Decoupled Fitness Criteria for Reactive Systems

Derek Egolf(✉) and Stavros Tripakis

Northeastern University, Boston, MA, USA
{egolf.d,stavros}@northeastern.edu

Abstract. The correctness problem for reactive systems has been thoroughly explored and is well understood. Meanwhile, the efficiency problem for reactive systems has not received the same attention. Indeed, one correct system may be less fit than another correct system and determining this manually is challenging and often done ad hoc. We (1) propose a novel and general framework which automatically assigns comparable fitness scores to reactive systems using interpretable parameters that are decoupled from the system being evaluated, (2) state the computational problem of evaluating this fitness score and reduce this problem to a matrix analysis problem, (3) discuss symbolic and numerical methods for solving this matrix analysis problem, and (4) illustrate our approach by evaluating the fitness of nine systems across three case studies, including the Alternating Bit Protocol and Two Phase Commit.

Keywords: Formal methods · Verification · Reactive systems

1 Introduction

Correctness guarantees help us avoid irritating, costly, and, in some cases, deadly implementation bugs. However, two systems that both satisfy a correctness specification may differ with respect to efficiency. Inefficient systems result in real world consequences: delaying content delivery, using excess energy, and wasting clock cycles better spent elsewhere.

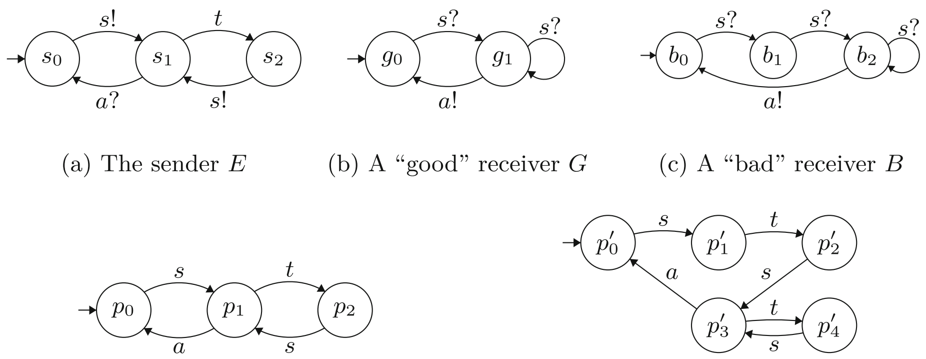
Much like reasoning about correctness, reasoning about efficiency is cognitively demanding, prone to errors, and requires expert insight. The framework proposed in this paper strives to eliminate this human burden, mitigate these errors, and capture the expert's insight and intentions in the parameters of the framework. The proposed framework accomplishes these goals by assigning a comparable *fitness score* to every system, such that we can decide between two systems on the basis of their score. Consider the following example.

Example 1. Consider the finite labeled transition systems (LTSs) depicted in Fig. 1. Labels s, a, t represent *send*, *acknowledge* (ack), and *timeout* respectively. The symbols !, ? (output, input) denote rendezvous communication in which a

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2023

! transition can only be taken in one LTS if the corresponding ? transition is taken in another LTS. Transitions with neither !, nor ?, can be taken freely.

LTS E represents a *sender* in the environment. LTSs G and B are 'good' and 'bad' receivers, respectively. B is 'bad' in the sense that it waits for two *send* actions before replying with an acknowledgement, whereas G replies right away. The synchronous products of the sender E with receivers G and B, denoted $E||G$ and $E||B$, are LTSs M and M', respectively. Both M and M' are *correct*, in the sense that they satisfy the specification *every s is eventually followed by an a* (given some fairness assumptions that prevent a from being ignored indefinitely). Because they both satisfy this specification, M and M' are indistinguishable from the perspective of traditional verification and synthesis. However, M is intuitively preferable to M' because G is a better receiver than B. As we will show in Sect. 5, our framework assigns fitness scores 0.25 and 0.14 to M and M', respectively, and thus distinguishes M as a better system. □



(a) The sender E (b) A "good" receiver G (c) A "bad" receiver B

(d) The product system $M := E||G$ (e) The product system $M' := E||B$

Fig. 1. A simple communication protocol modeled with finite LTSs.

The exact nature of the fitness score depends on the application domain. Our framework *decouples* the description of the system (e.g., the LTSs of Fig. 1) from a set of domain-specific *parameters* which capture user preferences.

By assigning fitness scores to systems, as in the example above, our framework can be used for performance evaluation. Our framework is additionally motivated by recent work in the synthesis of distributed protocols [5]. Unlike humans, synthesis tools typically ignore efficiency considerations. In some cases, these tools generate systems that are, strictly speaking, correct (i.e., they satisfy their logical specification), yet clearly unorthodox or even inefficient [6]. In such cases, we can use our framework to rank automatically generated systems according to their fitness score. In other cases, we may want to generate *all* correct systems [24], potentially with the aim of doing *fitness-optimal synthesis* (c.f. page 8).

In summary, the contributions of this paper are as follows: (1) We propose a novel and general framework for automatically assigning a comparable fitness score to a system; this framework uses interpretable parameters that are decoupled from the system being evaluated. (2) We provide an automated method for computing fitness scores; our method ultimately reduces the fitness-score computation problem to a matrix analysis problem. (3) We discuss symbolic and numerical methods for solving this matrix analysis problem. (4) We present an implementation and evaluation of our framework: our prototype tool allows, in a matter of seconds, to automatically compute the fitness of nine automatically synthesized systems.

We organize the rest of the paper as follows. Section 2 formalizes preliminary concepts. Section 3 presents our framework. Section 4 presents a method to compute fitness scores. Section 5 illustrates our approach on the communication protocol of Example 1, Two Phase Commit, and the Alternating Bit Protocol taken from [6]. Section 6 discusses related work. Section 7 concludes the paper.

2 Preliminaries

$\mathbb{N}$, $\mathbb{Q}$, $\mathbb{R}$, $\mathbb{R}_{\geq 0}$, and $\mathbb{B} = \{0, 1\}$ denote the sets of natural, rational, real, non-negative real numbers, and booleans, respectively. A function $h : \mathbb{N}^d \to \mathbb{Q}$ is a *scalar arithmetic function* if h can be written in terms of basic scalar arithmetic operations $+, -, \times, /$, applied to its natural number arguments.

In traditional verification, we typically only consider the yes/no question: does the system produce any violating traces. While this question allows us to discard of the relative abundance of traces, the question of fitness is not so. All else equal, if a system is capable of producing the same 'unfit' trace by executing any one of many distinct runs, then that system is worse than a system that can produce the unfit trace in just one particular way. Toward this end, we require a notion of multisets.

A **multiset** X over domain D is a function $X : D \to \mathbb{N}$, where $X(x)$ represents the *multiplicity* of element x, i.e., how many times x occurs in X. $\mathcal{M}(D)$ denotes the class of all multisets over D, i.e., the set of all functions $X : D \to \mathbb{N}$. If $X(x) = m$, then we write $x \in_m X$ (possibly, $m = 0$). The *cardinality* of X, denoted $|X|$, is the sum of the multiplicities of all members of the domain D. We write multisets as $\{\!\{...\}\!\}$ to differentiate them from sets.

Example 2. We denote by $X = \{\!\{0, 0, 1, 1, 1\}\!\}$ the multiset where $0 \in_2 X$ and $1 \in_3 X$. Then: $|X| = 2 + 3 = 5$. □

If $A \subseteq D$ and $X : D \to \mathbb{N}$ is a multiset, then X *restricted to* A is a new multiset, denoted $X|_A$: $D \to \mathbb{N}$ and defined as follows. If $x \notin A$, $X|_A\ (x) = 0$ and otherwise if $x \in A$, then $X|_A\ (x) = X(x)$. Let $X : D \to \mathbb{N}$ be a multiset and let $f : D \to D'$ be a function. Then intuitively, the *image of* X *by* f is a multiset denoted $f \odot X$ obtained by applying f to the members of X. E.g. if $f(x) = x^2$, then $f \odot \{\!\{2, -2, 3, 3, 3\}\!\} = \{\!\{4, 4, 9, 9, 9\}\!\}$. Formally, we define $f \odot X : D' \to \mathbb{N}$ as follows. $(f \odot X)(y) := |(X|_{D_y})|$, where $D_y := \{x \in D \mid f(x) = y\}$. We may treat a set as a multiset with all multiplicities as 0 or 1 and take its image by f to obtain a multiset. If $X \in \mathcal{M}(\mathbb{N}^d)$ and $1 \leqslant i \leqslant d$, then $sum(X, i) = \sum_{x \in_c X} cx_i$, where x_i is the ith component of $x \in \mathbb{N}^d$. E.g. $sum(\{\!\{(1,2),(1,2),(3,4)\}\!\}, 2) = 2 + 2 + 4$.

A **finite labeled transition system** (LTS) is a tuple $M = \langle \Sigma, Q, Q_0, \Delta \rangle$, where: Σ is a finite set of labels; Q is a finite set of states; $Q_0 \subseteq Q$ is the set of initial states; $\Delta \subseteq Q \times \Sigma \times Q$ is the transition relation. An *n-length run of M* is a sequence $t = q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} q_2 ... \xrightarrow{a_n} q_n$ such that $q_0 \in Q_0$ and $(q_i, a_{i+1}, q_{i+1}) \in \Delta$ for all $i = 0, ..., n-1$. The *trace* of t, denoted $\mathrm{Lab}(t)$, is the sequence of labels $a_1 a_2 ... a_n$, while $\mathrm{Sts}(t) = q_0 q_1 ... q_n$ is the sequence of states visited during t. $[\![M]\!]_n$ denotes the set of all n-length runs of M. Two runs $t_1, t_2 \in [\![M]\!]_n$ may have equivalent traces, i.e., $\mathrm{Lab}(t_1) = \mathrm{Lab}(t_2)$. We denote the multiset of all n length traces of M as $M_n = \mathit{Lab} \odot [\![M]\!]_n$. We denote the 0 length trace as ε. Then $[\![M]\!]_0 = Q_0$ and M_0 is the multiset containing ε once for each state in Q_0.

Example 3 (Two Systems). We define two LTSs $M^{(1)}$ and $M^{(2)}$ over $\Sigma = \{0, \$\}$. We interpret the traces of these systems as follows: \$'s are money that we receive, and 0's are lapses in this income. Intuitively, we prefer behaviors that maximize the rate at which we receive \$'s.

Let $M^{(1)}$ be the LTS with one state and a self-loop with label \$. So $M_n^{(1)}$ contains one n length trace of multiplicity 1: $\$^n$. Let $M^{(2)}$ be the LTS that alternates between two states, outputting \$ when leaving the initial state and 0 when leaving the other. So $M_n^{(2)}$ contains one trace of multiplicity 1: $(\$0)^{\lfloor n/2 \rfloor}\$^{(n \bmod 2)}$, i.e., even length prefixes end in 0 and odd length prefixes end in \$. □

A distributed system is typically modeled as the *product* of a set of LTSs. This product can be defined in the standard way, and is itself a monolithic LTS.

A **deterministic finite automaton** (DFA) is a tuple $M = \langle \Sigma, Q, q_0, Q_{acc}, \delta \rangle$, where: Σ is a finite alphabet; Q is a finite set of states; $q_0 \in Q$ is the single initial state; $Q_{acc} \subseteq Q$ is the set of accepting states; $\delta : Q \times \Sigma \to Q$ is the transition function. Unlike a generic LTS, every trace $w \in \Sigma^*$ corresponds to one and only one finite run of a DFA M.

3 A Formal Framework for Capturing Fitness

Our framework assigns a real number called a *fitness score* to every system. The key idea of our framework is that it *decouples* the description of the system from the following set of domain-specific framework parameters: (1) A finite alphabet Σ, e.g., $\{0, \$\}$. (2) A *fitness function*, $f : \Sigma^* \to \mathbb{N}^d$. This function measures some quantity of finite prefixes of infinite traces. (3) An *aggregate function*, $@ : \mathcal{M}(\mathbb{N}^d) \to \mathbb{Q}$. This function takes a multiset of fitness values, $X \in \mathcal{M}(\mathbb{N}^d)$, and compiles the values into a single value. Examples include min, max, average, etc. taken over arithmetic combinations of natural numbers.[1] In addition, the framework may also include: (4) A comparison relation, $\preccurlyeq$, used to compare the

[1] Slight generalizations to the framework, omitted here for the sake of simplicity, are able to capture, e.g., aggregates that output tuples of rational numbers [22].

fitness scores of two different systems. We next provide examples and formal definitions of these parameters.

Fitness Functions: The *rate* function is an example of a fitness function:

Definition 1 (Fitness Function: Rate of \$). *For $\Sigma = \{0, \$\}$ define rate*$_\$$ $(w) = (\#_\$(w), |w|)$*, where* $\#_\$(w)$ *is the number of \$'s in* w *and* $|w|$ *is the length of* w*. This fitness function treats a label as a unit of time.* □

Example 4 (Rate of \$ Applied). Recall the systems $M_n^{(1)} = \{\!\{\$^n\}\!\}$ and $M_n^{(2)} = \{\!\{(\$0)^{\lfloor n/2 \rfloor}\$^{(n \bmod 2)}\}\!\}$ from Example 3. We apply $f := rate_\$$ to the n-length partial runs of these systems. Taking the image of $M^{(1)}$ and $M^{(2)}$ by f yields:

$$f \odot M_n^{(1)} = \{\!\{f(\$^n)\}\!\} = \{\!\{(n, n)\}\!\}$$
$$f \odot M_n^{(2)} = \{\!\{f((\$0)^{\lfloor n/2 \rfloor}\$^{(n \bmod 2)})\}\!\} = \{\!\{(\lceil n/2 \rceil, n)\}\!\}$$

□

We represent a fitness function $f : \Sigma^* \to \mathbb{N}^d$ by a d-tuple $\langle f_1, ..., f_d \rangle$, where each $f_i = \langle \Sigma, Q_i, q_i^0, Q_i^{acc}, \delta_i \rangle$ is a DFA. Specifically, consider an input $w \in \Sigma^*$. When the DFA f_i consumes w, it visits a sequence of states, $\hat{q} = q_i^0, q_i^1, ..., q_i^m$. Interpreting f_i as a function $f_i : \Sigma^* \to \mathbb{N}$, we define $f_i(w)$ as the number of times an accepting state is visited in $\hat{q}$. We then define the fitness function $f : \Sigma^* \to \mathbb{N}^d$ so that $f(w) = (f_1(w), ..., f_d(w))$. For instance, Fig. 2 depicts the DFA representation of *rate*$_\$$ from Definition 1 and, e.g., $f(\$0\$\$0) = (3, 5)$.

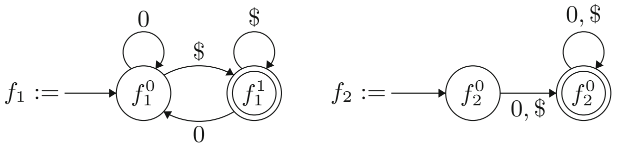


Fig. 2. The two DFA representing the *rate*$_\$$ fitness function: f_1 computes the number of \$'s in a word; f_2 computes the length of the word.

Aggregate Functions: The *average rate* function is one example of an aggregate function. The average rate function treats ordered pairs as fractions and takes the average value:

Definition 2 (Aggregate Function: Average Rate). *For* $X \in \mathcal{M}(\mathbb{N}^2)$*, let:*

$$@_{avg}(X) = \frac{1}{|X|} \sum_{(p,q) \in_m X} m \cdot \frac{p}{q}$$

□

Example 5. This example emphasizes the role of multiplicity in aggregates. For instance, if $X := \{\!\{(1,3),(1,3),(2,3)\}\!\}$, then the (1,3) term is counted twice:

$$@_{avg}(X) \quad = \quad \frac{1}{|X|} \sum_{(p,q) \in_m X} m \cdot \frac{p}{q} \quad = \quad \frac{1}{3}(2 \cdot \frac{1}{3} + \frac{2}{3}) \quad = \quad 4/9$$

□

Example 6. This example applies $@_{avg}$ to the running example (Example 3). The average is moot here as there is only one partial trace of each length. Recall from Example 4 that $f \odot M_n^{(1)} = \{\!\{(n,n)\}\!\}$ and $f \odot M_n^{(2)} = \{\!\{(\lceil n/2 \rceil, n)\}\!\}$, where $f := rate_\$$. We can apply average rate to these images: $@_{avg}(f \odot M_n^{(1)}) = n/n = 1$ and $@_{avg}(f \odot M_n^{(2)}) = \lceil n/2 \rceil / n$. □

Another example of an aggregate function is the *maximum rate* function:

Definition 3 (Aggregate Function: Maximum Rate). *For $X \in \mathcal{M}(\mathbb{N}^2)$:*

$$@(X) = \max\{p/q \mid (p,q) \in X\}$$

□

Example 7. For instance, if $X := \{\!\{(1,3),(1,3),(2,3)\}\!\}$, then:

$$@(X) = \max\{1/3, 1/3, 2/3\} = 2/3$$

□

In principle, an aggregate function can be any mathematical function with the appropriate type (c.f. page 4). But for the sake of computation, we want an aggregate function to be represented as a scalar arithmetic function $h(x_1, x_2, ..., x_d)$. We say that $h : \mathbb{N}^d \to \mathbb{Q}$ is a *faithful representation* of $@ : \mathcal{M}(N^d) \to \mathbb{Q}$ if and only if for all $X \in \mathcal{M}(\mathbb{N}^d), @(X) = h(sum(X,1), ..., sum(X,d))$. We will see in Sect. 4 that this form of representation and the definitions that follow are key, as the heart of our method is computing each $sum(X,i)$, where $X = f \odot M_n$. The importance should be clear by the time we state our primary correctness result, Theorem 1.

While h might not be a faithful representation of @ for all X, h may be a faithful representation assuming that X satisfies some condition. The fitness function may in turn guarantee that X satisfies that condition. Fortunately, this relationship holds between $@_{avg}$ (Def. 2) and $rate_\$$ (Def. 1). The following definition and lemmas capture this useful situation:

Definition 4 (Conditional Representation and Compatible). *Let Ψ be a predicate over $\mathcal{M}(\mathbb{N}^d)$, i.e., a mapping $\Psi : \mathcal{M}(\mathbb{N}^d) \to \mathbb{B}$. Additionally, let $@ : \mathcal{M}(\mathbb{N}^d) \to \mathbb{Q}$ be an aggregate function and $h : \mathbb{N}^d \to \mathbb{Q}$ be a scalar arithmetic function. Then h is a* conditional representation *of @ subject to Ψ if and only if for all $X \in \mathcal{M}(\mathbb{N}^d)$, if $\Psi(X)$ holds (i.e., $\Psi(X) = 1$), then $@(X) = h(sum(X,1), ..., sum(X,d))$.*

Let h be a conditional representation of the aggregate function @ subject to Ψ. Let f be a fitness function. We say that h and f are compatible *when $\Psi(f \odot M_n)$ holds for any LTS M and any $n \in \mathbb{N}$.* □

Let predicate $\Psi_{rate}(X)$:= 'If $(p, q), (p', q') \in X$, then $q = q'$.' Then we have the following two lemmas.

Lemma 1. *Let $X \in \mathcal{M}(\mathbb{N}^2)$ and suppose $\Psi_{rate}(X)$ holds. Then $@_{avg}(X) = sum(X, 1)/sum(X, 2)$. Therefore, $@_{avg}$ is conditionally represented by $h(x_1, x_2) = x_1/x_2$, subject to Ψ_{rate}.* □

Lemma 2. *For all $n \in \mathbb{N}$ and all LTS M, $\Psi_{rate}(rate_\$ \odot M_n)$ holds. Hence, $rate_\$$ and $h(x_1, x_2) = x_1/x_2$ are compatible.* □

Lemma 1 follows from the fact that the average of a multiset of fractions is equal to the sum of the numerators divided by the sum of the denominators when the denominators are all equal. Lemma 2 is immediate: if $w \in M_n$ and $rate_\$(w) = (p, q)$, then $q = n$. From Lemma 1 and 2 it follows that $@_{avg}$ and $rate_\$$ are compatible. Therefore, if the fitness function is $rate_\$$ we can represent $@_{avg}(X)$ with the expression $sum(X, 1)/sum(X, 2)$.

Note that fitness functions other than $rate_\$$ might not be compatible with $@_{avg}$. For instance, let $f(w) = (\#_\$(w), \#_0(w))$, which measures the number of \$'s per 0. f does not satisfy Ψ_{rate}, but it is a realistic fitness function. In the case of $rate_\$$, time is measured by the observation of any label from Σ. Now for f, time is measured using only 0. If \$ denotes a local action of a server and 0 an interaction between two servers, f captures communication complexity. We leave handling of such non-compatible fitness functions for future work.

Fitness Score: Given alphabet Σ, fitness function f, and aggregate function @, the *fitness score* of an LTS M, denoted $@_f M$, is defined to be the limit $@_f M := \lim_{n\to\infty} @(f \odot M_n)$. This limit is a value in $\mathbb{R}_{\geq 0} \cup \{\infty, \bot\}$. The limit either: converges to a value $v \in \mathbb{R}_{\geq 0}$, in which case the score is v; or increases without bound, in which case we assign the value ∞; or exhibits some other behavior such as oscillation, in which case we assign the ill-behaved value $\bot$.

Comparison Relations: A comparison relation $\preccurlyeq$ is a subset of $(\mathbb{R}_{\geq 0} \cup \{\infty, \bot\})^2$. If $(a, b) \in \preccurlyeq$, we write $a \preccurlyeq b$. If neither $a \preccurlyeq b$ nor $b \preccurlyeq a$, we say that a and b are *incomparable*. Ignoring ∞ and $\bot$ for the moment, $\preccurlyeq$ could be any one of $\leqslant, <, \geqslant$, or $>$ on $\mathbb{R}$. Extending this comparator to ∞ and $\bot$ would be up to the user. One choice is to have these values be incomparable to any other value. Note that, even though the aggregate @ maps to $\mathbb{Q}$, $\preccurlyeq$ needs to compare real (and not just rational) numbers because the fitness score involves taking a limit. The semantics of $a \preccurlyeq b$ are that a is preferrable to b.

Example 8. Concluding our analysis of Example 3, consider an instance of our framework with fitness function $rate_\$$ (Definition 1), aggregate function $@_{avg}$

(Definition 2), and comparison operator $\preccurlyeq := \geqslant$ (since we prefer high rates of income). We can then compare the two simple systems introduced in Example 3. Building on what we have presented so far (c.f. Examples 4 and 6), we have:

$$\begin{aligned} @_f M^{(1)} &= \lim_{n\to\infty} @(f \odot M_n^{(1)}) &= \lim_{n\to\infty} 1 &= 1 \\ @_f M^{(2)} &= \lim_{n\to\infty} @(f \odot M_n^{(2)}) &= \lim_{n\to\infty} \frac{\lceil n/2 \rceil}{n} &= 1/2 \end{aligned}$$

Because $@_f M^{(1)} \geqslant @_f M^{(2)}$, we conclude $@_f M^{(1)} \preccurlyeq @_f M^{(2)}$ and therefore we prefer $M^{(1)}$ to $M^{(2)}$. This result aligns with our intuitions; we would rather receive a dollar every day than a dollar every other day. □

Evaluation, Comparison, and Synthesis Problems: Within our framework, we can consider various types of computational problems. A basic problem is that of *evaluating* the fitness score of a given system: *Given a fitness function f, an aggregate function $@$, and a system M, compute $@_f M$.* Another problem is that of *comparing* two systems: *Given a fitness function f, an aggregate function $@$, a comparison relation $\preccurlyeq$, and two systems M_1, M_2, check whether $@_f M_1 \preccurlyeq @_f M_2$.* We can also consider *fitness-optimal synthesis* problems, which ask to find a system with the best fitness score, perhaps subject to some correctness constraint (e.g. an LTL formula). Of these problems, in the rest of this paper we will focus on the fitness evaluation problem:

Problem 1 (Fitness Evaluation Problem). Let $M = \langle \Sigma, Q, Q_0, \Delta \rangle$ be a finite LTS and let $f = \langle f_1, ..., f_d \rangle$, where each f_i is represented as a DFA. Let $@ : \mathcal{M}(\mathbb{N}^d) \to \mathbb{Q}$ be an aggregate function represented by the scalar arithmetic function $h : \mathbb{N}^d \to \mathbb{Q}$. Finally, suppose that h and f are compatible. The fitness evaluation problem is to compute the fitness score $@_f M$ of M, i.e., to compute $\lim_{n\to\infty} @(f \odot M_n)$. □

4 Reducing Fitness Evaluation to Matrix Analysis

In this section we propose a method to solve Problem 1 that consists in the following steps (assuming the same notation and setup as in Problem 1):

1. Compute the product automaton $P_i = M || f_i$, for each $i \in \{1, ..., d\}$.
2. For each P_i, compute a matrix-vector pair (ξ_i, v_i) representing a *recurrence relation*. We call the matrix ξ_i the *recurrence matrix* and the vector v_i the *initial condition vector*.
3. Solve the following matrix analysis problem:

Problem 2. Let $g_i(n) = (\xi_i^{n+1} v_i)_0$ for fixed square matrices $\xi_1, ..., \xi_d$ and vectors $v_1, ..., v_d$ with non-negative integer entries and where $(u)_0$ denotes the first entry of vector u. Let $h : \mathbb{N}^d \to \mathbb{Q}$ be a scalar arithmetic function. Compute $\lim_{n\to\infty} h(g_1(n), g_2(n), ..., g_d(n))$. □

The motivation for the above steps follows. In step 1, the product P_i represented all simultaneous paths through M and f_i. I.e., a path through P_i corresponds to taking a path through M and handing the transition label encountered at each step to the automaton representing f_i. As mentioned, step 2 computes a recurrence relation, which is reasonable because the number of accepting states visited across $(n+1)$-length paths is related to certain quantities computed over the n-length paths. The exact relationship is explained in detail in Sect. 4.1.

The correctness of the reduction to Problem 2 (Corollary 1) hinges on the fact that $g_i(n) = \mathit{sum}(f \odot M_n, i)$, i.e., computing $\mathit{sum}(f \odot M_n, i)$ (which is then an input to the aggregate function) reduces to computing the nth term of a recurrence relation, which in turn reduces to taking a matrix power.

Step 1 of the method (computing automata products) is standard. Therefore, in the rest of this section, we focus on explaining Steps 2 and 3.

4.1 Step 2: Constructing the Recurrence Relation

We will first explain the recurrence relation construction by example and then give the general construction.

By Example: We skip the first step of the method and assume that we have a product $P_1 = M||f_1$. In particular, we consider the automaton of Fig. 3.

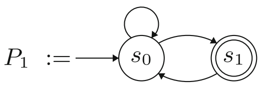


Fig. 3. A toy product $P_1 = M||f_1$. P_1 has two states named s_0 and s_1. s_0 is the initial state and s_1 is the accepting state. The transition labels from Σ are not needed and hence are omitted.

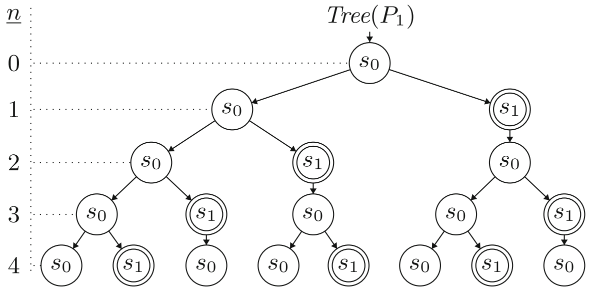


Fig. 4. Partial unfolding of the automaton of Fig. 3 into a tree up to depth 4. The column labeled n denotes the number of transitions taken.

From the automaton of Fig. 3 we extract the following recurrence relations:

$$\beta_{n+1}^{s_0} = \beta_n^{s_0} + \beta_n^{s_1}, \qquad \beta_0^{s_0} = 1 \tag{1}$$

$$\beta_{n+1}^{s_1} = \beta_n^{s_0}, \qquad \beta_0^{s_1} = 0 \tag{2}$$

$$\alpha_{n+1}^{s_0} = \alpha_n^{s_0} + \alpha_n^{s_1}, \qquad \alpha_0^{s_0} = 0 \tag{3}$$

$$\alpha_{n+1}^{s_1} = \alpha_n^{s_0} + \beta_n^{s_0}, \qquad \alpha_0^{s_1} = 0 \tag{4}$$

$$\alpha_n = \alpha_n^{s_0} + \alpha_n^{s_1}, \qquad \alpha_\varnothing = 0 \tag{5}$$

where (as visual aid we provide Fig. 4, which displays the unfolding of P_1 of Fig. 3 into a tree containing all paths up to length 4):

- β_n^q is the total number of n-length paths through P_1 ending in state q, e.g., $\beta_0^{s_0} = 1$, $\beta_0^{s_1} = 0$, $\beta_3^{s_0} = 3$, $\beta_4^{s_1} = 3$.
- α_n^q is the total number of accepting states visited along all n-length paths through P_1 restricted to paths terminating in state q, e.g., $\alpha_1^{s_0} = 0$, $\alpha_1^{s_1} = 1$, $\alpha_3^{s_0} = 2$.
- α_n is the total number of accepting states visited along all n-length paths through P_1, e.g., $\alpha_0 = 0$, $\alpha_1 = 1$, $\alpha_2 = 2$, $\alpha_3 = 5$, $\alpha_4 = 10$.
- $\alpha_\varnothing$ is a dummy variable representing the initial condition of α_n. Notice that the α_n term of the recurrence is unique in that no other term depends on it.

We determine each equation of the example recurrence relation as follows:

Equations (1) capture the number of paths of a certain length ending in state s_0. The initial value $\beta_0^{s_0}$ is 1 because s_0 is an initial state. Otherwise, notice that s_0 has two predecessors: s_0 and s_1. To walk an $(n+1)$-length path ending in s_0, it is necessary and sufficient to walk an n-length path to one of its predecessors and then take one more step. Hence, we compute $\beta_{n+1}^{s_0}$ as the sum of $\beta_n^{s_0}$ and $\beta_n^{s_1}$. Analogous reasoning yields Equations (2); notice the initial value $\beta_0^{s_1}$ is 0 since s_1 is not an initial state.

Equations (3) capture the number of accepting states visited along all paths of a certain length ending in state s_0. Importantly, s_0 is not an accepting state. Therefore, adding it to an n-length path will not change the number of accepting states visited along that path. Hence, as with β, we can compute $\alpha_{n+1}^{s_0}$ as the sum of $\alpha_n^{s_0}$ and $\alpha_n^{s_1}$. The initial value $\alpha_0^{s_0}$ is 0 because s_0 is an initial state, but not an accepting state.

Equations (4) capture the number of accepting states visited along all paths of a certain length ending in state s_1. Unlike s_0, the state s_1 is an accepting state. Therefore, the $(n+1)$th step contributes to the number of accepting states visited, in particular for each path it will increase the count by one. There are $\beta_n^{s_0}$ such paths, hence the inclusion of that term in addition to the α of the predecessor s_0. The initial value $\alpha_0^{s_1}$ is 0 because s_1 is an accepting state, but not an initial state.

Equations (5) capture the accepting states along all paths of a certain length. The initial value $\alpha_\varnothing$ is irrelevant; we use 0 for simplicity. Otherwise, this equation merely captures the fact that we can partition the paths of length n based on which state they end in and take a sum over that partition to compute a value over all paths.

We can represent these recurrence relation as a matrix-vector pair (ξ_1, v_1), where:

$$v_1 = \begin{bmatrix} \alpha_\varnothing \\ \alpha_0^{s_0} \\ \alpha_0^{s_1} \\ \beta_0^{s_0} \\ \beta_0^{s_1} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} \quad \text{and} \quad \xi_1 = \begin{bmatrix} 0\,1\,1\,0\,0 \\ 0\,1\,1\,0\,0 \\ 0\,1\,0\,1\,0 \\ 0\,0\,0\,1\,1 \\ 0\,0\,0\,1\,0 \end{bmatrix}$$

E.g. row 1 of ξ_1 indicates which terms are required to compute α_n.

In General: The key to generalizing the above method is the set of predecessors for each state and how each term should be computed using the predecessor terms. Not shown in this example is the case where a state q is both an initial state and an accepting state. In that case α_0^q is 1. Also there is at most one transition between two states in this example. In general, there may be multiple transitions between two states (with different labels). In that case, the equations will include factors in front of the α and β terms. In particular,

$$\beta_{n+1}^{q'} = \sum_{q \in Q} t_{q,q'} \cdot \beta_n^q$$

where $t_{q,q'}$ is the number of transition labels that transition from q to q' (Note: $t_{q,q'}$ is 0 if q is not a predecessor of q'). Likewise:

$$\alpha_{n+1}^{q'} = \sum_{q \in Q} (t_{q,q'} \cdot \alpha_n^q) + (t_{q,q'}^* \cdot \beta_n^q)$$

where $t_{q,q'}^*$ is $t_{q,q'}$ when q' is an accepting state and 0 otherwise.

Now we explain the recurrence relation extraction algorithm in general. Let $P = M||f$ be the synchronous product of some finite LTS M and some DFA f. We explain how to extract both the recurrence matrix ξ and the initial condition vector v from P.

In what follows, we assume that P has N states indexed by the set $\{1, ..., N\}$. We first define a matrix that encodes the transition relation of P:

Definition 5. *We define the* $N \times N$ predecessor matrix, *denoted* $\mathbf{D}$, *by its entries. We denote the entry in the ith row and jth column as* $\mathbf{D}_{ij}$. *Define* $\mathbf{D}_{ij}$ *to be the number of transitions from state j to state i in P.* □

Next, we define a matrix that encodes the accepting states of P:

Definition 6. *We define the* $N \times N$ accepting matrix, *denoted* $\mathbf{A}$, *so that* $\mathbf{A}_{ij} = \mathbf{D}_{ij}$ *if state i of P is an accepting state. Otherwise,* $\mathbf{A}_{ij} = 0$. □

We are now able to define the recurrence matrix ξ:

Definition 7. *The* recurrence matrix *of P is the* $(2N + 1) \times (2N + 1)$ *matrix*

$$\xi = \begin{bmatrix} 0 & \hat{1} & \hat{0} \\ \hat{0} & \mathbf{D} & \mathbf{A} \\ \hat{0} & \mathbf{0} & \mathbf{D} \end{bmatrix}$$

where $\hat{0}$ and $\hat{1}$ are n-dimensional vectors of 0's and 1's respectively and where $\mathbf{0}$ *is an $n \times n$ matrix of 0's.* □

We now explain how to extract the initial condition vector v from P. We first introduce some notation. For convenience, we vectorize the α_n^q and β_n^q terms. Let $\widehat{\alpha}_n := (\alpha_n^1, ..., \alpha_n^N)^T$ and $\widehat{\beta}_n := (\beta_n^1, ..., \beta_n^N)^T$. Then, the two vectors $\widehat{\alpha}_0$ and $\widehat{\beta}_0$ capture the initial conditions of terms α_n^i and β_n^i in the recurrence relation, and we can construct the $2N+1$ dimensional vector v by combining $\widehat{\alpha}_0$ and $\widehat{\beta}_0$ along with $\alpha_\varnothing = 0$, namely, $v := (\alpha_\varnothing, \widehat{\alpha}_0, \widehat{\beta}_0)^T$.

The vectors $\widehat{\alpha}_0$ and $\widehat{\beta}_0$ are extracted from P as follows: (1) The ith entry of $\widehat{\alpha}_0$ is 1 if and only if state i of P is both an accepting state and an initial state. Otherwise, that entry of $\widehat{\alpha}_0$ is 0. (2) The ith entry of $\widehat{\beta}_0$ is 1 if and only if state i of P is an initial state. Otherwise, that entry of $\widehat{\beta}_0$ is 0.

The following two statements (proven in Appendix A.4 of [22]) capture the correctness of our reduction.

Theorem 1. *Let α and β be the recurrence relation terms for the product $M||f_i$, as constructed above. Then for all $n \geqslant 0$, $\xi_i^{n+1} v_i = \begin{bmatrix} \alpha_n \\ \widehat{\alpha}_{n+1} \\ \widehat{\beta}_{n+1} \end{bmatrix}$. And hence $(\xi_i^{n+1} v_i)_0 = \alpha_n = sum(f \odot M_n, i)$.* □

Corollary 1. *Let ξ_i and v_i be the recurrence matrices and initial condition vectors for the products $M||f_i$, for $i = 1, ..., d$, as constructed above. Then*

$$@_f(M) = \lim_{n \to \infty} h((\xi_1^{n+1} v_1)_0, (\xi_2^{n+1} v_2)_0, ..., (\xi_d^{n+1} v_d)_0)$$

□

4.2 Step 3: Matrix Analysis

Next we will discuss two methods for solving the matrix analysis problem. One of these methods is *symbolic* and the other *numerical*. We illustrate them by continuing with the example of Fig. 3. We have constructed $g_1(n) = (\xi_1^{n+1} v_1)_0$. For sake of example, let us assume that $\xi_1 = \xi_2$ and that $v_2 = \xi_1 v_1$, so $g_2(n) = g_1(n+1)$. Let us also assume that the aggregate function is represented by $h(x_1, x_2) = x_1/x_2$.

Symbolic Method: The first step of the symbolic method is to compute closed-form expressions for each g_i. Tools such as Mathematica can solve for this closed-form expression using Jordan decomposition [31]. We omit the details. The result in the case of the example is:

$$g_1(n) = \frac{1}{25 \cdot 2^{(1+n)}} \left(4\sqrt{5} k_1^n - 4\sqrt{5} c_1^n - 5 k_1^n n + 5\sqrt{5} k_1^n n - 5 c_1^n n - 5\sqrt{5} c_1^n n \right)$$

where $c_1 := 1 + \sqrt{5}$ and $k_1 := 1 - \sqrt{5}$. As mentioned, $g_2(n) = g_1(n+1)$.

Once we have the closed-form expressions, we can ask Mathematica to solve the limit; it does so easily: $\lim_{n\to\infty} g_1(n)/g_2(n) = 2/(1+\sqrt{5})$ (the reciprocal of the golden ratio). Tools such as Mathematica can solve a broad class of limits using, e.g., Gruntz's method [29].

Computing the Jordan decomposition is currently the bottleneck for the symbolic method. Our experiments with Mathematica suggest that it cannot compute the Jordan decomposition for even moderately sized matrices, the run-time being exponential in the dimension of the matrix. There have been several recent attempts to improve the state of the art in Jordan decomposition [28] and we are hopeful that this sub-problem will soon be feasible to compute for large matrices.

Numerical Method: In this method, we compute $h(g_1(K), g_2(K))$ for large K, which we call a *K-approximation*. Although we have not yet established an error bound on the difference between the K-approximation and the true value of the limit, the K-approximation appears to converge relatively quickly. For instance, in the case of Example 3, the K-approximation for $K = 15$ and $K = 20$ are 0.6180344 and 0.6180339 respectively, which do not differ until the seventh decimal place. Our current approach is to compute the K-approximation for, e.g., $K = 8192$ and $K = 9000$ and determine at which decimal place they differ to establish the precision of the K-approximation for $K = 9000$. We can also plot intermediate K-approximations against K.

A naive implementation of K-approximation does not scale. Instead, we use the standard *exponentiation by squaring* technique to quickly compute K-approximations for large K. For example, to compute ξ^{11} for some matrix ξ, it suffices to compute ξ^2, ξ^4, and ξ^8, since $\xi^{11} = \xi \cdot \xi^2 \cdot \xi^8$. Note that $\xi^4 = (\xi^2)^2$ and $\xi^8 = (\xi^4)^2$, hence the name *exponentiation by squaring*. We need only compute $\log K$ squares and combine them per the binary representation of K. Furthermore, in our implementation, we found that we needed large datatypes (128 bit) to represent the entries of the matrix. As matrix power for large datatypes appears to not be implemented in the linear algebra library we used (numpy), we implemented this operation ourselves.

Although the examples in this section used $h(x_1, x_2) = x_1/x_2$, our method generalizes to any aggregate conditionally represented by a scalar arithmetic function $h(x_1, x_2, ..., x_d)$. This generality holds because the g_i are constructed independently of one another and combined according to h. For instance, if we had $h(x_1, x_2, x_3) = (x_1 + x_2)/x_3$, we construct $g_3(n)$ as we did for g_1 and g_2. We then take the limit or approximation of $(g_1(n) + g_2(n))/g_3(n)$ rather than $g_1(n)/g_2(n)$.

Comparison: The symbolic method gives an exact, symbolic representation of the fitness score, but unfortunately does not yet scale well, as we shall see from the experiments in Sect. 5 that follows. The numerical approach on the other hand can compute in seconds an approximation of the fitness score. As we shall show, these approximations are precise enough to distinguish between systems of different fitness.

5 Case Studies

We evaluate our framework on three case studies, described in detail in the subsections that follow, and summarized in Table 1. The symbolic method did not terminate after an hour for the larger two case studies (2PC and ABP) due to limitations imposed by the state of the art in Jordan decomposition (c.f. Section 4.2). Therefore, Table 1 reports the results obtained by the numerical method.

In each case study we compute the fitness score for different system variants (column M). Column $|M|$ represents the size (total number of states) of the system being measured, which is the product of all distributed processes. Time refers to the total execution time, in seconds. Column $@_f(M_{8192})$ refers to the K-approximation of the fitness score with $K = 8192$, and likewise for $K = 9000$. As can be seen, the two approximations are very close within each row (identical up to at least the 3rd decimal point), which indicates convergence. The reason we report the fitness score for K = 8192 instead of another number, say K = 8000 or K = 8500, is efficiency: 8192 the largest power of two less than 9000, and in order to compute the fitness score for K = 9000 we need to compute it anyway for K = 8192. Our results can be reproduced using a publicly available artifact, which is structured, documented, and licensed for ease of repurposing [23].

Let us remark that in the 2PC and ABP case studies, the systems being measured were automatically generated by a distributed protocol synthesis tool, which is an improved version of the tool described in [5,6]. As our goal in this paper is fitness evaluation, we omit discussing the synthesis tool. But, as mentioned in the introduction, evaluation of automatically synthesized systems is a promising application of our framework.

All case studies use the $@_{avg}$ aggregate function. Additionally, we use three variations of the fitness function in Fig. 5. This parametric fitness function suggests the possibility of constructing a library of general, reusable fitness functions. Although it was straightforward to construct fitness functions for our purposes, this library would further reduce that burden for users.

In the rest of this section we provide further details on each case study. Some supporting figures and intermediate results are provided in Appendix A.5 of [22].

5.1 Case Study #1: Simple Communication Protocol

This section treats the communication protocol presented in Example 1. We instantiate the framework to measure the average rate at which send-ack sequences are executed and apply this instance of the framework to M and M' (Fig. 1). The python representations of all simple communication protocol

Table 1. A summary of the numerical method results of the three case studies.

case study	M	$\|M\|$	total time (sec.)	$@_f(M_{8192})$	$@_f(M_{9000})$
simple comm	good	3	0.0052	0.249970	0.249972
simple comm	bad	5	0.006	0.138165	0.138168
2PC	H	58	0.41	0.0833	0.0832
2PC	A1	30	0.25	0.07856	0.07857
2PC	A2	25	0.1	0.0833	0.0832
ABP	HH	144	9.1	0.016864	0.016859
ABP	HA	144	8.6	0.015435	0.015430
ABP	AH	144	8.7	0.015218	0.015212
ABP	AA	144	8.6	0.01391	0.01390

processes and fitness functions are available in `toy_automata.py` of the artifact [23].

Recall that $\Sigma = \{s, t, a\}$. Let $f_1(w)$:= 'the number of send-ack sequences of the form st^*a in w'. For instance (brackets [and] added for emphasis), $f_1(aat[sa][sta]as[stta]stt[sa]) = 4$. Additionally, let $f_2(w) := |w|$ (the length of w) and let the fitness function be $f := \langle f_1, f_2 \rangle$. The functions f_1, f_2 can be represented as the DFA shown in Fig. 5, with $L = \{s\}$ and $R = \{a\}$. This fitness function is measuring the number of send-ack sequences per unit of discrete time, which is analogous to the traditional measure of throughput in distributed systems.

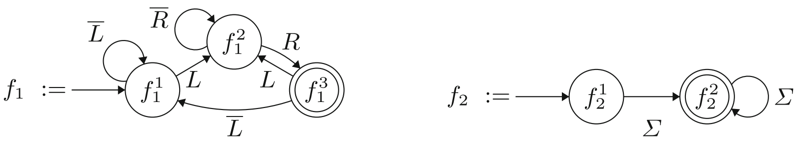


Fig. 5. The DFA representations of f_1 and f_2 for the case studies, parameterized by the set of labels Σ, as well as a set of *left endpoints* $L \subseteq \Sigma$ and *right endpoints* $R \subseteq \Sigma$. $\overline{L} = \Sigma \setminus L$ and likewise for $\overline{R}$.

As reported in Table 1, the system that uses the good receiver has a fitness score of about 0.25 and the system using the bad receiver a score of about 0.138. These scores are *interpretable* in that they have units: send-ack sequences per unit of discrete time. Hence, the framework deems the good receiver as more fit and this determination aligns with our intuitions. Because this example is relatively small, Mathematica was able to compute the exact fitness scores of these systems. The system that uses the good receiver has a fitness score of exactly 1/4 (obtained after 34 s) and the system that uses the bad receiver has a score of exactly $\frac{5-\sqrt{5}}{20} \approx 0.138$ (obtained after 563 s).

5.2 Case Study #2: Two Phase Commit (2PC)

Two phase commit (2PC) is a protocol for making transactional changes to a distributed database atomically; if one sub-operation of the transaction is aborted at one remote database, so too must the sub-operations at all other remote databases. Although each iteration of 2PC is terminating, it is typical to assume there will be infinitely many such iterations, and our model reflects this. In our model of 2PC, a user initiates a transaction by synchronizing with a *transaction manager* on the label x. The transaction is complete when the transaction manager synchronizes with the user on label *fail* or *succ*. We omit the details of the intermediate exchanges between the transaction manager and database managers. The python representations of all 2PC processes and fitness functions are available in `_2pc_automata.py` of the artifact [23].

The fitness function for this case study is as depicted in Fig. 5, with $L = \{x\}$, $R = \{\mathit{fail}, \mathit{succ}\}$, and Σ has a total of 18 labels. This fitness function measures the rate at which transactions are initiated and then completed.

We study three 2PC implementations, each using a different transaction manager LTS. The system labeled H in Table 1 uses a previously manually constructed transaction manager that the synthesis tool was also able to discover automatically, while the systems labeled A1 and A2 use new transaction managers generated by the synthesis tool. The automatically generated transaction managers have 12 states each and it is therefore hard to tell at a glance which will give rise to the most efficient protocol. Our tool automatically reports, in fractions of a second, a fitness score of about 0.083 for both systems H and A2, and a score of about 0.079 for system A1. These fitness scores have units: transactions per unit time. Hence, in the same amount of time, A1 completes about 5% fewer transactions than H or A2.

5.3 Case Study #3: Alternating Bit Protocol (ABP)

The Alternating Bit Protocol (ABP) allows reliable communication over an unreliable network. As with the prior two case studies, we use the fitness function depicted in Fig. 5, except with $L = \{\mathit{send}\}, R = \{\mathit{done}\}$, and Σ of size 12. Similar to case study #1 we are measuring the rate of send-done sequences. The python representations of all ABP processes and fitness functions are available in `abp_automata.py` of the artifact [23].

In [6], the authors present a method to automatically synthesize (distributed) ABP sender and receiver processes. Here, we evaluate the fitness of the ABP variants that use these various synthesized processes. Together the synthesized sender and receiver processes have 14 states, which again makes manual determinations about the fitness very challenging—even more so due to the distributed nature of the problem. It is no longer necessarily a question of which sender or receiver is better than the other sender or receiver, but a question of which combination of sender and receiver is best. Once again, our framework allows to automatically make this determination in a matter of seconds.

The systems are ranked by fitness in the following order: HH, HA, AH, AA. H stands for human-designed (and then also rediscovered during synthesis) and A stands for newly discovered during synthesis. The first position is for the sender process and the second for the receiver. In this case study, the newly discovered processes do worse than the manually constructed processes. The difference in fitness scores is meaningful: in the same amount of time, AA will complete about 18% fewer sequences on average. AH and HA will both complete about 8.5% fewer sequences than HH.

6 Related Work

Our work is broadly related to the field of performance analysis and evaluation. Mathematical models typically used there include Markov Chains, Markov Decision Processes, Markov Automata, queueing models, Petri nets, timed or hybrid automata, etc., e.g., see [9,15–17,25,34–36]. Our approach differs as our mathematical framework uses neither timed nor probabilistic models such as the ones above. Because we do not use stochastic models, our work is also different from the work on probabilistic verification, e.g., see [8–10,18,33]. Our work also differs from performance analysis approaches that use max-plus algebra based frameworks such as the real-time calculus, e.g., see [30,38,44,45].

Our work is also related to non-boolean interpretations of temporal semantics, such as the 5-valued robust temporal logic rLTL [7,43]. However, our motivation is performance comparisons rather than robustness. Our framework also differs from that of signal temporal logic (STL) [11,12,27,39–42], which is valued over real-time traces. Our framework is over discrete traces, although there have been recent STL extensions which handle both real and discrete time [26]. In addition, our framework is parameterized by generic quantitative concepts (the fitness and aggregate functions and the comparison relation) that are present neither in rLTL nor in STL or its variants.

Our work is closely related to the field of quantitative verification, synthesis, and games, e.g., see [1,2,13,14,19–21,32]. Typically, these works assign values to *weighted automata*. These automata blend in a single model both the description of the system and the description of any performance or fitness functions associated with the system. In comparison, our framework decouples the description of the system (e.g., a plain LTS without any weights) from the description of the fitness function (e.g., a DFA). These works support aggregates like sup while our framework is defined for more general aggregates, including averages.

Sensing cost [4] and propositional quality [3] are two other ways to measure the fitness of a system. Sensing cost is a specific measure of fitness, whereas our framework is a more general setting. The work on propositional quality is quite general, like our work, but it uses a quantitative variant of LTL to assign scores rather than DFA. This logic induces a sort of recursive computation that can never be captured by a DFA. The logic is limited though in that it can only characterize finite chunks of a trace at one time (and no limit is taken), whereas our characterization applies to the infinite trace after taking a limit. Hence propositional quality and our fitness evaluation are fundamentally distinct.

7 Conclusions and Future Work

We proposed a formal framework that assigns *fitness scores* to systems modeled as finite LTSs. The main novelty of our framework is that it *decouples* the description of the system from the set of domain-specific parameters such as fitness and aggregate functions, which determine the final fitness score. Furthermore, the user defines these fitness scores and aggregate functions over partial runs, which are easier for the user to reason about—our framework does the heavy lifting of extending this reasoning to infinite traces. This decoupling and finite reasoning make our framework more useable and its results more *interpretable*. Indeed, in all of our case studies the scores are not merely numbers; they have meaningful units, e.g., send-ack sequences per unit of time.

We used our framework to evaluate the automatically synthesized ABP protocols presented in [6] as well as our own automatically synthesized 2PC protocols. We showed that some of these protocols are better than others. Inspired by this application, we plan to investigate the use of our framework in protocol synthesis, specifically in synthesizing protocols that not only satisfy a given correctness specification but are also *optimal* with respect to a fitness score, i.e., *fitness-optimal synthesis* (c.f. page 8).

We are also actively exploring ways to improve the scalability of the symbolic method. In particular, we may be able to feasibly compute a simplified version of the recurrence matrix ξ_i without sacrificing the accuracy of the final computed limit. Additionally, we would like to generalize our method to aggregates like $\min / \max$, which do not have conditional representations, and to systems that cannot be represented as finite labeled transition systems. We suspect that best/worst-case analysis reduces to the minimal cost-to-time ratio problem [37], but in general aggregates with no conditional representation may be more challenging.

Acknowledgements. Derek Egolf's research has been initially supported by a Northeastern University PhD fellowship. This material is based upon work supported by the National Science Foundation Graduate Research Fellowship under Grant No. (1938052). Any opinion, findings, and conclusions or recommendations expressed in this material are those of the authors(s) and do not necessarily reflect the views of the National Science Foundation. We thank the anonymous reviewers for their helpful comments and feedback.

References

1. de Alfaro, L., Faella, M., Henzinger, T.A., Majumdar, R., Stoelinga, M.: Model checking discounted temporal properties. Theor. Comput. Sci. **345**(1), 139–170 (2005)
2. Almagor, S., Alur, R., Bansal, S.: Equilibria in quantitative concurrent games. eprint arXiv:1809.10503 (2018)

3. Almagor, S., Boker, U., Kupferman, O.: Formalizing and reasoning about quality. In: Fomin, F.V., Freivalds, R., Kwiatkowska, M., Peleg, D. (eds.) ICALP 2013. LNCS, vol. 7966, pp. 15–27. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39212-2_3
4. Almagor, S., Kuperberg, D., Kupferman, O.: Regular sensing. In: FSTTCS. LIPIcs, vol. 29. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik (2014)
5. Alur, R., Martin, M., Raghothaman, M., Stergiou, C., Tripakis, S., Udupa, A.: Synthesizing finite-state protocols from scenarios and requirements. In: Yahav, E. (ed.) HVC 2014. LNCS, vol. 8855, pp. 75–91. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-13338-6_7
6. Alur, R., Tripakis, S.: Automatic synthesis of distributed protocols. SIGACT News **48**(1), 55–90 (2017)
7. Anevlavis, T., Philippe, M., Neider, D., Tabuada, P.: Being correct is not enough: efficient verification using robust linear temporal logic. ACM Trans. Comput. Log. **23**(2), 8:1–8:39 (2022)
8. Baier, C., Haverkort, B.R., Hermanns, H., Katoen, J.P.: Performance evaluation and model checking join forces. Commun. ACM **53**(9), 76–85 (2010)
9. Baier, C., Katoen, J.P.: Principles of Model Checking. MIT Press, Cambridge (2008)
10. Baier, C., de Alfaro, L., Forejt, V., Kwiatkowska, M.: Model checking probabilistic systems. In: Handbook of Model Checking, pp. 963–999. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-10575-8_28
11. Beg, O.A., Nguyen, L.V., Johnson, T.T., Davoudi, A.: Signal temporal logic-based attack detection in DC microgrids. IEEE Trans. Smart Grid **10**(4), 3585–3595 (2019)
12. Bortolussi, L., Gallo, G.M., Křetínský, J., Nenzi, L.: Learning model checking and the kernel trick for signal temporal logic on stochastic processes. In: Learning model checking and the kernel trick for signal temporal logic on stochastic processes. LNCS, vol. 13243, pp. 281–300. Springer, Cham (2022). https://doi.org/10.1007/978-3-030-99524-9_15
13. Bouyer, P., Gardy, P., Markey, N.: Quantitative verification of weighted kripke structures. In: Cassez, F., Raskin, J.-F. (eds.) ATVA 2014. LNCS, vol. 8837, pp. 64–80. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11936-6_6
14. Brihaye, T., Geeraerts, G., Haddad, A., Monmege, B., Pérez, G.A., Renault, G.: Quantitative games under failures. In: FSTTCS. Leibniz International Proceedings in Informatics (LIPIcs), vol. 45, pp. 293–306. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik (2015)
15. Bucci, G., Sassoli, L., Vicario, E.: A discrete time model for performance evaluation and correctness verification of real time systems. In: 10th International Workshop on Petri Nets and Performance Models, 2003. Proceedings, pp. 134–143 (2003)
16. Bucci, G., Sassoli, L., Vicario, E.: Correctness verification and performance analysis of real-time systems using stochastic preemptive time petri nets. IEEE Trans. Softw. Eng. **31**(11), 913–927 (2005)
17. Cassandras, C.G., Lafortune, S.: Introduction to Discrete Event Systems, 3rd edn. Springer (2021). https://doi.org/10.1007/978-0-387-68612-7
18. Cauchi, N., Hoque, K.A., Abate, A., Stoelinga, M.: Efficient probabilistic model checking of smart building maintenance using fault maintenance trees. eprint arXiv:1801.04263 (2018)
19. Černý, P., Chatterjee, K., Henzinger, T.A., Radhakrishna, A., Singh, R.: Quantitative synthesis for concurrent programs. In: Gopalakrishnan, G., Qadeer, S. (eds.)

CAV 2011. LNCS, vol. 6806, pp. 243–259. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_20
20. Chatterjee, K., Doyen, L., Henzinger, T.A.: Quantitative languages. ACM Trans. Comput. Log. **11**(4) (2010)
21. Chatterjee, K., de Alfaro, L., Faella, M., Henzinger, T.A., Majumdar, R., Stoelinga, M.: Compositional quantitative reasoning. In: QEST, pp. 179–188. IEEE Computer Society (2006)
22. Egolf, D., Tripakis, S.: Decoupled fitness criteria for reactive systems. eprint arXiv: 2212.12455 (2023)
23. Egolf, D., Tripakis, S.: Decoupled Fitness Criteria for Reactive Systems (Artifact, SEFM 2023) (2023). https://doi.org/10.5281/zenodo.8168367
24. Egolf, D., Tripakis, S.: Synthesis of distributed protocols by enumeration modulo isomorphisms. In: ATVA. Springer (2023)
25. Fakih, M., Grüttner, K., Fränzle, M., Rettberg, A.: Towards performance analysis of SDFGs mapped to shared-bus architectures using model-checking. In: DATE, pp. 1167–1172. EDA Consortium San Jose, CA, USA/ACM DL (2013)
26. Ferrère, T., Maler, O., Ničković, D.: Mixed-time signal temporal logic. In: André, É., Stoelinga, M. (eds.) FORMATS 2019. LNCS, vol. 11750, pp. 59–75. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-29662-9_4
27. Finkbeiner, B., Fränzle, M., Kohn, F., Kröger, P.: A truly robust signal temporal logic: monitoring safety properties of interacting cyber-physical systems under uncertain observation. Algorithms 15(4) (2022)
28. Ghabbour, R.R., Abdelgaliel, I.H., Hanna, M.T.: A directed graph and MATLAB generation of the Jordan canonical form for a class of zero-one matrices. In: ICENCO, vol. 1, pp. 86–91 (2022)
29. Gruntz, D.W.: On Computing Limits in a Symbolic Manipulation System. Ph.D. thesis (1996)
30. Guan, N., Yi, W.: Finitary real-time calculus: efficient performance analysis of distributed embedded systems. In: RTSS, pp. 330–339 (2013)
31. Hefferon, J.: Linear Algebra, pp. 440-463 (2020). https://hefferon.net/
32. Henzinger, T.A.: Quantitative reactive modeling and verification. Comput. Sci. Res. Dev. **28**(4), 331–344 (2013). https://doi.org/10.1007/s00450-013-0251-7
33. Jansen, N., et al.: Accelerating parametric probabilistic verification. In: Norman, G., Sanders, W. (eds.) QEST 2014. LNCS, vol. 8657, pp. 404–420. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-10696-0_31
34. Kempf, J.-F., Bozga, M., Maler, O.: Performance evaluation of schedulers in a probabilistic setting. In: Fahrenberg, U., Tripakis, S. (eds.) FORMATS 2011. LNCS, vol. 6919, pp. 1–17. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-24310-3_1
35. Kwiatkowska, M.Z., Norman, G., Parker, D., Sproston, J.: Performance analysis of probabilistic timed automata using digital clocks. Formal Methods Syst. Des. **29**(1), 33–78 (2006)
36. Larsen, K.G.: Automatic verification, performance analysis, synthesis and optimization of timed systems. In: TIME, pp. 1–1 (2016)
37. Lawler, E.L.: Optimal cycles in graphs and the minimal cost-to-time ratio problem. Tech. Rep. UCB/ERL M343, EECS Department, UC, Berkeley (1972)
38. Lu, Q., Madsen, M., Milata, M., Ravn, S., Fahrenberg, U., Larsen, K.G.: Reachability analysis for timed automata using max-plus algebra. J. Logic Algebraic Program. **81**(3), 298–313 (2012)

39. Ničković, D., Lebeltel, O., Maler, O., Ferrère, T., Ulus, D.: AMT 2.0: qualitative and quantitative trace analysis with extended signal temporal logic. Int. J. Softw. Tools Technol. Transfer **22**(6), 741–758 (2020). https://doi.org/10.1007/s10009-020-00582-z
40. Prabhakar, P., Lal, R., Kapinski, J.: Automatic trace generation for signal temporal logic. In: RTSS, pp. 208–217 (2018)
41. Puranic, A.G., Deshmukh, J.V., Nikolaidis, S.: Learning from demonstrations using signal temporal logic. eprint arXiv:2102.07730 (2021)
42. Salamati, A., Soudjani, S., Zamani, M.: Data-driven verification of stochastic linear systems with signal temporal logic constraints. Automatica **131**, 109781 (2021)
43. Tabuada, P., Neider, D.: Robust linear temporal logic. In: EACSL, LIPIcs, vol. 62. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2016)
44. Thiele, L., Chakraborty, S., Naedele, M.: Real-time calculus for scheduling hard real-time systems. In: ISCAS, pp. 101–104 (2000)
45. Wandeler, E., Thiele, L.: Performance analysis of distributed embedded systems. In: Embedded Systems Handbook. CRC Press (2005)