

For some $\mathcal{F}$, we will write $V(\mathcal{F})$, $V_0(\mathcal{F})$ and $Y(\mathcal{F})$, for its sets of states, initial states, and observations, respectively. We'll present filters visually as graphs, with states as vertices and transitions as directed edges with observations. We shall assume that $Y(\mathcal{F})$ is finite (it affords some simplicity and will suffice for our needs here, though cf. [7]). To be consistent with automata theory, we also call $Y(\mathcal{F})$ the alphabet of $\mathcal{F}$. The outputs will appear visually as colors at each vertex, going some way to explain the names C and $c(\cdot)$. We say that $\mathcal{F}$ is *vertex single-output*, if every vertex v in $\mathcal{F}$ has $|c(v)| = 1$. Otherwise, it is *vertex multi-output*, a term rather less vague than the phrase 'output nondeterminism' used in the preceding section.

For some filter $\mathcal{F} = (V, V_0, Y, \tau, C, c)$, an observation sequence (or a string) $s = y_1 y_2 \dots y_n \in Y^*$, and states $v, w \in V$, we say that w is *reached by* s (or s *reaches* w) when traced from v, if some sequence of states $w_0, w_1, \dots, w_n$ in $\mathcal{F}$, such that $w_0 = v$, $w_n = w$, and $\forall i \in \{1, 2, \dots, n\}, y_i \in \tau(w_{i-1}, w_i)$. The set of all states reached by s from a state v in $\mathcal{F}$ will be denoted $\mathcal{V}_{\mathcal{F}}(v, s)$, and we will use $\mathcal{V}_{\mathcal{F}}(s)$ for all states reached by s from any initial state of the filter, i.e., $\mathcal{V}_{\mathcal{F}}(s) = \bigcup_{v_0 \in V_0} \mathcal{V}_{\mathcal{F}}(v_0, s)$. If $\mathcal{V}_{\mathcal{F}}(v, s)$ is empty, then we say that string s *crashes* in $\mathcal{F}$ starting from v. Otherwise, we say that s is an *extension* of v in $\mathcal{F}$. The set of all extensions of v in $\mathcal{F}$ is denoted as $\mathcal{L}_{\mathcal{F}}(v) = \{s \in Y^* | \mathcal{V}_{\mathcal{F}}(v, s) \neq \varnothing\}$. Specifically, the set of all strings that are extensions of any initial state in $\mathcal{F}$ is called the *interaction language* (or, briefly, just *language*) of $\mathcal{F}$, and is written as $\mathcal{L}(\mathcal{F}) = \cup_{v_0 \in V_0(\mathcal{F})} \mathcal{L}_{\mathcal{F}}(v_0)$. Contrariwise, the set of strings reaching w from some initial state in $\mathcal{F}$ is denoted as $\mathcal{S}_w^{\mathcal{F}} = \{s \in Y^* | w \in \mathcal{V}_{\mathcal{F}}(s)\}$. Without loss of generality and to help dispose of turgid statements of conditions, we shall assume that every state in $\mathcal{F}$ can be reached by some string from an initial state. Otherwise, we may remove this state from $\mathcal{F}$ with no impact on the language of $\mathcal{F}$ or outputs for strings in the language.

Although quite standard, the following is paramount, so justifies emphasis:

Definition 2 (tracing-deterministic). *A filter* $\mathcal{F} = (V, V_0, Y, \tau, C, c)$ *is* tracing-deterministic *or* state-determined*, if* $|V_0| = 1$*, and for every* $v_1, v_2, v_3 \in V$ *with* $v_2 \neq v_3$*,* $\tau(v_1, v_2) \cap \tau(v_1, v_3) = \varnothing$*.*

We shall say that a filter which is not tracing-deterministic is tracing-non-deterministic. Examples of tracing-deterministic and tracing-nondeterministic filters appear in Figure 1b and 1a.

Next, we define the crucial concept of functional substitutability, giving conditions when one filter may serve as a replacement for another.

Definition 3 (output simulating [3]). *Let $\mathcal{F}$ and $\mathcal{F}'$ be two filters, then $\mathcal{F}'$ output simulates $\mathcal{F}$ if (1). $\mathcal{L}(\mathcal{F}) \subseteq \mathcal{L}(\mathcal{F}')$ and (2). $\mathcal{C}(\mathcal{F}', s) \subseteq \mathcal{C}(\mathcal{F}, s)$.*

For convenience, we refer to property (1) and (2) as *language inclusion* and *output compatibility*, respectively. The intuition is that language inclusion ensures that $\mathcal{F}'$ is able to process any input that $\mathcal{F}$ can. When the output that $\mathcal{F}'$ produces could be some output produced by $\mathcal{F}$, then it is considered compatible.

The three vertex single-output filters in Figure 1 provide a basic feel for the concept; shortly we shall consider vertex multi-output instances too.

The core optimization question is one of reducing state complexity in a filter:

Problem: Filter Minimization (FM)

Input: A filter $\mathcal{F}$.
Output: A filter $\mathcal{F}^\dagger$ with fewest states, such that $\mathcal{F}^\dagger$ output simulates $\mathcal{F}$.

The decision version of FM provides some $k \in \mathbb{N}$ and asks if any $\mathcal{F}^\dagger$ with no more than k states can output simulate $\mathcal{F}$. We shall consider variations on this problem under different constraints (to be detailed in Section 3.3).

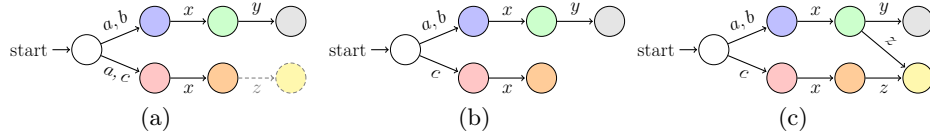(a)   (b)   (c)

Fig. 1: Three simple vertex single-output filters: (a) a tracing-nondeterministic filter; (b) a tracing-deterministic one that output simulates the filter in (a) when the dashed z-edge and yellow-state are absent; and (c) another one that output simulates (a), now with the dashed edge/state, or without, and also (b) as well.

3   Some examples leading to our key definition

Starting from an important problem instance, this section builds up to Definition 4, the central definition of the paper.

3.1   An interesting filter and two of its minimizers

Consider Figure 2. It shows an example of a tracing-nondeterministic and vertex multi-output filter, $\mathcal{F}_{\text{inp}}$. It has the form of a tree, and hence has a finite language. When processing strings beginning with $\{a, b, c, d, e\}$, the tracing leads to what we'll term the lower-half; strings starting with $\{f, g, \ldots, \ell\}$ lead to the upper-half.
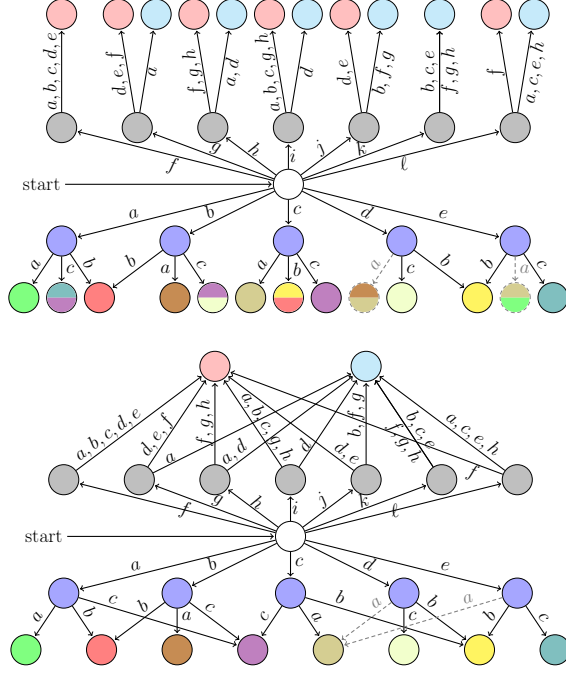
Fig. 2: A filter $\mathcal{F}_{\text{inp}}$ with $Y(\mathcal{F}_{\text{inp}}) = \{a, b, c, \ldots, k, \ell\}$ that is tracing-deterministic and vertex multi-output. Tracing string 'ac' may produce either **purple** or **teal** as an output. (The dashed a-edges/vertices should be ignored at first; they will be introduced when the discussion re-visits the example.)

Fig. 3: A filter $\mathcal{F}_{\text{det}}$ that is tracing-deterministic and vertex single-output. It output simulates $\mathcal{F}_{\text{inp}}$ and is a solution to the FM problem in the sense that no other tracing-deterministic filter with fewer vertices output simulates $\mathcal{F}_{\text{inp}}$. The reduction is from $|V(\mathcal{F}_{\text{inp}})| = 36$ to $|V(\mathcal{F}_{\text{det}})| = 23$.

Figure 3 gives a tracing-deterministic vertex single-output filter, $\mathcal{F}_{\text{det}}$, that output simulates $\mathcal{F}_{\text{inp}}$. In the upper-half, all the pink states have been merged together, and the same has occurred with the light-blue ones. For the lower-half, single colors have been chosen for the multi-output vertices (picking purple and yellow) and then possible mergers made. In fact, $\mathcal{F}_{\text{det}}$ is a minimal tracing-deterministic filter that output simulates $\mathcal{F}_{\text{inp}}$:

Lemma 1. *The $\mathcal{F}_{\text{det}}$ of Figure 3 is a minimal tracing-deterministic filter that output simulates $\mathcal{F}_{\text{inp}}$.*

Proof. The proof appears in the supplementary material.   □

Now examine Figure 4, which gives $\mathcal{F}_{\text{nd}}$, a filter that output simulates $\mathcal{F}_{\text{inp}}$ and is smaller than $\mathcal{F}_{\text{det}}$. It is a tracing-<u>non</u>deterministic vertex single-output filter. It was constructed as follows. In the upper-half, pink and light-blue extensions of strings with final symbol $\{a, b, \ldots, h\}$ have been separated and reconstituted in only 6 gray vertices. The lower-half has exploited the fact that some strings have multiple valid outputs (like 'ac' giving teal or purple); using this freedom allows sharing of some vertices. And this $\mathcal{F}_{\text{nd}}$ is the minimal tracing-nondeterministic filter that output simulates $\mathcal{F}_{\text{inp}}$.

Lemma 2. *$\mathcal{F}_{\text{nd}}$ is a minimal tracing-nondeterministic filter that output simulates $\mathcal{F}_{\text{inp}}$.*

Proof. The proof appears in the supplementary material.   □

Fig. 4: A filter $\mathcal{F}_{\mathrm{nd}}$ that is tracing-nondeterministic and vertex single-output, and which output simulates $\mathcal{F}_{\mathrm{inp}}$. It solves Problem FM in the sense that no other filter with fewer vertices output simulates $\mathcal{F}_{\mathrm{inp}}$. $|V(\mathcal{F}_{\mathrm{nd}})| = 21$.
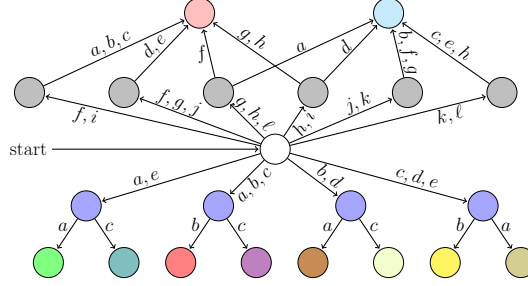
3.2   Processing inputs incrementally and string single-output filters

Thus far, filters have been discussed almost as though they were entirely abstract objects. When employed in practice, their output (an element from set C, which we've shown visually through colors) is grounded to some specific meaning. It may have an interpretation as a sufficient statistic for some estimation task, or as an answer to a particular query, or it may be an action to have the robot execute. Figure 5 provides an example of a feedback plan for a manipulation problem (re-drawn from [3], itself inspired from [10] and work in this line). The outputs describe how the system should operate: orange and green map to actions to execute (specific motions for a gripper), and white indicates that the system should terminate as the task has been completed successfully.

We envision a controller for the robot being represented internally via a encoding of this filter, which is then used to process sensor observations and generate actions. It does this incrementally. For instance, in Figure 5, the controller simply has to track the current vertex to know what to have the robot do. Minimizing the filter has the advantage, then, of reducing the controller's memory footprint. So now consider what happens when, in light the previous examples of superior reduction via nondeterminism, the minimized filter exhibits some nondeterminism. Suppose something simple like the 6-state filter in Figure 1b is obtained as a result.

For the filter in Figure 1b, when the robot obtains, first, an 'a', it has a choice from the initial white vertex. It might make the arbitrary decision to take the lower-branch. If the sensor provides it an 'x' next, it
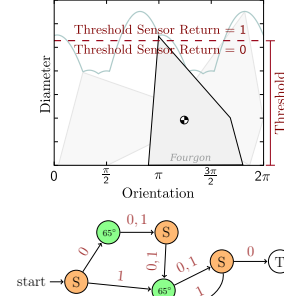

Fig. 5: A small feedback plan for a planar manipulation problem expessesd as a filter. Using 0 or 1 readings from a threshold sensor, it orients the 4-sided part shown via a sequence of squeeze-gripper and rotate-by-65° actions. (Based on [3, Figs. 14 and 15].)

would proceed to the orange state. Then, if a 'y' is obtained (possible, since 'axy' is in the input language and so some output should be produced) it has then become clear that the initial choice could've been better made if the upper-branch were taken. (And introducing the dashed z-edge and yellow-state shows that, though in the basic setup the upper-branch has a superset of the strings of the

lower-branch, there needn't be a choice that covers the other options.) In order to resolve this missing y, the filter effectively needs to rewind back to the 'a' and take a different edge. Such an approach is undesirable because it requires the controller store the (unbounded) history of readings, and the filter is no longer operating in an efficient incremental manner.

One solution is to trace forward on all matching edges: so in Figure 1b, after 'a', some markers keep the position on both upper- and lower-branches. Then, when a 'y' arrives, though one trace dies off, the second can proceed. This requires a number of markers, but they are bounded in the size of the filter. One question, when one has multiple vertices that are currently reached (so both blue and red after 'a' in Figure 1b), is which output should be chosen? This motivates the following definition, wherein we are spared the burden of making a choice:

Definition 4 (string single-output). *A filter* $\mathcal{F} = (V, V_0, Y, \tau, C, c)$ *is* string single-output *if* $\forall s \in \mathcal{L}(\mathcal{F})$, $|\mathcal{C}(\mathcal{F}, s)| = 1$; *Otherwise, call it* string multi-output.

Immediate remark: Any tracing-deterministic vertex single-output filter will be a string single-output filter.

In the discussion earlier, we made the case that the rewinding interpretation for tracing-nondeterminism is undesirable from a practical point of view. But notice how Definition 4 comes into play also under that interpretation: for a general tracing-nondeterministic filter, when a string is rewound and traced forward anew, it might generate a different sequence of output colors. Returning to our concrete example in Figure 1b, following the symbol 'y' being encountered on the lower-branch, after rewinding and tracing 'a' forward a second time on the upper-branch now, the color blue is associated with the string. If this filter is encoding a plan, then blue corresponds to an action; but it is not appropriate to generate that action—that represents a point in the past—as the action for red was already executed, with sensor reading 'x' being produced thereafter. Had both branches produced the same outputs (that is, been string single-output), then even if the robot decided to store the sequence of input symbols and rewind to process the string forward, avoiding crashing, and reaching a vertex from which to resume tracing at the newest symbol, this would be a purely internal affair. The actions generated in the world would be consistent with the ones actually executed earlier in time.

Is the class of string single-output filters useful? We can answer in the affirmative. Suppose one wishes to minimize a filter, but maintain the sort of temporal/causal consistency just described. Then one may seek a string single-output minimizer and, indeed, there exist such filters that are smaller than any deterministic filter. Returning to example filter $\mathcal{F}_{\text{inp}}$ of Figure 2, now examine $\mathcal{F}_{\text{sso}}$ presented in Figure 6. It is tracing-nondeterministic but is string single-output, and it has one fewer vertex than $\mathcal{F}_{\text{det}}$. Further, this particular filter $\mathcal{F}_{\text{sso}}$ is a string single-output minimizer of $\mathcal{F}_{\text{inp}}$.

Lemma 3. $\mathcal{F}_{\text{sso}}$ *is a string single-output minimizer of* $\mathcal{F}_{\text{inp}}$.

Proof. Proof appears in the supplementary material.   □
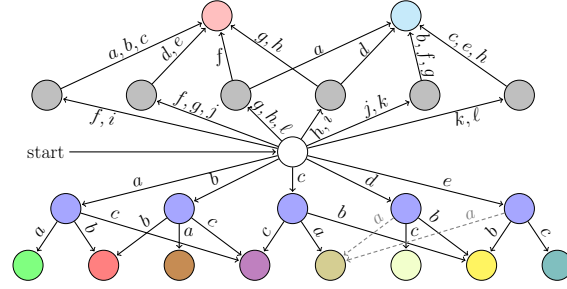
Fig. 6: A filter $\mathcal{F}_{\mathrm{sso}}$ that is tracing-nondeterministic and vertex single-output, and which output simulates $\mathcal{F}_{\mathrm{inp}}$. It solves FM in the sense that no other string single-output filter with fewer vertices output simulates $\mathcal{F}_{\mathrm{inp}}$. $|V(\mathcal{F}_{\mathrm{sso}})| = 22$.

One additional point worth noting is that $\mathcal{F}_{\mathrm{nd}}$ has a language which is larger than that of $\mathcal{F}_{\mathrm{inp}}$. Both $\mathcal{F}_{\mathrm{det}}$ and $\mathcal{F}_{\mathrm{sso}}$ match the input language exactly. Specifically, $\mathcal{F}_{\mathrm{nd}}$ will process strings 'da' and 'ea', and assign them outputs. These show up as a sort of aliasing in the compression down to 4 royal-blue vertices. It would be erroneous to believe that this is necessary for the gap in minimizer sizes. If one adds the dashed gray a-edges to Figures 2, 3, and 6, then all have precisely the same language.

3.3   Classes of minimizer

The three filters $\mathcal{F}_{\mathrm{det}}$, $\mathcal{F}_{\mathrm{sso}}$, and $\mathcal{F}_{\mathrm{nd}}$ are all minimizers of $\mathcal{F}_{\mathrm{inp}}$ within their respective classes. The class of tracing-nondeterministic filters, by definition, omits the constraint that forces determinism, and thus contains all the tracing-deterministic ones as well. As already remarked, the class of string single-output filters includes all the tracing-deterministic ones. And tracing-nondeterministic filters may well violate the string single-output constraint. So, as solutions to the problem of minimization, tracing-deterministic ones can be no smaller than tracing-nondeterministic or string single-output ones; and string single-output ones may be no smaller than tracing-nondeterministic ones. Of course, for a particular problem, there *might* be a size gap the other way, but there needn't always be. (Consider, for instance, Figure 1b; it is its own nondeterministic minimizer, but is also the deterministic minimizer as well.) The three $\mathcal{F}_{\mathrm{det}}$, $\mathcal{F}_{\mathrm{sso}}$, and $\mathcal{F}_{\mathrm{nd}}$ show a separation in their sizes for the single input of $\mathcal{F}_{\mathrm{inp}}$.

This motivating discussion now concluded, we will always be explicit in what follows about the class of minimizer sought when studying a variant of FM. Further, we will also clarify in assumptions on the input filter. We shall abbreviate:

FM(DF→DF): where both the input and output filter are tracing-deterministic (DF), e.g. $\mathcal{F}_{\mathrm{det}}$.

FM(DF→SSO): where the input filter is tracing-deterministic (DF), and the output filter must be string single-output (SSO), e.g. $\mathcal{F}_{\mathrm{sso}}$ or $\mathcal{F}_{\mathrm{sso}}$ plus dashed a-edges.

FM(DF→SMO): where the input filter is tracing-deterministic (DF), and the output filter can be tracing-nondeterministic (SMO), e.g. $\mathcal{F}_{\mathrm{nd}}$.

3.4   Aside: single- and multi-output vertices

One observation is that $\mathcal{F}_{\mathrm{det}}$ and $\mathcal{F}_{\mathrm{nd}}$ are vertex single-output filters. Indeed, no minimizer need have ever multi-output vertices itself.

Lemma 4. *If $\mathcal{F}_m$ is a vertex multi-output filter that output simulates $\mathcal{F}$, then there exists some $\mathcal{F}_s$ as an output simulator of $\mathcal{F}$, where $\mathcal{F}_s$ is vertex single-output and, moreover, $|V(\mathcal{F}_m)| = |V(\mathcal{F}_s)|$.*

Proof. Any vertex in $\mathcal{F}_m$ with multiple outputs can have a single one selected arbitrarily to yield $\mathcal{F}_s$. Then $\mathcal{F}_s$ output simulates $\mathcal{F}_m$, and hence $\mathcal{F}$ too.   □

Multi-output vertices allow expression of some flexibility in the Y to C mapping. As already apparent in Figure 2 in the lead-up to Figures 3, 4, and 6, multi-output vertices can be helpful for expressing some input to a minimization problem, specifically in providing a specification which grants a degree of freedom. (Both [13] and [12], in their respective introductory sections, have detailed robotics scenarios in which there is a degree of freedom which is natural, and is cleanly expressed via multi-output vertices.)

The preceding helps emphasize that one use of filters is as specifications of the appropriate range of system responses for a given input. This is a distinct use from their adoption directly as the object that computes outputs, incrementally, from inputs as and when they arrive.

3.5   The role of output simulation and consistency across sequences

Let's revisit the topics arising in the discussion directly following Definition 4, but now from a point of view that puts aside considerations of specific edge/vertex structure. Any filter $\mathcal{F}$ that is string single-output, when interpreted as an input–output map on sequences, i.e., from $\mathcal{L}(\mathcal{F}) \subseteq Y^*$ to C^*, is deterministic. That is, it is a function. Though the filter may be tracing-nondeterministic, any of those branching choices are purely internal to it, and any observer treating it as a black-box is shielded from those decisions.

For some given filter $\mathcal{G}$, its string single-output minimizer must produce, for each string, some output compatible with that produced on $\mathcal{G}$. Language inclusion ensures that the minimizer can process every string that filter $\mathcal{G}$ can, and all filters process a prefix-closed collection of strings. But it is important to note that Definition 3 does not place any additional requirement on the relationship between the outputs produced for a string and those produced by its prefixes. Or, when thinking of the filter as a function on sequences: if sequence s maps to t, then a subsequence of s need not map to a subsequence of t. Put differently: it is a mapping from $\mathcal{L}(\mathcal{F}) \subseteq Y^*$ to C not to C^*, and when one naturally uses the prefix closure property to interpret it as lifted to sequences of C, something extra appears, seemingly. This *extra* is not in the requirements for output simulation.

Specifically, no 'tracing correlations' need be preserved. To make this concrete, see in Figure 1a how on string 'ax' only blue followed by green, or red followed by orange, can be produced. Output simulation permits any filter that produces either red or blue on 'a', and green or orange on 'ax'. This freedom (four cases rather than only two) may be necessary for behavior to be implementable incrementally. Using our example: after adding in the z-edge/yellow-vertex, Figure 1c gives a tracing-deterministic output simulator that does not preserve the

red–orange–yellow output ordering; but no tracing-deterministic output simulator exists that can preserve the output orderings produced by both 'axy' and 'axz' strings.

In the very next section, this will not be a concern for a much simpler reason: when we consider the hardness of forms of minimization to string single-output filters compared to other cases (i.e., the FM(DF→SSO) problem *vs.* FM(DF→DF) and FM(DF→SMO)), we shall assume we are given a tracing-deterministic input. (Without this restriction, minimization can only become harder—a fact we use in Section 5) And the process of converting some general filter into a tracing-deterministic input (e.g. [7, Algorithm 2]) removes the tracing dependent output orderings.

4   Hardness of FM(DF→DF), FM(DF→SSO), and FM(DF→SMO)

Having seen the gap between the minimizers of FM(DF→DF), FM(DF→SSO) and FM(DF→SMO), we now examine the computational complexity of minimization.

4.1   Complexity of FM(DF→DF), FM(DF→SSO), and FM(DF→SMO)

In this section, we will give worst-case complexity analyses for the 3 problems.

Prior work has proved the hardness results for FM(DF→DF) by reducing from a graph coloring problem:

Theorem 1 (Theorem 9 [13]). FM(DF→DF) *is NP-complete.*

Next, we will leverage the existing results to show that both FM(DF→SMO) and FM(DF→SSO) are PSPACE-complete.

Prior results in tracing-nondeterministic filter minimization show that to check whether one tracing-nondeterministic filter output simulates another is in PSPACE.

Lemma 5 (Lemma 7 [14]). *Given a tracing-nondeterministic filter $\mathcal{F}$ and a tracing-nondeterministic filter $\mathcal{F}'$, it is in PSPACE to check whether $\mathcal{F}'$ output simulates $\mathcal{F}$.*

In both FM(DF→SMO) and FM(DF→SSO), the input filter is tracing-deterministic, which is a special case in Lemma 5. Hence, it is also PSPACE to check output simulation for FM(DF→SMO) and FM(DF→SSO). Accordingly, we have the PSPACE results for FM(DF→SMO) immediately:

Lemma 6 (FM(DF→SMO) is in PSPACE). *Given a tracing-deterministic filter $\mathcal{F}$ and a tracing-nondeterministic filter $\mathcal{F}'$, it is in PSPACE to check whether $\mathcal{F}'$ output simulates $\mathcal{F}$.*

But for FM(DF→SSO), we additionally need to check whether the output filter is string single-output, and are required to show whether this procedure is in PSPACE as well.

To facilitate the proof for FM(DF→SSO), we use the following graph product:

Definition 5 (product graph). *Given filters $\mathcal{F}$ and $\mathcal{F}'$, all the strings that are in both $\mathcal{F}$ and $\mathcal{F}'$ are produced by their tensor product graph $\mathcal{G}$, denoted as $\mathcal{G} = \mathcal{F} \otimes \mathcal{F}'$. $\mathcal{G}$ has initial states $V_0(\mathcal{F}) \times V_0(\mathcal{F}')$, and for every string $s \in \mathcal{L}(\mathcal{F}) \cap \mathcal{L}(\mathcal{F}')$, $(v, v') \in \mathcal{V}(\mathcal{G}, s)$.*

Using the above operator, we show that FM(DF→SSO) is also in PSPACE:

Lemma 7 (FM(DF→SSO) is in PSPACE). *Given a determinsitic filter $\mathcal{F}$ and an non-determinsitic filter $\mathcal{F}'$, it is in PSPACE to check whether $\mathcal{F}'$ is string single-output, and $\mathcal{F}'$ output simulates $\mathcal{F}$ or not is in PSPACE.*

Proof. Following Lemma 5, it is in PSPACE to check whether the output simulating condition holds for FM(DF→SSO). Next, we will show that it also takes polynomial space to check whether $\mathcal{F}'$ is string single-output or not. First, if there is a state v' in $\mathcal{F}'$ that has more than a single output, i.e., $|c(v')| > 1$, then the strings in $\mathcal{S}_{v'}^{\mathcal{F}'}$ have more than one output, and $\mathcal{F}'$ is not string single-output. Otherwise, we build a product with itself $\mathcal{G} = \mathcal{F}' \otimes \mathcal{F}'$ to vet states that are reached by the same string. According to the construction, there are at most $|V(\mathcal{F}')|^2$ states in $\mathcal{G}$. Additionally, every pair of states that are nondeterministically reached by the same string in $\mathcal{F}'$ will appear as a state in $\mathcal{G}$. And every state in $\mathcal{G}$ consists of two states that are reached by some string in $\mathcal{F}'$. Hence, the states in $\mathcal{G}$ capture all pairs of states that are non-deterministically reached. For every string s in $\mathcal{F}'$, let $V_s' = \mathcal{V}(\mathcal{F}', s)$ be the set of states non-deterministically reached by s. Then we know that s has precisely one output if and only if every pair of states in V_s' share the same output mutually. Therefore, we can say $\mathcal{F}'$ is string single-output, if for every state (v_i', v_j') in $\mathcal{G}$, $c(v_i') = c(v_j')$. Otherwise, it is not. So we have a polynomial space procedure. $\square$

Next, we will examine the hardness results for both FM(DF→SSO) and FM(DF→SMO) via some other results from automata theory. Similar to a filter, an automaton $\mathcal{A}$ is defined as a tuple $(V_0, V, \Sigma, \tau, F)$, where Σ is the alphabet, F is the set of final states. Different from the filter, the language of an automaton is called the accepting language, denoted henceforth as $\mathcal{L}^A(\mathcal{A})$, which is the set of strings that reach the final states. But the difference between $\mathcal{L}^A(\cdot)$ and just $\mathcal{L}(\cdot)$ disappears when all states in the automaton are final. Automata in which $F = V$ are called ASF automata (where 'ASF' stands for 'All States Final').

A prior result for ASF automata shows that it is PSPACE-complete to check whether an ASF NFA with alphabet Σ has accepting language Σ^* or not:

Lemma 8 (NFA-NonUniversality-ASF [2]). *Given an ASF NFA $\mathcal{A}$ with alphabet Σ, if $|\Sigma| \geq 2$, it is PSPACE-complete to check whether $\mathcal{L}^A(\mathcal{A}) = \Sigma^*$ holds or not.*

Then we will show that both FM(DF→SSO) and FM(DF→SMO) are PSPACE-hard by reducing from NFA-NonUniversality-ASF.

Lemma 9 (FM(DF→SSO) and FM(DF→SMO) are PSPACE-hard). *Given a tracing-deterministic filter $\mathcal{F}$ and an string single-output tracing-nondeterministic filter $\mathcal{F}'$, if $|Y(\mathcal{F})| \geq 2$ and $|Y(\mathcal{F}')| \geq 2$, it is PSPACE-hard to check whether $\mathcal{F}'$ output simulates $\mathcal{F}$ or not.*

Proof. Proof by reduction from NFA-NONUNIVERSALITY-ASF. Given an ASF NFA $\mathcal{A}$ with alphabet Σ and $|\Sigma| \geq 2$, then treat $\mathcal{A}$ as a filter $\mathcal{F}'$ with an output function that colors every state the same color c_0. Then the interaction language of filter $\mathcal{F}'$ is the same as the accepting language of automata $\mathcal{A}$, i.e., $\mathcal{L}(\mathcal{F}') = \mathcal{L}^A(\mathcal{A})$. Next, create a tracing-deterministic filter $\mathcal{F}$, where there is only a single state with a self-loop bearing Σ. This state is colored c_0. Then $\mathcal{L}(\mathcal{F}) = \Sigma^*$. Therefore, $\mathcal{L}^A(\mathcal{A}) = \Sigma^* \iff \mathcal{L}(\mathcal{F}) \subseteq \mathcal{L}(\mathcal{F}')$. Hence, to check NFA-NONUNIVERSALITY-ASF, i.e., whether $\mathcal{L}^A(\mathcal{A}) = \Sigma^*$ holds or not, is equivalent to checking $\mathcal{L}(\mathcal{F}) \subseteq \mathcal{L}(\mathcal{F}')$. Additionally, for every string s in $\mathcal{F}$, the output from $\mathcal{F}$ is the same as that from $\mathcal{F}'$. Hence, $\mathcal{L}(\mathcal{F}) \subseteq \mathcal{L}(\mathcal{F}')$ if and only if $\mathcal{F}'$ output simulates $\mathcal{F}$. Therefore, NFA-NONUNIVERSALITY-ASF has been reduced to FM(DF→SSO) in polynomial time, and FM(DF→SSO) is PSPACE-hard. The same reduction also shows that FM(DF→SMO) is PSPACE-hard.   □

Note that this reduction requires the input to have a non-unitary alphabet, so as to be general enough to model the NFA-NONUNIVERSALITY-ASF problem.

Theorem 2. *Both* FM(DF→SSO) *and* FM(DF→SMO) *are* PSPACE-*complete.*

Proof. Combine Lemma 7 and Lemma 9.   □

4.2   Minimization problems with a unitary alphabet

Lemma 9 indicates that when the alphabet comprises 2 or more symbols, both FM(DF→SSO) and FM(DF→SMO) are PSPACE-hard. We examine hardness results for unitary alphabet versions of the problems.

With a unitary alphabet, the tracing-deterministic filter has either a finite chain of states, or a finite chain with a cycle attached at the end of the chain. In such cases, which we write as FM(DF$_{\#1}$→DF) can be solved efficiently:

Theorem 3 (FM(DF$_{\#1}$→DF) is in P). *Given a tracing-deterministic input filter $\mathcal{F}$ with $|Y(\mathcal{F})| = 1$ (unitary alphabet $Y = \{y\}$), then finding the minimal tracing-deterministic filter that output simulates $\mathcal{F}$ is in P.*

Proof. We prove the hardness by giving a polynomial algorithm. First, there is always a tracing-deterministic minimizer that also has a unitary alphabet. Given some minimizer that is otherwise, simply remove the labels that are not in Y and the edges accordingly. Since the minimizer is tracing-deterministic with a unitary alphabet, it can either be (1) a finite chain of states (for finite language), or (2) a finite chain with a cycle (for Y^*). For any type-(1) minimizer, we can add a self-loop bearing label y at the leaf node, obtaining a type-(2) minimizer, being no larger while also output simulating the input filter. Thus, the tracing-deterministic minimizer can be parameterized as a type-(2) filter, i.e.,a finite chain with k states and a cycle with m states, where $k \in \mathbb{N}$ and $m \in \mathbb{N}^{>0}$. Since the $\mathcal{F}$ output simulates itself, the minimizer can be no larger. Let $n = |V(\mathcal{F})|$. Then, with $k \in \{0, 1, \ldots, n\}$ and $m \in \{1, 2, \ldots, n\}$ we can enumerate the $O(n^2)$ potential tracing-deterministic minimizers candidates. For each filter $\mathcal{F}^{\dagger}$, it takes

polynomial time to check whether $\mathcal{F}^\dagger$ output simulates $\mathcal{F}$ or not: First, denote the initial state from $\mathcal{F}$ and $\mathcal{F}^\dagger$ as v_0 and $v_0^\dagger$ respectively. Then check whether $c(v_0) \supseteq c(v_0^\dagger)$ or not. If not output simulation of $\mathcal{F}^\dagger$ is violated. Otherwise, move on to check their y-children. If the state from $\mathcal{F}$ has a y-child but the state from $\mathcal{F}^\dagger$ does not, then violation (owing to failure of language inclusion) has been detected. Once all $\mathcal{F}$'s y-children have been checked, output simulation is satisfied. It is, thus, in P to find the tracing-deterministic minimizer for $\mathcal{F}$.   □

Next, we will show that the tracing-nondeterministic minimizer for a tracing-deterministic input filter with unitary alphabet is tracing-deterministic, and it can be found in polynomial time as well:

Theorem 4 (FM(DF$_{\#1}$→SSO) and FM(DF$_{\#1}$→SMO) are in P). *Given a tracing-deterministic input filter $\mathcal{F}$ with $|Y(\mathcal{F})| = 1$ (unitary alphabet $Y = \{y\}$), then it is P to find the minimal tracing-nondeterministic filter that output simulates $\mathcal{F}$.*

Proof Sketch. One shows (see Appendix for details) that this reduces to the case in Theorem 3.   □

So far, no difference has manifested in the hardness of finding string single-output *vs.* general tracing-nondeterministic minimizers. The general cases are both intractable, and the special unitary cases, both efficient. The next section tries to probe this difference.

5   Differences between string single-output minimization and general tracing-nondeterministic minimization

In order to better understand whether there is any complexity difference between string single-output and general tracing-nondeterministic minimizers, we next consider input filters that may also include tracing-nondeterministic ones. The previous results apply to both equally because they leverage the first requirement of output simulation, namely language inclusion. The second requirement, output compatibility, relates to the colors generated, so seems more likely to be where a difference, if any exists, might be pinpointed. Thus, for this more general class of inputs, we lay open the monolithic definition of output simulating, examining the component strands, i.e., both requirements separately. As we show, language inclusions for both string single-output and general tracing-nondeterministic minimization is PSPACE-hard, but they differ in the difficulty of checking output compatibility. Output compatibility for general tracing-nondeterministic minimizers is PSPACE-hard to check, but it only takes polynomial time to check output compatibility for string single-output tracing-nondeterministic ones.

The naming convention used in the problems before is now extended: SSO→SSO indicates both the input and output filter are string single-output and potentially tracing-nondeterministic (SSO), while SMO→SMO means both input and output filters are tracing-nondeterministic (SMO).

Following the results from the previous section, it is PSPACE-hard to check language inclusion for FM(SSO→SSO) or FM(SMO→SMO):

Lemma 10. *Given a string single-output tracing-nondeterministic filter $\mathcal{F}$ and $\mathcal{F}'$, it is* **PSPACE**-*hard to check whether $\mathcal{L}(\mathcal{F}) \subseteq \mathcal{L}(\mathcal{F}')$ holds or not.*

Proof. This has been proved by the reduction presented as Lemma 9.   □

A difference does show up in checking outputs: the string single-output property means output compatibility for SSO↠SSO can be checked efficiently:

Lemma 11 (output compatibility for SSO↠SSO is in P). *Given a tracing-nondeterministic filter $\mathcal{F}$ and a non-deterministic filter $\mathcal{F}'$ such that $\mathcal{L}(\mathcal{F}) \subseteq \mathcal{L}(\mathcal{F}')$, then checking output compatibility: $\forall s \in \mathcal{L}(\mathcal{F}), \mathcal{C}(\mathcal{F}, s) \supseteq \mathcal{C}(\mathcal{F}', s)$ is in P.*

Proof. We will give a polynomial time procedure to check whether this property holds or not. First, construct a tensor product graph $\mathcal{G} = \mathcal{F} \otimes \mathcal{F}'$. Next, for every state v' in $\mathcal{F}'$, collect the set of states in $\mathcal{F}$ that are paired with it in $\mathcal{G}$, and call it $R_{v'} = \{v \in V(\mathcal{F}) | (v, v') \in V(\mathcal{G})\}$. $R_{v'}$ can be constructed in polynomial time. Then $R_{v'}$ contains the set of all states from $\mathcal{F}$ that are reached by some string that reaches v' in $\mathcal{F}'$. If there is no string from $\mathcal{F}$ that reaches v' in $\mathcal{F}'$, i.e., $\mathcal{S}_{v'}^{\mathcal{F}'} \cap \mathcal{L}(\mathcal{F}) = \varnothing$, then $R_{v'} = \varnothing$. Since both $\mathcal{F}$ and $\mathcal{F}'$ are string single-output, they are also vertex single-output. For each v', if the output of v' is the same as that of every state in $R_{v'}$, then it satisfies the output simulation criterion. Otherwise, if there exists a state $v \in R_{v'}$ such that $c(v) \neq c(v')$, then it violates output simulation, since $\mathcal{S}_v^{\mathcal{F}} \cap \mathcal{S}_{v'}^{\mathcal{F}'}$ have output that is incompatible. Therefore, we only need to check whether the output of every state paired with v' in $\mathcal{G}$ is the same as that of v', which can be done in polynomial time.   □

However, without input filter being string single-output, checking output compatibility is **PSPACE**-hard:

Lemma 12 (output compatibility for SMO↠SMO is PSPACE-hard). *Given a tracing-nondeterministic input filter $\mathcal{F}$ and a tracing-nondeterministic filter $\mathcal{F}'$ such that $\mathcal{L}(\mathcal{F}) \subseteq \mathcal{L}(\mathcal{F}')$, then it is* **PSPACE**-*hard to check the output compatibility: $\forall s \in \mathcal{L}(\mathcal{F}), \mathcal{C}(\mathcal{F}, s) \supseteq \mathcal{C}(\mathcal{F}', s)$.*

Proof. Via reduction from language inclusion, which is shown to be **PSPACE**-hard in Lemma 10. Given non-deterministic filters $\mathcal{A}$ and $\mathcal{B}$ such that $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{B})$, to check the property of output compatibility, we first construct a product graph between $\mathcal{A}$ and $\mathcal{B}$, denoted as $\mathcal{J} = \mathcal{A} \otimes \mathcal{B}$. Then we construct a graph union with $\mathcal{J}$ and $\mathcal{A}$, calling it $\mathcal{G}$, consisting of vertices and edges from both $\mathcal{J}$ and $\mathcal{A}$. Hence, we have $\mathcal{L}(\mathcal{G}) = \mathcal{L}(\mathcal{A})$. Next, treat $\mathcal{G}$ as a filter $\mathcal{F}$ by coloring the vertices from $\mathcal{J}$ green, and vertices from $\mathcal{A}$ red. Treat $\mathcal{A}$ as a filter $\mathcal{F}'$ with all states green. Then, we have $\mathcal{L}(\mathcal{F}') = \mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{G}) = \mathcal{L}(\mathcal{F})$. If $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{B})$, then for every string $s \in \mathcal{L}(\mathcal{A})$, s will reach a red state (from $\mathcal{A}$) and a green state (from $\mathcal{J}$) in $\mathcal{F}$. Hence $\mathcal{C}(F, s) = \{\text{red}, \text{green}\}$. Since $\forall s \in \mathcal{L}(F), \mathcal{C}(F', s) = \{\text{green}\}$, we have that $\mathcal{F}$ and $\mathcal{F}'$ are output compatible. If $\mathcal{L}(\mathcal{A}) \nsubseteq \mathcal{L}(\mathcal{B})$, then there exists a string s such that $s \in \mathcal{L}(\mathcal{A})$ but $s \notin \mathcal{L}(\mathcal{B})$. Hence, s will only reach red states (from $\mathcal{A}$) in $\mathcal{F}$. Since s reaches green states in $\mathcal{F}'$, then it violates the output compatibility property. Therefore, the problem of checking language inclusion was reduced in

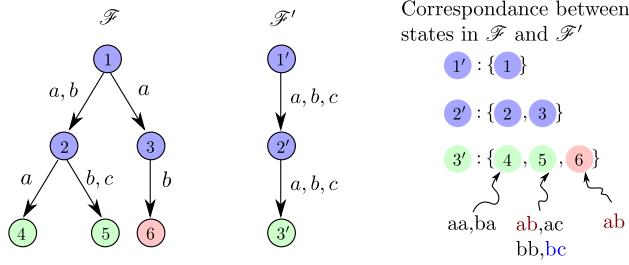polynomial time to the problem of checking output compatibility. So the problem is PSPACE-hard as well. ☐

Fig. 7: An illustration that develops intuition for the hardness of checking output compatibility for SMO→SMO.

The difference in hardness for checking output compatibility between SSO→SSO and SMO→SMO can be seen intuitively. As illustrated via Figure 7: a tracing-nondeterministic input filter $\mathcal{F}$ is shown at the left, and an output filter $\mathcal{F}'$ is shown in the middle. For every state v' in $\mathcal{F}'$, we have collected the set of states $R_{v'}$ in $\mathcal{F}$ that are reached by the strings reaching v', exactly as in the procedure in Lemma 11. For every v', its corresponding set $R_{v'}$ is shown at the right of Figure 7. As illustrated in Lemma 11, if $\mathcal{F}$ is string single-output, then we only need to check the output of every individual state inside each $R_{v'}$ for v'. We assert a violation of output compatibility if there is a state whose output is different from that of v'. But if $\mathcal{F}$ is not string single-output, we cannot immediately claim that a failure was found when an output inconsistency is found between state $v \in R_{v'}$ and v' because the strings $\mathcal{S}_v^{\mathcal{F}} \cap \mathcal{S}_{v'}^{\mathcal{F}'}$ may non-deterministically reach other states that share the same output as v'. An example is shown in Figure 7, for vertex $3'$, $R_{3'} = \{4, 5, 6\}$. Vertex 6 outputs pink, while $3'$ outputs green. So there is an inconsistency. But the strings 'ab' reaching 6 and $3'$, also reach a green state nondeterministically. In this case, $\mathcal{C}(\mathcal{F}', ab) \subseteq \mathcal{C}(\mathcal{F}, ab)$ still holds. Therefore, when $\mathcal{F}$ is not string single-output, one must check the output across all the states that are nondeterministically reached in $\mathcal{F}$, which is PSPACE-hard.

Speaking informally, given that checking language inclusion is at least as hard as either of the output compatibility checks, one expects to pay at least that price for determining output simulation. Lemma 11 suggests that price must be paid twice for SMO→SMO, in the sense that there are two instances of this problem embedded in corroborating output simulation. While for SSO→SSO the language inclusion check seems to dominate. We note that the conditions on the last two lemmas may have some relevance in applications, for instance, when one has domain knowledge (or an oracle) that tells you language inclusion holds.

6 Summary and Conclusion

This paper explores a new type of nondeterminism that is novel in the context of combinatorial filtering. It has attractive properties when used, for instance, in encoding feedback plans/policies concisely: First, the outputs that such filters produce are consistent, isolating specific tracing choices (or rewinding operations) made during processing from being manifested externally. As mappings,

they exhibit deterministic behavior. Secondly, they provide degrees of freedom absent from deterministic filters, which facilitate greater compression. It is curious that this should actually be possible, but our example demonstrates a clear separation in sizes between the classes. To initiate study of this class, the string single-output filters, we have examined hardness of size minimization, establishing that the general problems are of the same complexity class as classical nondeterminism. String single-output filter minimization is PSPACE-hard in terms of the language inclusion for output simulation, and not output compatibility; general nondeterministic filter minimization, it turns out, is PSPACE-hard in terms of both the properties — a fact never before realized.

References

1. Connell, J.H.: Minimalist mobile robotics: a colony-style architecture for an artificial creature, Perspectives in AI, vol. 5. Academic Press, Inc. (1990)
2. Kao, J.Y., Rampersad, N., Shallit, J.: On NFAs where all states are final, initial, or both. Theoretical Computer Science **410**(47-49), 5010–5021 (2009)
3. O'Kane, J.M., Shell, D.A.: Concise planning and filtering: hardness and algorithms. IEEE Trans. on Automation Science and Engineering **14**(4), 1666–1681 (Oct 2017)
4. Rabin, M.O., Scott, D.: Finite Automata and Their Decision Problems. IBM Journal of Research and Development **3**(2), 114–125 (1959)
5. Rahmani, H., O'Kane, J.M.: On the relationship between bisimulation and combinatorial filter reduction. In: Proceedings of IEEE International Conference on Robotics and Automation. pp. 7314–7321. Brisbane, Australia (2018)
6. Rahmani, H., O'Kane, J.M.: Integer linear programming formulations of the filter partitioning minimization problem. Journal of Combinatorial Optimization **40**, 431–453 (2020)
7. Saberifar, F.Z., Ghasemlou, S., O'Kane, J.M., Shell, D.A.: Set-labelled filters and sensor transformations. In: Robotics: Science and Systems. Ann Arbor, MI (2016)
8. Saberifar, F.Z., Mohades, A., Razzazi, M., O'Kane, J.M.: Combinatorial filter reduction: Special cases, approximation, and fixed-parameter tractability. Journal of Computer and System Sciences **85**, 74–92 (2017)
9. Saberifar, F.Z., O'Kane, J.M., Shell, D.A.: Inconsequential improprieties: Filter reduction in probabilistic worlds. In: Proceedings of IEEE/RSJ International Conference on Intelligent Robots and System. Vancouver, Canada (2017)
10. Taylor, R.H., Mason, M.T., Goldberg, K.Y.: Sensor-based manipulation planning as a game with nature. In: International Symposium on Robotics Research (1988)
11. Tovar, B., Cohen, F., Bobadilla, L., Czarnowski, J., Lavalle, S.M.: Combinatorial filters: Sensor beams, obstacles, and possible paths. ACM Transactions on Sensor Networks **10**(3), 1–32 (2014)
12. Zhang, Y., Rahmani, H., Shell, D.A., O'Kane, J.M.: Accelerating combinatorial filter reduction through constraints. In: Proceedings of IEEE International Conference on Robotics and Automation. pp. 9703–9709 (May 2021)
13. Zhang, Y., Shell, D.A.: Cover combinatorial filters and their minimization problem. In: Algorithmic Foundations of Robotics XIV. pp. 90–106. Springer (2021)
14. Zhang, Y., Shell, D.A.: On nondeterminism in combinatorial filters. In: Proceedings of IEEE International Conference on Robotics and Automation. (To appear). Philadelphia, PA, USA (2022), available at https://arxiv.org/abs/2107.07111