Quantitative Abstraction Refinement *

Pavol Černý Thomas A. Henzinger Arjun Radhakrishna
IST Austria

Abstract

We propose a general framework for abstraction with respect to quantitative properties, such as worst-case execution time, or power consumption. Our framework provides a systematic way for counter-example guided abstraction refinement for quantitative properties. The salient aspect of the framework is that it allows anytime verification, that is, verification algorithms that can be stopped at any time (for example, due to exhaustion of memory), and report approximations that improve monotonically when the algorithms are given more time.

We instantiate the framework with a number of quantitative abstractions and refinement schemes, which differ in terms of how much quantitative information they keep from the original system. We introduce both state-based and trace-based quantitative abstractions, and we describe conditions that define classes of quantitative properties for which the abstractions provide over-approximations. We give algorithms for evaluating the quantitative properties on the abstract systems. We present algorithms for counter-example based refinements for quantitative properties for both state-based and segment-based abstractions. We perform a case study on worst-case execution time of executables to evaluate the anytime verification aspect and the quantitative abstractions we proposed.

Categories and Subject Descriptors D.2.4 [*Software/Program Verification*]

General Terms Theory, Verification

Keywords abstraction, refinement, quantitative analysis

1. Introduction

The quantitative analysis of systems is gaining importance due to the spread of embedded systems with requirements on resource consumption and timeliness of response. Quantitative analyses have been proposed for properties such as worst-case execution time (see [19] for a survey), power consumption (pioneered in [17]), and prediction of cache behavior for timing analysis (see, for example, [9]).

Anytime algorithms (see [1]) are algorithms that generate imprecise answers quickly and proceed to construct progressively better approximate solutions over time, eventually finding the correct solution. Anytime algorithms are useful in verification, as they offer a way to deal with the state-space explosion problem — if the algorithm is terminated early, for example due to memory exhaustion, it is still able to report an approximation of the desired result. The term *anytime verification* has been proposed [16] recently in the context of verifying boolean properties, as a way to get (non-quantifiably) better estimates on whether a property holds for a system. The anytime concept, however, is particularly well-suited in the context of *quantitative* verification. In this context, abstraction gives a quantitative over-approximation of the quantitative answer to a verification question, and it is natural to require the anytime property: the more time the verification run is given, the (quantifiably) better the over-approximation of the correct answer should be. We implement this anytime property for quantitative verification through an abstraction refinement scheme that monotonically improves the answer. For instance, abstraction refinement may compute increasingly better approximations of power consumption of a system.

We propose a framework for abstraction and abstraction refinement for quantitative properties that is suitable for anytime verification. We explain the motivation and intuition behind the framework using the following example.

Motivating example. Consider the problem of estimating the worst-case execution time (WCET) of the program in Figure 1. We assume an idealized situation where the performance is affected mostly by the cache behavior. Let each program statement have a cost depending on whether it accesses only the cache (or no memory at all), for a cost of 1, or main memory for a cost of 25. we assume that the program variables `i,a,v` are mapped to different cache entries, while `b` and `c` are mapped to the same entry (different from the entries for the other variables). We consider abstractions that abstract only the cache (not the program). The cache is abstracted by an *abstract cache* with a smaller number of entries (accesses to the entries not tracked in the abstract cache are always considered to be a cache miss). Let us start the analysis with an abstract cache of size 2 which caches variables `i` and `v`. In the abstract system (i.e., the original program composed with the abstract cache), the worst-case trace has `v` equal to 1, and the program accesses the (uncached) variable `a` 16 times. The analysis then uses this trace to refine the abstraction. The refinement extends the cache to include the cache entry for `a`. The worst-case execution then has v equal to 2, and it has 8 accesses to `b`. The analysis now refines the abstraction by extending the cache with an entry for `b`.

```
a,b,c,i,v: int; input v;
if (v == 1)
 for (i=0;i<16;i++) read(a);
else if(v == 2)
 for (i=0;i<16;i++)
  if (i mod 2 = 0) read(b);
else
 for (i=0;i<16;i++)
  if (i mod 4 = 0) read(c);
```

Figure 1: Example 1

* This research was supported in part by the European Research Council (ERC) Advanced Investigator Grant QUAREM and by the Austrian Science Fund (FWF) project S11402-N23.

POPL'13, January 23–25, 2013, Rome, Italy.

The WCET estimate is thus tightened, until either the highest-cost trace corresponds to a real execution (and thus the WCET estimate is precise), or the analysis runs out of resources and reports the computed over-approximation.

Abstraction for quantitative properties. Our model of systems is weighted transition systems. We provide a way of formalizing quantitative properties of systems which capture important properties studied in literature, including *limit-average*, *discounted-sum*, and boolean properties such as safety and liveness. The framework makes it possible to investigate quantitative versions of the boolean properties: for instance, for safety one could ask not only if an error state is reached, but also how often it is reached. We focus on properties that admit a linear trace that maximizes (or minimizes) the value of the quantitative property. Such a trace is called the *extremal trace* (ext-trace for short).

We present two types of quantitative abstraction schemes. The first is *state-based*, that is, the elements of the abstract domain correspond to sets of states. The second is *segment-based*, that is, the elements of the abstract domain correspond to sets of trace segments.

State-based quantitative abstractions. The abstraction scheme $ExistMax$ is state-based. It is a direct extension of predicate abstraction, where each abstract state corresponds to an equivalence class of concrete states. In addition, with each abstract state, $ExistMax$ stores the maximum weight of the corresponding concrete states. We give conditions for the class of quantitative properties for which $ExistMax$ is a *monotonic over-approximation* (that is, it provides better estimates as the underlying equivalence relation on states is refined). This class includes all the quantitative properties mentioned above. However, we show that there are naturally defined properties for which $ExistMax$ is not a monotonic over-approximation. This is in contrast to the abstraction refinement of boolean properties (in the boolean case, we do not get less precise invariants if we add more predicates).

Segment-based quantitative abstractions. We introduce a number of segment-based abstraction schemes. A *segment* is a (finite or infinite) sequence of states that is a consecutive subsequence of an execution trace. As the quantitative properties we consider accumulate quantities along (infinite) traces, it is natural, and advantageous, to consider abstract domains whose elements correspond to sets of segments, not sets of states. This is similar to termination and liveness analysis using transition predicates [14], and their generalizations to segment covers [7]. We build upon these approaches to develop our quantitative abstractions. The $PathBound$ abstraction scheme stores with each abstract state t (representing a set of segments) (i) $minp(t)$, the length of a shortest finite segment in t, (ii) $maxp(t)$, the length of the longest finite segment in t, (iii) $hasInfPath(t)$ a bit that is true if t contains a segment of infinite length, and (iv) $val(t)$, a summary value of the weights of the states. Defining $val(t)$ as the maximal weight of a state occurring in one of the segments in t makes $PathBound$ a sound over-approximation for a general class of quantitative properties. To get better approximations for particular quantitative properties $val(t)$ can be a different summary of the weights. For instance, we specialized $PathBound$ to limit average by storing not the maximal occurring value, but the maximal average of values along a segment.

In order to compare state-based and segment-based abstractions, let us consider the limit-average property applied to a program with a simple `for` loop for which the loop bound is statically known to be 10. Let us assume that the cost of the operations inside the loop is much greater than the cost of the operations outside the loop. Now consider the state-based abstraction $ExistMax$ with an abstract state t that groups together all states whose control location is in the `for` loop. In the $ExistMax$ abstraction, this abstract state has a self-loop. Analyzing the abstract system would conclude that the highest-cost trace is the one which loops forever in t. This would be a very imprecise result, as the concrete traces all leave the loop after 10 iterations. To correct for this, the loop would have to be unrolled 10 times, using 10 counterexample-guided refinement steps. On the other hand, consider the $PathBound$ abstraction, with an abstract state t representing all the segments in the loop. For t, $hasInfPath(t)$ is false and $maxp(t) = 10$, allowing immediately for more precise estimates.

The $PathBound$ abstraction scheme is a sound abstraction in the sense that the quantitative value we obtain by analyzing the abstract system over-approximates the value for the concrete system. As in $ExistMax$, this holds for a large class of quantitative properties. However, we show that $PathBound$ is not a monotonic over-approximation even for standard quantitative properties such as the limit-average property in the sense that after refinement of the abstract states, we may get worse estimates. We therefore present a hierarchical generalization of $PathBound$ called $HPathBound$. In $PathBound$, each abstract state represents a set of segments, and stores some quantitative characteristics (such as $minp$, $maxp$) of that set. In order to compute better estimates of these quantitative characteristics, one can perform another level of refinement, within abstract states. This leads to the idea of a hierarchical abstraction. It is particularly useful for software, which already has a hierarchical structure in many cases (e.g., nested loops, function calls). This approach corresponds to the (multi-level) abstract inductive segment cover of [7, Section 16.1].

Refinement of state-based abstractions for quantitative properties. For the state-based abstraction scheme $ExistMax$, we give an algorithm for counterexample-guided abstraction refinement (CEGAR) for quantitative properties. The algorithm is based on the classical CEGAR algorithm [2], which we extend to the quantitative case. In the classical CEGAR loop, the counterexample to be examined is chosen using heuristics. For quantitative properties, it is clear that an extremal counterexample trace (the ext-trace) should be chosen for refinement. The reason is that if the ext-trace does not correspond to a real trace, then a refinement which does not eliminate this trace would have the same value as the previous abstract system.

Refinement of segment-based abstractions for quantitative properties. We propose a refinement algorithm for the segment-based abstraction $HPathBound$. We chose $HPathBound$ because, as discussed above, it is particularly suitable for software, and it is a monotonic over-approximation for a large set of quantitative properties. An abstract counterexample for $HPathBound$ is a hierarchical trace. Given an extremal abstract counterexample which does not correspond to a concrete counterexample, the abstract counterexample is traversed, similarly as in the classical CEGAR algorithm, until an abstract segment that provides values that are too "pessimistic" (e.g., $maxp$ and val values which cannot be achieved in the concrete system by a concretization of the abstract counterexample) is found. This abstract segment is then refined. Note that the fact that the counterexample is hierarchic gives freedom to the traversal algorithm — at each step, it can decide whether or not to descend one level lower and to find a mismatch between a concrete and abstract execution there.

Experimental results In order to evaluate the proposed abstraction schemes, both state-based and segment-based, we performed a case study on worst-case execution time analysis of x86 executables. We focused on one aspect, the cache behavior analysis, and in particular, on estimating the rate of cache misses over the course of the worst-case execution. In order to abstract the cache, we used the abstractions introduced in [9]. To the best of our knowledge, this is the first work on automated refinement for these abstractions.

We implemented two abstraction schemes: the state-based $ExistMax$ and the segment-based $HPathBound$. We performed the case study on our own (small) examples, and on some of the benchmarks collected in [11]. The experiments show that we obtain more precise quantitative results as the abstraction is refined, for example, by having a larger abstract cache. Furthermore, we show that using the segment-based abstraction $HPathBound$ enables scaling up. This is due to the fact that in the presence of loops, the $HPathBound$ abstraction can quickly obtain good over-approximations if it can statically over-approximate loop bounds, whereas the $ExistMax$ abstraction would have to unroll the loop many times to get comparable results. Similarly to the $ExistMax$ case, the experiments show that we obtain more precise quantitative results as the $HPathBound$ abstraction is (hierarchically) refined by computing better estimates for loop bounds. The running time of the analysis was under 35 seconds in all cases.

Related work. The theory of abstractions for programs was introduced in [5]. We build on the transition predicates of [14] and segment covers of [7] to construct our quantitative abstractions. The CEGAR algorithm was introduced in [2] and is widely used. Automated abstraction refinement for transition predicates was presented in [4]. To the best of our knowledge, CEGAR-like algorithms for quantitative properties have not yet been studied.

However, quantitative abstractions and refinements have been introduced for *stochastic* systems [8, 12, 13], where the need for quantitative reasoning arises because of stochasticity. They are mainly directed towards the estimation of expected values, and the algorithms reflect this fact. The probabilistic work does not aim at handling accumulative properties like the limit-average property.

Abstractions (but not CEGAR-like algorithms) have been proposed for certain quantitative properties of non-probabilistic systems, such as cache abstractions for WCET analysis [9]. WCET analysis using interval abstraction was performed in [15].The power consumption analysis of software based on the costs of single instructions were presented in [17]. Our WCET analysis of executables based on quantitative abstraction refinement could be adapted for power consumption analysis using these models.

2. Quantitative properties

A *weighted transition system* (WTS) is defined by a tuple $S = (Q, \delta, \varrho, q_\iota)$, with a (finite or infinite) set of states Q, a transition set $\delta \subseteq Q \times Q$, weight function $\varrho : Q \to \mathbb{R}$, and an initial state $q_\iota \in Q$. Let $\mathcal{S}$ denote the set of all WTSs and let $\mathcal{S}(\mathcal{Q}) \subseteq \mathcal{S}$ be the set of all WTSs with a set of states Q such that $Q \subseteq \mathcal{Q}$. We assume that $\forall q \in Q : \exists q' : (q, q') \in \delta$. System S_1 in Figure 2 is an example of a WTS.

A *trace* of a WTS S is an infinite sequence $\pi = q_0 q_1 \ldots$ with $q_0 = q_\iota$, and $\forall i \geq 0 : (q_i, q_{i+1}) \in \delta$. The set of all traces of S is denoted by $\Pi(S)$. We extend the weight function ϱ to traces as $\varrho(q_0 q_1 \ldots) = r_0 r_1 \ldots \in \mathbb{R}^\omega$ where $r_i = \varrho(q_i)$, and to sets of traces as $\varrho(T) = \{w \mid \exists \pi \in T : w = \varrho(\pi)\}$.

A trace $\pi = q_0 q_1 \ldots$ is *memoryless* if for all $i, j \geq 0$, we have $q_i = q_j \to (q_{i+1} = q_{j+1})$. For instance, system S_1 in Figure 2 has two memoryless traces, a trace π_1 with $\varrho(\pi_1) = (1\ 10\ 4\ 3\ 1\ 1\ 1)^\omega$, and a trace π_2 with $\varrho(\pi_2) = (1\ 10\ 4\ 3\ 1\ 1\ 1\ 1)^\omega$.

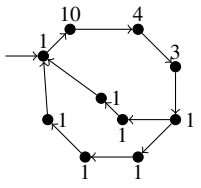


Figure 2: System S_1

A *quantitative property* $f : \mathcal{S} \to \mathbb{R}$ is defined using two functions: a *trace value* function f_t and a *system value* function f_s. The trace value function $f_t : \mathbb{R}^\omega \to \mathbb{R}$ summarizes a single trace of the system. The system value functions $f_s : 2^{\mathbb{R}} \to \mathbb{R}$ summarizes the set of f_t values of all traces of the system. We then have $f(S) = f_s(\{f_t(r) \mid \exists \pi \in \Pi(S) : \varrho(\pi) = r\})$. We extend f_t to $2^{\mathbb{R}^\omega}$ by letting $f_t(T) = \{f_t(r) \mid r \in T\}$. Hence, we have that $f(S) = f_s(f_t(\varrho(\Pi(S))))$.

We present a number of common quantitative properties below. We also model some classical boolean properties in our framework. First, we give examples of trace value functions:

Limit-average. The limit-average trace value function measures the maximal long-term average weight over a trace. The function $limavg : \mathbb{R}^\omega \to \mathbb{R}$ is defined by $limavg(r_0 r_1 \ldots) = \liminf_{i\to\infty} \frac{1}{i} \cdot \sum_{k=0}^{i-1} r_k$. For the trace π_1 of S_1, the value $limavg(\varrho(\pi_1)) = \frac{21}{7} = 3$.

Discounted-sum. The discounted-sum trace value function accumulates the weights over a trace where weights occurring further along the trace are discounted by a factor $\lambda \in (0, 1)$. Formally, the function $disc_\lambda(r_0 r_1 \ldots) = \sum_{i=0}^{\infty} \lambda^i \cdot r_i$.

Safety and quantitative safety. The classical safety property is defined by a set of unsafe states U, and a trace is safe if it never visits a state in U. To model a safety property, we assign the weight 1 to each state in U and the weight 0 to all other states. The safety trace value function is defined as $safety(r_0 r_1 \ldots) = \sup r_i$. Now, for any trace π, we have $safety(\varrho(\pi)) = 0$ if and only if π is safe. Note that this modeling of safety is still boolean, because the trace value can only be either 0 or 1. In some applications, a more quantitative view might be useful. Consider $safety_\lambda$ defined by $safety_\lambda(r_0 r_1 \ldots) = \lim_{n\to\infty} \sum_{k=1}^{n-1} \lambda^k \cdot r_k$ (discounted sum of the number of 1's in the run). For a safe trace π, we still have that $safety_\lambda(\varrho(\pi)) = 0$. However, for an unsafe trace, $safety_\lambda(\varrho(\pi))$ gives a more fine-grained picture: it measures how early and often the bad states are visited.

Liveness and quantitative liveness. The liveness property is defined by a set of Büchi (live) states L, and a trace is live if it visits states in L infinitely often. To model a liveness property, we assign weight 0 to each state in L and weight 1 to all other states. The trace value function $live(r_0 r_1 \ldots) = \liminf_{i\to\infty} r_i$. This function is faithful to the liveness property, i.e, $live(\varrho(\pi)) = 0$ if and only if the trace is live. As in the safety case, we can get a finer view of liveness by measuring how often states in L are visited. We do it by letting $live(r_0 r_1 \ldots) = \liminf_{i\to\infty} \frac{1}{i} \cdot \sum_{k=0}^{i-1} r_i$.

Second, we give examples of standard system value functions.

- *Supremum and Infimum.* Intuitively, the supremum and infimum functions measure the worst-case and best-case traces in the system.
- *Threshold.* The threshold function checks if any of the values in the given set are above or below given thresholds. Formally, $threshold_u(R) = 1$ if $\exists r \in R : r \geq u$, and $threshold_u(R) = 0$ otherwise.

Any combination of a trace value function and a system value function defines a quantitative property. In this paper, we implicitly assume that the system value function for any quantitative property is sup unless otherwise mentioned.

A trace π is an *extremal counterexample trace* (or ext-trace for short) if $f(S) = f_s(\{f_t(\varrho(\pi))\})$. We restrict ourselves to a class of quantitative properties that admit *memoryless extremal counterexample traces*, i.e., every system S has a extremal counterexample trace that is memoryless. Formally, f is *memoryless* if and only if $\forall S \in \mathcal{S} : \exists \pi \in \Pi(S) : f(S) = f_s(\{f_t(\varrho(\pi))\})$. Note that all properties mentioned above are memoryless.

3. State-based quantitative abstractions

A *quantitative abstraction* $C = (\mathcal{S}^C, f^C, \alpha^C)$ is triple consisting of a set of abstract systems $\mathcal{S}^C$, an abstract quantitative property $f^C : \mathcal{S}^C \to \mathbb{R}$ and an abstraction function $\alpha^C : \mathcal{S} \to \mathcal{S}^C$. A quantitative abstraction C is an *over-approximation* of f, if for all $S \in \mathcal{S}$, $f^C(\alpha^C(S)) \geq f(S)$.

3.1 *ExistMax* abstraction.

In this section, we present a quantitative abstraction technique based on state abstractions. In this case, the abstract system is a WTS whose states are sets of states of the concrete systems. In particular, our abstraction scheme *ExistMax* is a direct extension of the classical predicate abstraction. Compared to predicate abstraction, *ExistMax* additionally stores, with each abstract state, the maximum weight occurring in the set of corresponding concrete states. The name *ExistMax* refers to transitions abstracted existentially, and that the weights abstracted by maxima.

ExistMax is a state based abstraction scheme: a family of quantitative abstractions parameterized by equivalence relations on $\mathcal{Q}$. Given an equivalence relation $\approx \subseteq \mathcal{Q} \times \mathcal{Q}$, the quantitative abstraction $ExistMax_{\approx} = (\mathcal{S}^{em}, f^{em}, \alpha^{em}_{\approx})$ has (a) $\mathcal{S}^{em} = \mathcal{S}(2^{\mathcal{Q}})$, (i.e., abstract states are set of concrete states), (b) $f^{em} = f$ as the abstract quantitative property, and (c) for a WTS $S = (Q, \delta, \varrho, q_\iota)$ we have that the abstract system $\alpha^{em}_{\approx}(S)$ is a WTS $S^{em} = (Q^{em}, \delta^{em}, \varrho^{em}, t^{em}_\iota)$ where:

- Q^{em} are equivalence classes of $\approx$ that contain states from Q;
- $(t_1, t_2) \in \delta^{em} \Leftrightarrow \exists q_1 \in t_1, q_2 \in t_2 : (q_1, q_2) \in \delta$;
- $\varrho(t) = \sup\{d \mid \exists q \in t : \varrho(q) = d\}$; and
- t^{em}_ι is the equivalence class in Q^{em} that contains q_ι.

Intuitively, $\alpha^{em}(S)$ is an existential abstraction and ϱ^{em} maps an abstract state to the maximum weight of the corresponding concrete state.

Example 1. *Consider again the system S_1 from Figure 2, and the equivalence relation $\approx$, whose two equivalence classes (indicated by the dashed shapes) are shown in the upper part of Figure 3. The abstract system $S^{em} = \alpha^{em}_{\approx}(S_1)$ is in the lower part of Figure 3. We have $limavg^{em}(\alpha^{em}_{\approx}(S^{em})) = 10$, due to a self-loop on the abstract node with weight 10.*

3.1.1 Over-approximation and monotonicity.

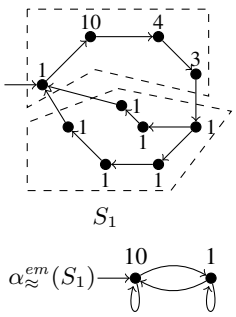


Fig. 3: *ExistMax* abs.

We characterize the quantitative properties for which *ExistMax* is an over-approximation, and for which monotonicity of refinements holds, i.e., where refinement of the abstraction leads to tighter approximations of the system value.

We borrow the classical notion of refinement for abstractions. An equivalence relation $\equiv$ is a *refinement* of an equivalence relation $\approx$ if and only if every equivalence class of $\equiv$ is a subset of an equivalence class of $\approx$.

We define the following quasi-orders:

- let $\leq_p \subseteq \mathbb{R}^\omega \times \mathbb{R}^\omega$ be defined by: $r^1_0 r^1_1 \ldots \leq_p r^2_0 r^2_1 \ldots$ if and only if $\forall i : r^1_i \leq r^2_i$, and
- let $\sqsubseteq \subseteq 2^{\mathbb{R}} \times 2^{\mathbb{R}}$ be defined by: for $U, U' \subseteq \mathbb{R}$, we have $U \sqsubseteq U'$ iff $\sup U \leq \sup U'$.

A quantitative property f is $(\leq_p, \sqsubseteq)$-*monotonic* if $\forall r, r' \in \mathbb{R}^\omega : r \leq_p r' \implies f_t(r) \leq f_t(r')$ and $\forall U, U' \in 2^{\mathbb{R}} : U \sqsubseteq U' \implies f_s(U) \leq f_s(U')$.

ExistMax is a *monotonic over-approximation* for a quantitative property f, if (a) *ExistMax* is an over-approximation for f, and (b) if for all $S \in \mathcal{S}(\mathcal{Q})$, and for all equivalence relations $\equiv$ and $\approx$ on $\mathcal{Q}$, such that $\equiv$ is a refinement of $\approx$, we have that $f^{em}(\alpha^{em}_{\equiv}(S)) \leq f^{em}(\alpha^{em}_{\approx}(S))$.

Theorem 2. *If f is $(\leq_p, \sqsubseteq)$-monotonic quantitative property, then ExistMax is a monotonic over-approximation of f.*

Proof. We first prove the monotonicity property of *ExistMax*. The over-approximation property follows naturally as follows. For any system S, we have $S = \alpha^{em}_{id}(S)$ where id is the identity relation. As id is a refinement of any equivalence relation $\approx$, by monotonicity it follows that $f(S) = f^{em}(\alpha^{em}_{id}(S)) \leq f^{em}(\alpha^{em}_{\approx}(S))$.

Let S be a system in $\mathcal{S}(\mathcal{Q})$, and let $\equiv$ and $\approx$ be equivalence relations on $\mathcal{Q}$. Let $\alpha_{\approx}(S) = (Q^1, \delta^1, \varrho^1, q^1_\iota)$ and $\alpha_{\equiv}(S) = (Q^2, \delta^2, \varrho^2, q^2_\iota)$ be *ExistMax* abstractions of S, where $\equiv$ is a refinement of $\approx$. Furthermore, let f defined by f_t and f_s be $(\leq_p, \sqsubseteq)$-monotonic. For each equivalence class t of $\equiv$, let $t^\sharp$ be the unique equivalence class of $\approx$ for which $t \subseteq t^\sharp$. The class $t^\sharp$ is guaranteed to exist as $\equiv$ is a refinement of $\approx$. Furthermore, by the definition of *ExistMax*, we have that: (a) $\varrho^1(t) \leq \varrho^2(t^\sharp)$, and (b) $(t_1, t_2) \in \delta^1 \implies (t^\sharp_1, t^\sharp_2) \in \delta^2$. Therefore, for any trace $\pi = t_0 t_1 \ldots$ of $\alpha_{\equiv}(S)$, there exists a trace $\pi^\sharp = t^\sharp_0 t^\sharp_1 \ldots$ of $\alpha_{\approx}(S)$ such that: $\varrho^1(t_0)\varrho^1(t_1)\ldots \leq_p \varrho^2(t^\sharp_0)\varrho^2(t^\sharp_1)\ldots$. By $\leq_p$-monotonicity of f_t, we get $f_t(\varrho(\pi)) \leq f_t(\varrho(\pi^\sharp))$. Hence, for each $w \in f_t(\varrho(\Pi(\alpha_{\equiv}(S))))$, there exists $w^\sharp \in f_t(\varrho(\Pi(\alpha_{\approx}(S))))$ with $w \leq w^\sharp$. This, in turn, gives us $f_t(\varrho(\Pi(\alpha_{\equiv}(S)))) \sqsubseteq f_t(\varrho(\Pi(\alpha_{\equiv}(S))))$. Hence, by $\sqsubseteq$-monotonicity of f_s, we get $f_s(f_t(\varrho(\Pi(\alpha_{\equiv}(S))))) \leq f_s(f_t(\varrho(\Pi(\alpha_{\approx}(S))))$, or equivalently, $f(\alpha_{\equiv}(S)) \leq f(\alpha_{\approx}(S))$. This proves the required theorem. as $f = f^{em}$. □

It is easy to show that safety, liveness, limit average, and discounted sum are $(\leq_p, \sqsubseteq)$-monotonic. The following proposition is a direct consequence.

Proposition 3. *ExistMax is a monotonic over-approximation for the limit-average, discounted-sum, safety, and liveness properties.*

Example 4. *We describe a property for which ExistMax is not a monotonic over-approximation. Let f be defined by $f_t(r) = \sup_{i,j \geq 0}(r_i - r_j)$ and $f_s(U) = \sup U$. The property f can be used to measure the variance in resource usage (where the usage in each step is given by the weight) during the execution of a program. Consider the system in Figure 4 and the ExistMax abstraction with abstract states given by the rectangles (the nodes outside the dotted boxes are each in a separate singleton equivalence classes) . Property f has value 2 on the abstract system due to the trace $A \to B \to C$ having maximal and minimal weights as 5 and 3 (under ExistMax abstract state A, B, and C have weights 5, 3, and 3 respectively). Refining the abstraction by completely splitting state B increases f to 4. Refining further by splitting both states A and C decreases f to 3 which is the true value of the concrete system. The sequence of refinements show that for property f, the ExistMax abstraction is neither an over-approximation (as the first abstract system has value 2 which is less than 3, the value of the concrete system), nor monotonic (as the sequence of values 2, 4 and 3 obtained through subsequent refinements first increase and then decrease).*

3.1.2 Evaluating quantitative properties on *ExistMax* abstractions.

Recall $f^{em} = f$ for *ExistMax* abstractions. To evaluate f^{em} (and obtain an *ext*-trace for refinement), any algorithm for finding *ext*-traces for the quantitative property f suffices. Standard algorithms exist when f is one of safety, liveness, discounted-sum, and limit-average properties. For limit-average, we use the classical Howard's policy iteration.

4. Segment-based quantitative abstractions

In this section, we present quantitative abstractions where elements of the abstract domain correspond to sets of trace segments.

Segments. A *segment* is a finite or infinite sequence of states in $\mathcal{Q}$. Let $\mathcal{Q}^*$ be the set of all finite segments, $\mathcal{Q}^\infty$ the set of all infinite segments, and let $\mathcal{Q}^{*\infty} = \mathcal{Q}^* \cup \mathcal{Q}^\infty$ be the set of all finite and infinite segments. Given a segment σ, let $|\sigma|$ denote the length of

the segment (the range of $|.|$ is thus $\mathbb{N} \cup \{\infty\}$). Given two segments σ_1 and σ_2 in $\mathcal{Q}^{*\infty}$ we write $\sigma_1\sigma_2$ for their concatenation, with $\sigma_1\sigma_2 = \sigma_1$, if σ_1 is in $\mathcal{Q}^{\infty}$. Also, we the notation $last(\sigma_1)$ and $first(\sigma_2)$ to represent the last state of a finite segment σ_1 and the first state of a segment σ_2.

We dub a nonempty set of segments a *SegmentSet*. We define the following operations and relations on SegmentSets and sets of SegmentSets.

- For SegmentSets T_1 and T_2, we have $T_1 \Subset T_2$ if $T_1 \subseteq \{w \mid \exists x \in \mathcal{Q}^*, \exists y \in \mathcal{Q}^{*\infty} : xwy \in T\}$, that is, all segments from T_1 occur as sub-segments of segments in T_2.
- For a set of SegmentSets $\mathcal{T}$, we define $\biguplus \mathcal{T}$ to be set of segments which can be obtained by concatenation of segments contained in SegmentSets in $\mathcal{T}$. Formally, $\biguplus \mathcal{T} = \{\sigma_0\sigma_1 \dots \sigma_n \mid \exists T_0, T_1 \dots T_n : (\forall i : 0 \leq i \leq n \implies \sigma_i \in T_i)\} \cup \{\sigma_0\sigma_1 \dots \mid \exists T_0, T_1 \dots : (\forall i : 0 \leq i \rightarrow \sigma_i \in T_i)\}$.
- A set of SegmentSets $\mathcal{T}$ *covers* a SegmentSet T if and only if
  - for all $T_i \in \mathcal{T}$, we have $T_i \Subset T$, and
  - $T \subseteq \biguplus \mathcal{T}$.

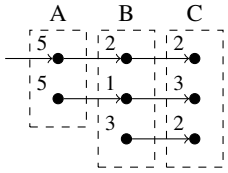


Fig. 4: Non-monotonic refinements

Note that for a WTS S, the set of all its traces $\Pi(S)$ is a SegmentSet. We call $\mathcal{T}$ a *segment cover* of a system S if and only if $\mathcal{T}$ covers $\Pi(S)$. For example, the two SegmentSets T_1 and T_2 in Figure 5 form a segment cover of the system in Figure 2. It is easy to see that all traces of S are covered by segments in T_1 and T_2. Our notion of segment cover corresponds to the inductive trace segment cover from [7] with height 1. The notion of the segment cover plays the same role in segment-based abstractions as the equivalence relation on states plays in state-based abstractions.

4.1 $PathBound$ abstraction.

$PathBound$ is a segment-based abstraction scheme: a family of quantitative abstractions parameterized by sets of SegmentSets on $\mathcal{Q}$. Given a set of SegmentSets $\mathcal{T}$, the quantitative abstraction $PathBound_{\mathcal{T}}$ is defined by $(\mathcal{S}^{pb}_{\mathcal{T}}, f^{pb}, \alpha^{pb}_{\mathcal{T}})$. We now define each element of the triple.

Abstract systems $\mathcal{S}^{pb}_{\mathcal{T}}$. An abstract system in $\mathcal{S}^{pb}_{\mathcal{T}}$ is a tuple $(R, \delta_R, val, minp, maxp, hasInfPath, R_0)$, where R is $\mathcal{T}$, δ_R is a transition relation, and R_0 is the set of initial states. The type of val, $minp$ and $maxp$ functions is $R \rightarrow \mathbb{R}$ and the type of $hasInfPath$ is $R \rightarrow \mathbb{B}$. Their intuitive meaning is given below. The systems in $\mathcal{S}^{pb}_{\mathcal{T}}$ are called *pb-systems*.

Abstraction $\alpha^{pb}_{\mathcal{T}}$. The partial abstraction function $\alpha^{pb}_{\mathcal{T}}$ is defined as follows. For a WTS S, if $\mathcal{T}$ is not a segment cover of $\Pi(S)$, then the value $\alpha^{pb}_{\mathcal{T}}(S)$ is undefined. Otherwise, given $S = (Q, \delta, \varrho, q_\iota)$, let $\alpha^{pb}_{\mathcal{T}}(S) = S^{pb} \in \mathcal{S}^{pb}$ where $S^{pb} = (Q^{pb}, \delta^{pb}, val, minp, maxp, hasInfPath, Q^{pb}_0)$ and

- Q^{pb} is $\mathcal{T}$
- $(T_1, T_2) \in \delta^{pb}$ if $\exists \sigma_1 \in T_1, \sigma_2 \in T_2$, such that σ_1 is a finite segment in $\mathcal{Q}^*$, $(last(\sigma_1), first(\sigma_2)) \in \delta$.
- $val(T) = \max\{\varrho(q) \mid \exists \sigma \in T$ and q occurs in $\sigma\}$, i.e., $val(T)$ is the maximal weight of a state occurring in one of the segments in T
- $minp(T) = \min\{|\sigma| \mid \sigma \in T$ is a finite segment$\}$ if T contains a finite segment, and is ∞ otherwise.
- $maxp(T) = \max\{|\sigma| \mid \sigma \in T$ is a finite segment$\}$, if T contains a finite segment, and is ∞ otherwise.
- $hasInfPath(T) = true$ iff $T \cap \mathcal{Q}^{\infty} \neq \emptyset$, i.e., $hasInfPath(T)$ is true if and only if T contains an infinite segment,

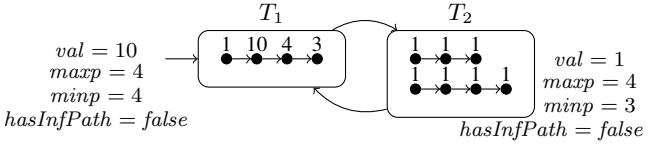


Figure 5: $PathBound$ abstraction of S_1

- Q^{pb}_0 contains a set T in $\mathcal{T}$ iff T contains a segment whose first state is q_ι. As $\mathcal{T}$ is a segment cover of S, we have that Q^{pb}_0 is non-empty.

A *pb-trace* ρ of a pb-system S^{α} is either (a) a finite sequence $T_0T_1 \dots T_n$ such that $T_0 \in Q^{pb}_0$, $hasInfPath(T_n)$, and $\forall i : 0 \leq i < n : (T_i, T_{i+1}) \in \delta^{pb}$, or (b) an infinite sequence $T_0T_1 \dots$ with $T_0 \in Q^{pb}_0$, and $\forall i \geq 0 : (T_i, T_{i+1}) \in \delta^{pb}$. The set of all pb-traces of S^{α} is denoted by $\Pi^{pb}(S^{\alpha})$.

Example 5. *Recall the system S_1 from Figure 2. Consider a segment cover $\mathcal{T} = \{T_1, T_2\}$ of $\Pi(S)$ depicted in Figure 5. T_1 and T_2 can now act as abstract states, with the values val,$minp$,$maxp$, and $hasInfPath$ given in Figure 5.*

Abstract quantitative property f^{pb}. In order to define the abstract quantitative property f^{pb}, we will need the following notions.

Let us fix a system $S = (Q, \delta, \varrho, q_\iota)$. Let us also fix a set $\mathcal{T}$ of SegmentSets, such that $\mathcal{T}$ is a segment cover for S. Let $\alpha^{pb}_{\mathcal{T}}(S) = S^{\alpha} = (Q^{pb}, \delta^{pb}, val, minp, maxp, hasInfPath, Q^{pb}_0)$ be a $PathBound$ abstraction of S for $\mathcal{T}$.

We now define a function B that for a given pb-trace ρ returns a set of possible sequences of weights that correspond to ρ. The function $B : \Pi(S^{\alpha}) \rightarrow 2^{\mathbb{R}^{\omega}}$ is defined as follows. The set $B(\rho)$ contains a sequence:

- $w_0^{n_0}w_1^{n_1} \dots w_n^{\infty}$ in $\mathbb{R}^{\omega}$ iff ρ is a finite pb-trace $T_0T_1T_2 \dots T_n$, such that (a) $\forall i$ such that $0 \leq i \leq n$, we have $w_i = val(T_i)$, and (b) $\forall i$ such that $0 \leq i < n$, we have $minp(T_i) \leq n_i \leq maxp(T_i)$ and $0 < n_i \neq \infty$.
- $w_0^{n_0}w_1^{n_1} \dots$ in $\mathbb{R}^{\omega}$ iff ρ is an infinite pb-trace $T_0T_1 \in \Pi^{pb}(S^{\alpha})$ such that (a) $\forall i \geq 0 : w_i = val(T_i)$, and (b) $\forall i \geq 0 : minp(T_i) \leq n_i \leq maxp(T_i) \wedge 0 < n_i \neq \infty$.

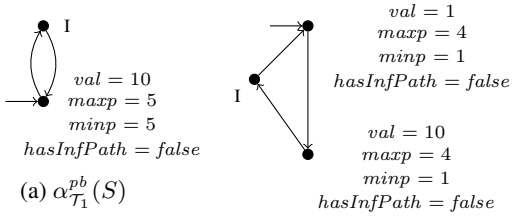
Let f be a quantitative property defined by a trace value function f_t and a system value function f_s. We are now able to define the abstract quantitative property f^{pb} by $f^{pb}(S^{\alpha}) = f_s(f_t((\bigcup_{\rho \in \Pi^{pb}} B(\rho))))$.

Example 6. *Recall again the system S_1 from Figure 2 and the abstract cover described in Example 5. Consider the abstraction $\alpha^{pb}_{\mathcal{T}}(S_1)$ (the abstract system is depicted in Figure 5). There is only one pb-trace ρ of the abstract system, and we have $\rho = (T_1T_2)^{\omega}$. Let us assume that the quantitative property we are interested in is the limit average quantitative property. We get that $B(\rho) = \{(10\ 10\ 10\ 10\ 1\ 1\ 1\ 1)^{\omega}, (10\ 10\ 10\ 10\ 1\ 1\ 1)^{\omega}\}$. We therefore obtain $f^{pb}(\alpha^{pb}_{\mathcal{T}}(S_1)) = (10 \cdot 4 + 1 \cdot 3)/(4 + 3) = \frac{43}{7}$, as the maximum value is achieved if the execution stays at the more costly abstract state T_1 as much as possible ($maxp(T_1)$ times), and at the less costly abstract state T_2 as little as possible ($minp(T_2)$ times).*

4.1.1 Over-approximation and monotonicity

The following theorem states that the abstraction scheme $PathBound$ is an over-approximation for a large class of quantitative properties.

Theorem 7. *$PathBound$ abstraction scheme is an over-approximation for a quantitative property f if f is $(\leq_p, \sqsubseteq)$-monotonic.*



(a) $\alpha^{pb}_{\mathcal{T}_1}(S)$

(b) $\alpha^{pb}_{\mathcal{T}_2}(S)$

Figure 7: Abstractions of system S_2

A set of SegmentSets $\mathcal{T}_1$ *refines* a set of SegmentSets $\mathcal{T}_2$ iff for all $T \in \mathcal{T}_2$, there exists a set of SegmentSets $\mathcal{T}$, such that $\mathcal{T} \subseteq \mathcal{T}_1$, and $\mathcal{T}$ covers T.

$PathBound$ is a *monotonic over-approximation* for a quantitative property f, if (a) $PathBound$ is an over-approximation for f, and (b) if for all $S \in \mathcal{S}(\mathcal{Q})$, and for all sets $\mathcal{T}_1$ and $\mathcal{T}_2$ of SegmentSets such that (a) $\mathcal{T}_1$ covers $\Pi(S)$, (b) $\mathcal{T}_2$ covers $\Pi(S)$, and (c) $\mathcal{T}_2$ is a refinement of $\mathcal{T}_1$, we have that $f^{pb}(\alpha^{pb}_{\mathcal{T}_2}(S)) \leq f^{pb}(\alpha^{pb}_{\mathcal{T}_1}(S))$.

The abstraction $PathBound$ is not a monotonic approximation in general even for quantitative properties that are $(\leq_p, \sqsubseteq)$-monotonic. We show this by constructing a counterexample (see Example 10) for which the abstraction is not a monotonic approximation for the limit-average property.

4.2 State-equivalence induced segment-based abstraction

Given an equivalence relation on states, we can define a set of SegmentSets. Let $S = (Q, \delta, \varrho, q_t)$ be a WTS, and let $\approx$ be an equivalence relation on states in $\mathcal{Q}$. Given an equivalence class e of $\approx$, we can define a corresponding SegmentSet T_e as follows. First, let T'_e be the set of (finite or infinite) segments σ such that all states q that occur in σ are in e. Now we define T_e as the set of maximal segments in T'_e. A segment σ is *maximal* in T_e iff (a) σ is in T'_e; (b) there is a transition $(q_b, first(\sigma)) \in \delta$ such that $q_b \notin e$; and (c) either $\sigma \in \mathcal{Q}^\infty$ or there is a transition $(last(\sigma), q_f) \in \delta$ such that $q_f \notin e$. Let $\mathcal{T}_\approx$ be a set of SegmentSets defined by $\{T_e \mid e \text{ is an equivalence class of } \approx\}$.

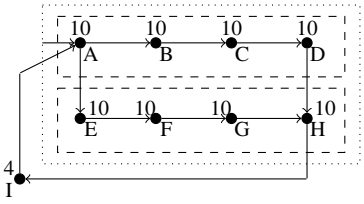


Figure 6: System S_2

Example 8. *Consider again the system S_1 in Figure 2, and the equivalence relation $\approx$ on its states given by the dashed shapes in Figure 3. The SegmentSets we get from the equivalence classes are given by the nodes T_1 and T_2 in Figure 5. The set of these SegmentSets is $\mathcal{T}_\approx$. As calculated in Example 6, the value for the abstract system (for the limit-average objective) given by $PathBound_{\mathcal{T}_\approx}$ is $\frac{43}{7}$. Note that this is better (more precise) than the value given by the $ExistMax$ abstraction defined by the same equivalence relation $\approx$. As calculated in Example 1, the value given by $ExistMax$ is* 10.

Given an equivalence relation $\approx$ on states, the abstraction $PathBound_{\mathcal{T}_\approx}$ gives a better over-approximation for limit-average objective than the $ExistMax_\approx$.

Proposition 9. *Let f be a $(\leq_p, \sqsubseteq)$-monotonic quantitative property. Let $\approx$ be an equivalence relation on $\mathcal{Q}$ and consider the two abstraction schemes $PathBound_{\mathcal{T}_\approx} = (\mathcal{S}^{pb}, f^{pb}, \alpha^{pb}_{\mathcal{T}_\approx})$ and $ExistMax_\approx = (\mathcal{S}^{em}, f^{em}, \alpha^{em}_\approx)$ parameterized by $\approx$. Then, for all $S \in \mathcal{S}(\mathcal{Q})$, $f^{pb}(\alpha^{pb}_{\mathcal{T}_\approx}(S)) \leq f^{em}(\alpha^{em}_\approx(S))$.*

We observe that if $\approx_2$ is a refinement $\approx_1$, then $\mathcal{T}_{\approx_2}$ is a refinement $\mathcal{T}_{\approx_1}$. However, we show an example system where the over-approximation computed using the $\mathcal{T}_{\approx_2}$ abstraction is worse (less precise) than the over-approximation computed using the $\mathcal{T}_{\approx_1}$. This means that, in general, $PathBound$ is not a monotonic over-approximation for $(\leq_p, \sqsubseteq)$-monotonic quantitative properties.

Example 10. *Consider the system S_2 in Figure 6. Consider an equivalence relation $\approx$ on states given by the dotted rectangle in the figure (that is all states except the state I are equivalent to each other). This equivalence relation defines a set $\mathcal{T}_1$ of SegmentSets. The resulting abstract system $\alpha^{pb}_{\mathcal{T}_1}(S)$ is in Figure 7 (a). Note that in Figure 7 (a), the node for which the values of $maxp$, $minp$, etc. are given corresponds to the dotted rectangle in Figure 6, the other node in the abstract system corresponds to the singleton segment of length one generated from the singleton equivalence class of the node I of system S_2. Consider now a refinement of $\approx$ where the equivalence class of the dotted rectangle is split into two equivalence classes given by the dashed rectangles. The new equivalence relation defines a set $\mathcal{T}_2$ of SegmentSets. The abstract system in Figure 7 (b) results from a refinement where the equivalence class of the dotted rectangle is split into two equivalence classes given by the dashed rectangles. The resulting abstract system $\alpha^{pb}_{\mathcal{T}_2}(S)$ is in Figure 7 (b).*

Let us now assume that the abstract quantitative objective is $limavg$. The value we get for system $\alpha^{pb}_{\mathcal{T}_1}(S)$ is $((10{\cdot}5{+}4{\cdot}1)/6) = 9$. The value we get for its refinement $\alpha^{pb}_{\mathcal{T}_2}(S)$ is $((10 \cdot 4 + 10 \cdot 4 + 4 \cdot 1)/9) = \frac{84}{9} > 9$. This shows that the estimate is worse (less precise) for the refinement $\alpha^{pb}_{\mathcal{T}_2}(S)$ than for $\alpha^{pb}_{\mathcal{T}_1}(S)$.

4.3 Specialization of $PathBound$ abstraction for the limit-average quantitative property

We presented a general definition of the $PathBound$ abstraction which is an over-approximation for a large class of quantitative properties. We now specialize the $PathBound$ abstraction for the limit-average property by introducing sound optimizations.

We define the limit-average $PathBound$-abstraction (denoted by $PathBoundLA$) scheme as $(\mathcal{S}^{pba}, f^{pba}, \alpha^{pba}_{\mathcal{T}})$. Let S be a system and let $S^{pb} = \alpha^{pb}_{\mathcal{T}}(S) = (Q^{pb}, \delta^{pb}, val, minp, maxp, hasInfPath, Q^{pb}_0)$ be a $PathBound$ abstraction of S. Furthermore, we fix f to be the limit-average property, i.e., $f_t = limavg$ and $f_s = \sup$ for the remainder of this subsection. We have that $S^{pba} = \alpha^{pba}_{\mathcal{T}}(S) = (Q^{pb}, \delta^{pb}, val^{pba}, minp, maxp, hasInfPath)$ is a *pb*-system similar to $\alpha^{pb}_{\mathcal{T}}(S)$. In the $PathBoundLA$ abstraction scheme, we have the following differences:

- $val^{pba}(T) = \max\{\sup_{\sigma \in (T \cap \mathcal{Q}^*)} \frac{\sum_{s \in \sigma} \varrho(s)}{|\sigma|}, \sup_{\sigma \in (T \cap \mathcal{Q}^\infty)} limavg(\sigma)\}$. Here, we let the value of an abstract SegmentSet T be the supremum of the average weight of the segments in T, rather than the maximum weight occurring in T. Note that if $T \subseteq \mathcal{Q}^\infty$, we have $val^{pba}(T) = \sup(limavg(\varrho(T)))$.
- The abstract quantitative property f^{pba} is defined in the same way as f^{pb} (at the beginning of Section 4), except that the definition of $B : \Pi(S^\alpha) \to 2^{\mathbb{R}^\omega}$ we have that:
  - $w_0^{n_0} w_1^{n_1} \ldots \in B(\rho)$ if and only if $\rho = T_0 T_1 \ldots$ with $w_i = val^{pba}(T_i) \wedge 0 < n_i < \infty \wedge n_i \in \{minp(T_i), maxp(t_i)\}$.
  - $w_0^{n_0} w_1^{n_1} \ldots w_n^\infty \in B(\rho)$ if and only if $\forall 0 \leq i \leq n : w_i = val^{pba}(T_i) \wedge (i < n \implies 0 < n_i < \infty \wedge n_i \in \{minp(T_i), maxp(t_i)\})$.

The above differences between the $PathBoundLA$ and $PathBound$ can be summarized as follows: (a) the value summarization function for each SegmentSet can be average instead of maximum, and (b) more crucially (from a practical point of view), the evaluation of the abstract property on a pb-system can be done by considering only the lengths of the longest and shortest finite paths of an SegmentSet, rather than considering all lengths between them. This is because limit-average is a memoryless property.

The following theorem states that $PathBoundLA$ provides a better approximations of the limit-average property than $PathBound$.

Theorem 11. *Given a system S and an segment cover $\mathcal{T}$ of $\Pi(S)$ and f being the limit-average property, we have*

$$f(S) \leq f^{pba}(\alpha^{pba}_{\mathcal{T}}(S)) \leq f^{pb}(\alpha^{pb}_{\mathcal{T}}(S))$$

Example 12. *Consider again the system S_1 from Figure 2, and the SegmentSets T_1 and T_2 from Figure 5. In $PathBoundLA$ abstraction, we have $val(T_1) = (10 + 1 + 4 + 3)/4 = 18/4$ (while the other values for T_1 are as in Figure 5). For $val(T_2)$, we have $val(T_2) = (1 + 1 + 1)/3 = 1$. The value of the system is $f^{pba}(\alpha_{\mathcal{T}}(S_1)) = (18 + 3)/7 = 3$. Recall that for $PathBound$ abstraction, the value f^{pb} for S_1 was calculated in Example 6 to be $\frac{43}{7}$. For $PathBoundLA$ abstraction, we thus get a better approximation than in the $PathBound$ abstraction.*

Evaluating limit-averages on pb-abstract systems. Minimum-mean cycle algorithms compute the limit-average value for a graph with weights on edges. Therefore, from a pb-system, we construct a graph which has weights and lengths on edges, rather than nodes. Intuitively, we consider the graph with edges of the pb-system being the nodes. There are two edges between the node (T_1, T) and (T, T_2): one of weight $maxp(T) \times val(T)$ and length $maxp(T)$, and another of weight $minp(T) \times val(T)$ and length $minp(T)$. The node (T_1, T) has the self-loop of weight $val(T)$ and length 1, if $hasInfPath(T)$ is true. We denote this graph by $S^{\dagger}$.

Howard's policy iteration was extended in [3] to compute the limit-average values in graphs where edges have both weight and length, as is the case of $S^{\dagger}$. Howard's policy iteration works by picking a policy (that maps states to successors) and improves the policy as long as possible. Each improvement takes linear time, but only an exponential upper-bound is known on the number of improvements required. However, a number of reports state that only linear number of improvements are required for most cases in practice [3].

5. Generalizations of $PathBound$ abstractions

In this section, we present two generalizations of the $PathBound$ abstraction scheme. The first one generalizes $PathBound$ by considering different summaries of SegmentSets rather than set of properties $\{minp, maxp, val, hasInfPath\}$. The second one generalizes $PathBound$ by allowing inductive fixed-point style computations of properties.

5.1 Generalized segment-based abstraction

Let S be a WTS, and let $\mathcal{T}$ be a segment over of $\Pi(S)$. In the $PathBoundLA$ abstraction from Section 4.3, we used the values of $maxp$, $minp$, $hasInfPath$, and val^{pba} to abstract SegmentSets in $\mathcal{T}$. The abstract values are in turn used to compute an over-approximation of the limit-average property for S. In this subsection, we provide a generic segment-based abstraction scheme for any set of properties.

More specifically, let us assume that we have a set $\mathcal{P}$ of quantitative properties, a set $\mathcal{T}$ of SegmentSets that is a segment cover of a segment set T. We provide a generic technique to answer the following question: if we know the values of quantitative properties in $\mathcal{P}$ on all SegmentSets in $\mathcal{T}$, can we compute the values of quantitative properties in $\mathcal{P}$ on SegmentSet T?

We need to extend notation in two ways: (a) We will use quantitative properties f defined by f_t and f_s for both finite and infinite traces. The type of f_t will thus be $\mathbb{R}^{\omega} \cup \mathbb{R}^{*} \to \mathbb{R}$; and (b) We will evaluate quantitative properties on a SegmentSet T (instead of a WTS) by letting $f(T) = f_s(\{f_t(\varrho(\pi)) \mid \pi \in T\})$.

Fix an arbitrary WTS $S = (Q, \delta, \varrho, q_\iota)$. Consider an arbitrary set of SegmentSets $\mathcal{T} = \{T_0, T_1, \ldots, T_n\}$, where all segments are sequences of states from Q. We define the set of *valid segments* generated by $\mathcal{T}$ as $\biguplus^{\delta} \mathcal{T} = \{\sigma_0 \ldots \sigma_n \mid \forall j : \sigma_j \in \bigcup \mathcal{T} \wedge \forall j < n : (last(\sigma_j), first(\sigma_{j+1})) \in \delta\} \cup \{\sigma_0 \ldots \mid \forall j : \sigma_j \in \bigcup \mathcal{T} \wedge \forall j \geq 0 : (last(\sigma_j), first(\sigma_{j+1})) \in \delta\}$ where $first(\sigma)$ and $last(\sigma)$ denote the first and last states of a segment. Intuitively, the set $\biguplus^{\delta} \mathcal{T}$ is the set of segments generated by $\mathcal{T}$ where the transition relation of the system S holds at the sub-segment boundaries.

Let $\mathcal{T} = \{T_1, \ldots, T_n\}$ be a cover of the SegmentSet T (not necessarily $\Pi(S)$). Note that $\biguplus^{\delta} \mathcal{T}$ can be a proper subset of T, i.e., $T \subseteq \biguplus \mathcal{T}$, but $\biguplus^{\delta} \mathcal{T} \subsetneq T$. We call the SegmentSet $T \cap \biguplus^{\delta} \mathcal{T}$ the *strengthening* of the SegmentSet T by $\mathcal{T}$ and δ. Our question thus becomes: provided that we know the values of quantitative properties in $\mathcal{P}$ on all SegmentSets in $\mathcal{T}$, can we compute the values of quantitative properties in $\mathcal{P}$ on the strengthening of the SegmentSet T by $\mathcal{T}$ and δ, i.e. on $T \cap \biguplus^{\delta} \mathcal{T}$?

Abstract SegmentSet and property domains. Let $\langle SEG, \subseteq \rangle$ be the set of all SegmentSets partially ordered by the subset relation, and let $\langle \mathcal{L}, \preceq \rangle$ be a lattice. The lattice serves as an abstract domain for describing SegmentSets. Elements of L can be for instance syntactical objects, such as formulas in a logic. Let $\langle SEG, \subseteq \rangle \underset{\alpha^{SEG}}{\overset{\gamma^{SEG}}{\leftrightarrows}} \langle \mathcal{L}, \preceq \rangle$ be a Galois connection (see [6]). We call the domain $\langle \mathcal{L}, \preceq \rangle$ the *abstract SegmentSet domain* and each element $\phi \in \mathcal{L}$ an *abstract SegmentSet*.

A *property set* $\mathcal{P}$ is a tuple $\langle \langle f^l_1, \ldots f^l_n \rangle, \langle f^u_1, \ldots f^u_m \rangle \rangle$ where (a) all f^l_i's are quantitative properties where $f_s = \inf$; and (b) all f^u_i's are quantitative properties where $f_s = \sup$. We define the corresponding property domain $\mathcal{D}_{\mathcal{P}} = \langle (\mathbb{R}^n \times \mathbb{R}^m), \sqsubseteq_{\mathcal{P}} \rangle$ to be the abstract domain where $((a^l_1, \ldots, a^l_n), (a^u_1, \ldots, a^u_n)) \sqsubseteq ((b^l_1, \ldots, b^l_n), (b^u_1, \ldots, b^u_n))$ if and only if all $a^l_i \geq b^l_i$ and $a^u_i \leq b^u_i$. We write $\mathcal{P}(T)$ for $((f^l_1(T), \ldots, f^l_n(T)), (f^u_1(T), \ldots, f^u_n(T)))$.

Example 13. *The property set $\mathcal{P}^{LA} = \langle \langle minp \rangle, \langle maxp, val^{pba}, hasInfPath \rangle$ is a property set. For example, $minp(T) = \inf(\{|\sigma| \mid \sigma \in T\})$ and $hasInfPath(T) = \sup(\{hasInfPath_t(\sigma) \mid \sigma \in T\})$ where $hasInfPath_t(\sigma)$ is 1 if $|\sigma| = \infty$ and 0 otherwise.*

It is easy to see that a Galois connection $\langle SEG, \subseteq \rangle \underset{\alpha^{\mathcal{P}}}{\overset{\gamma^{\mathcal{P}}}{\leftrightarrows}} \langle \mathcal{D}_{\mathcal{P}}, \sqsubseteq_{\mathcal{P}} \rangle$ can be defined by letting $\alpha^{\mathcal{P}}(T) = \mathcal{P}(T)$ and $\gamma^{\mathcal{P}}(P) = \bigcup \{T \mid \mathcal{P}(T) \sqsubseteq P\}$. Intuitively, $((l_1, \ldots, l_n), (u_1, \ldots, u_n)) \in \mathcal{P}(T)$ represents the largest SegmentSet T that respects the lower and upper bounds placed by P, i.e., $f^l_i(T) \geq l_i \wedge f^u_j(T) \leq u_j$. We call $\mathcal{D}_{\mathcal{P}}$ the *property bound domain* and an individual $P \in \mathcal{D}_{\mathcal{P}}$ a *property bound*.

Let $\langle \mathcal{L} \times \mathcal{D}_{\mathcal{P}}, \leq \rangle$, where $\leq$ is $\preceq \times \sqsubseteq_{\mathcal{P}}$, be the product of $\langle \mathcal{L}, \preceq \rangle$ and $\langle \mathcal{D}_{\mathcal{P}}, \sqsubseteq_{\mathcal{P}} \rangle$. We call $\mathcal{L} \times \mathcal{D}_{\mathcal{P}}$ the domain of abstract bound pairs and each element (written as $\phi \wedge P$) an abstract bound pair. Let $\langle SEG, \subseteq \rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle \mathcal{L} \times \mathcal{D}_{\mathcal{P}}, \preceq \times \sqsubseteq_{\mathcal{P}} \rangle$ be a Galois connection naturally defined for the product of abstract domains, where $\alpha(T) = (\alpha^{SEG}(T), \alpha^{\mathcal{P}}(T))$, and $\gamma(\phi, P) = \gamma^{SEG}(\phi) \cap \gamma^{\mathcal{P}}(P)$. Intuitively, the element $\phi \wedge P$ represents a SegmentSet that is contained in ϕ and respects the property bounds P. We identity abstract segments $\phi \in \mathcal{L}$ with the abstract bound pair in $\phi \wedge \top$

where $\top = ((-\infty, \ldots, -\infty), (\infty, \ldots, \infty))$, i.e., there are no bounds on any of the properties in $\mathcal{P}$.

Example 14. *Let $\mathcal{L} = SEG$, i.e., the abstract SegmentSets are the same as concrete SegmentSets. Note that this assumption is to simplify the example. It would be more natural to use syntactical objects (e.g. formulas in some logic) for $\mathcal{L}$. Consider $\mathcal{D}_{\mathcal{P}}^{LA}$ for the property bound domain defined in Example 13. An example of an element in $\mathcal{L} \times \mathcal{D}_{\mathcal{P}}^{LA}$ is $T_1 \wedge ((4), (4, \frac{18}{4}, 0))$ where T_1 is from Example 12. Here, $((4), (4, \frac{18}{4}, 0))$ represent bounds on values of properties $((minp), (maxp, val^{pba}, hasInfPath))$. Note that $T_1 \wedge ((4), (4, \frac{18}{4}, 0))$ contains exactly the same information that $PathBoundLA$ stores about a SegmentSet.*

Evaluating quantitative properties on abstract SegmentSets. In what follows, we fix a WTS S, and a segment cover $\mathcal{T} = \{T_1, T_2, \ldots, T_{|\mathcal{T}|}\}$ of $T = \Pi(S)$. Let K be the interval $[1..|\mathcal{T}|]$. Given an abstract bound pair domain $\langle \mathcal{L} \times \mathcal{D}_{\mathcal{P}}, \preceq \times \sqsubseteq_{\mathcal{P}} \rangle$, where f is a property in $\mathcal{P}$, we can perform the computation of $f(T \cap \biguplus^{\delta} \mathcal{T})$ in the abstract domain. For this computation, the abstract bound pair domain needs to support the following additional operations.

Let $\phi \wedge P$ be an abstract bound pair, and $\Phi = \{\phi_1 \wedge P_1, \ldots, \phi_{|\mathcal{T}|} \wedge P_{|\mathcal{T}|}\}$ be a set of abstract bound pairs. The abstract bound pairs domain $\langle \mathcal{L} \times \mathcal{D}_{\mathcal{P}}, \preceq \times \sqsubseteq_{\mathcal{P}} \rangle$ is an *inductive domain* if it supports the following operations in addition to the standard lattice operations.

- *Transition check.* This operation checks for two abstract SegmentSets, whether a segment from the first can be validly followed by a segment from the second. Formally, $TrCheck_{\phi}(\phi_i, \phi_j)$ is true if and only if there is validly generated segment σ, i.e., $\sigma \in \gamma(\phi) \cap \biguplus^{\delta} \{\gamma(\phi_i) \mid i \in K\}$, where $\sigma = \sigma' \sigma_i \sigma_j \sigma''$ such that (a) $\sigma_i \in \gamma(\phi_i) \wedge \sigma_j \in \gamma(\phi_j)$, and (b) σ' and σ'' are also validly generated.
- *Reduce Property Bounds.* Given an abstract SegmentSet ϕ, compute (an over-approximation) of the property bounds on ϕ. Formally, $ReduceBound(\phi)$ returns $\phi \wedge P^*$ such that $\gamma(\phi \wedge P^*) \supseteq \gamma(\phi)$.
- *Property computation.* Given only the bounds P_i on abstract SegmentSets ϕ_i and the values $TrCheck_{\phi}(\phi_i, \phi_j)$, $PropComp$ computes (an over-approximation) of the property bounds on the abstract segment $\phi \wedge P$. Formally, given $\delta^{gpb} = \{(i, j) \mid TrCheck_{\phi}(\phi_i, \phi_j) = true\}$, and the values P_i for all i, the function $PropComp(\phi, \delta^{gpb}, \langle P_1 \ldots P_{|\mathcal{T}|} \rangle)$ outputs $\phi^* \wedge P^*$ such that $\phi^* \wedge P^* \geq (\phi \wedge \top) \wedge \alpha(\biguplus^{\delta} \{\gamma(\phi_i \wedge P_i) \mid i \in K\})$. Note that we sometimes abuse the notation and write $PropComp(\phi, \{\phi_1 \wedge P_1, \ldots, \phi_{|\mathcal{T}|} \wedge P_{|\mathcal{T}|}\})$ instead of $PropComp(\phi, \delta^{gpb}, \langle P_1 \ldots P_{|\mathcal{T}|} \rangle_i)$.

Example 15. *Continuing from Example 14, i.e., $\mathcal{L} = SEG$ and the system under consideration is S_1 from Figure 5.*

- *$TrCheck_{\Pi(S)}(T_1, T_2)$ can be as precise as the transition relation of $PathBoundLA$, i.e, $(T_1, T_2) \in \delta^{pba} \Leftrightarrow TrCheck_{\Pi(S)}(T_1, T_2)$. Suppose $\sigma_1 \in T_1$ and $\sigma_2 \in T_2$. We have that σ_1 can be followed by σ_2 if and only if $(last(\sigma_1), first(\sigma_2)) \in \delta$. This is the condition that defines $(T_1, T_2) \in \delta^{pb}$.*
- *If we take $ReduceBound(T_2)$ to be precise (and computationally expensive) procedure that concretizes abstract states, it can return $T_2 \wedge ((3), (4, 1, 0))$, i.e., it computes (an over-approximation) of the information stored for the particular SegmentSetT_2.*
- *$PropComp(\Pi(S), \{T_1 \wedge P_1, T_2 \wedge P_2\})$ (where $P_1 = ((4), (4, \frac{18}{4}, 0))$ and $P_2 = ((3), (4, 1, 0))$) computes an over-approximation of $\mathcal{P}(\Pi(S))$ by computing on abstract states. If $PropComp$ is defined using the same approach as the B and f^{pba} definitions from Section 4.3, it returns $((4), (\infty, 3, 1))$. Here, $val^{pba}(\Pi(S))$ is 3, $minp = 4$, and $maxp$ and $hasInfPath$ are over-approximated to the largest possible values.*

Remark 16. *Note that in the arguments of $PropComp$, we do not require P_i to be equal to $\mathcal{P}(\phi_i)$. In fact, even in the case where P_i is a $\sqsubseteq_{\mathcal{P}}$-over-approximation of $\mathcal{P}(\phi_i)$, the procedure $PropComp$ is required to produce a valid $\sqsubseteq_{\mathcal{P}}$-over-approximation of the $\mathcal{P}$ value of $\gamma(\phi) \cap \biguplus^{\delta} \{\gamma(\phi_i) \mid i \in K\}$.*

We call an abstract bound pair domain *effective* if: (a) each of the operations $ReduceBound$, $TrCheck$, and $PropComp$ can be computed effectively, i.e., by a terminating procedure; and (b) $ReduceBound$ and $PropComp$ are monotonic, i.e., giving more precise inputs produces more precise outputs. Formally, (a) $\phi \preceq \phi' \implies ReduceBound(\phi) \leq ReduceBound(\phi')$; (b) $(\phi \preceq \phi' \wedge \forall i \in K : P_i \sqsubseteq P_i' \wedge \phi_i \preceq \phi_i') \implies PropComp(\phi, \{\phi_1 \wedge P_1, \ldots, \phi_{|\mathcal{T}|} \wedge P_{|\mathcal{T}|}\}) \leq PropComp(\phi, \{\phi_1 \wedge P_1', \ldots, \phi_{|\mathcal{T}|} \wedge P_{|\mathcal{T}|}'\})$; and (c) $(\phi_1^1 \wedge P_1^1 \leq \phi_1 \wedge P_1) \wedge (\phi_1^2 \wedge P_1^2 \leq \phi_1 \wedge P_1) \implies PropComp(\phi, \{\phi_1^1 \wedge P_1^1, \phi_1^2 \wedge P_1^2, \phi_2 \wedge P_2, \ldots, \phi_{|\mathcal{T}|} \wedge P_{|\mathcal{T}|}\}) \leq PropComp(\phi, \{\phi_1 \wedge P_1, \phi_2 \wedge P_2, \ldots, \phi_{|\mathcal{T}|} \wedge P_{|\mathcal{T}|}\})$.

We can now generalize $PathBound$ abstraction scheme by letting the summaries of SegmentSets be the set of values of properties from any effectively inductive quantitative property set. Intuitively, given a WTS S, and a property set $\mathcal{P}$, the $PathBound[\mathcal{L} \times \mathcal{D}_{\mathcal{P}}]$ abstraction stores with each SegmentSet T in the cover of $\Pi(S)$ the values $\alpha^{SEG}(T) \wedge \mathcal{P}(T)$. To compute the value of the abstract system, the procedures $ReduceBound$, $TrCheck$, and $PropComp$ are used.

Formally, $PathBound[\mathcal{L} \times \mathcal{D}_{\mathcal{P}}]_{\mathcal{T}} = \langle \mathcal{S}[\mathcal{L} \times \mathcal{D}_{\mathcal{P}}], \alpha[\mathcal{L} \times \mathcal{D}_{\mathcal{P}}]_{\mathcal{T}}, f[\mathcal{L} \times \mathcal{D}_{\mathcal{P}}] \rangle$ is defined as:

- $\alpha[\mathcal{L} \times \mathcal{D}_{\mathcal{P}}]_{\mathcal{T}}(S) = \langle \phi, \langle \phi_1, \ldots, \phi_{|\mathcal{T}|} \rangle, \delta^{gpb} \rangle$ where (a) $\phi = \alpha^{SEG}(\Pi(S))$; (b) $\phi_i = \alpha^{SEG}(T_i)$; and (c) $(i, j) \in \delta^{gpb}$ if and only if $TrCheck_{\phi}(\phi_i, \phi_j)$ returns true.
- $f[\mathcal{L} \times \mathcal{D}_{\mathcal{P}}]$ is computed as the value of f in P^* where $\phi^* \wedge P^* = PropComp(\phi, \delta^{gpb}, \langle ReduceBound(\phi_1), \ldots, ReduceBound(\phi_{|\mathcal{T}|}) \rangle) \wedge ReduceBound(\phi)$.

Precise abstractions. In the $ExistMax$ case, if the abstraction function does not lose any information, i.e., if the equivalence relation used is the identity relation, the abstract system value is the same as the concrete system value. We give the conditions when an segment-based abstraction does not lose any information either.

Intuitively, we want the property computation for a SegmentSet T from a cover $\mathcal{T} = \{T_1, \ldots T_n\}$ to be accurate when T is exactly covered by $\mathcal{T}$, i.e, $T = \biguplus^{\delta} \mathcal{T}$.

An quantitative property set is *precisely inductive* if there exists a $PropComp$ procedure which produces the output $(\phi, \mathcal{P}(\phi))$ when the following hold: (a) $P_i = \mathcal{P}(\gamma(\phi_i))$, and (b) $\gamma(\phi) = \biguplus^{\delta} \{\gamma(\phi_i) \mid i \in K\}$.

Example 17. *The limit-average property is not precisely inductive, i.e., by knowing only the limit-average value of the subsegments of a segment, we cannot estimate the limit-average accurately without knowing the length of the subsegments. However, strengthening it to the property set $\langle \langle minp \rangle, \langle maxp, limavg, hasInfPath \rangle \rangle$ makes it precisely inductive.*

5.2 Hierarchical segment-based abstraction

Effective abstract bound pair domains allow computation of property bounds for a whole set of properties on a SegmentSet from the property bounds on each element of the cover. For example, if $\mathcal{T}$ covers T, the four properties $minp, maxp, val, hasInfPath$ of SegmentSets in $\mathcal{T}$ are used to compute not only the $limavg$, but also the $maxp$, $minp$, and $hasInfPath$ values of the SegmentSet T

too. This leads to the possibility of computing these properties hierarchically for a multi-level trace segment cover.

An *inductive trace segment cover* [7] $\mathcal{C}$ is a finite rooted-tree where the nodes are labelled with abstract bound pairs such that for every non-leaf node labelled with SegmentSet $\phi \wedge P$, and $\phi_1 \wedge P_1, \ldots, \phi_n \wedge P_n$ the set of labels of its children, $\{\gamma(\phi_1 \wedge P_1), \ldots, \gamma(\phi_n \wedge P_n)\}$ is a segment cover of $\gamma(\phi \wedge P)$. An inductive trace segment cover $\mathcal{C}$ *inductively covers* a SegmentSet T if $T \subseteq \gamma(root(\mathcal{C}))$. Similarly, $\mathcal{C}$ inductively covers a system S if $\Pi(S) \subseteq \gamma(root(\mathcal{C}))$.

We can now introduce an abstraction scheme $HPathBound[\mathcal{L} \times \mathcal{D}_\mathcal{P}]_\mathcal{C} = \langle \mathcal{S}^{hpb}[\mathcal{L} \times \mathcal{D}_\mathcal{P}], \alpha^{hpb}[\mathcal{L} \times \mathcal{D}_\mathcal{P}]_\mathcal{C}, f^{hpb}[\mathcal{L} \times \mathcal{D}_\mathcal{P}] \rangle$ parameterized by an abstract bound pair domain $\mathcal{L} \times \mathcal{D}_\mathcal{P}$, and an abstract inductive trace segment cover $\mathcal{C}$. Intuitively, $HPathBound$ stores for each internal node a of $\mathcal{C}$, a $PathBound[\mathcal{L} \times \mathcal{D}_\mathcal{P}]_{\mathcal{T}(a)}$ abstraction $\langle label(a), \mathcal{T}(a), \delta^{gpb} \rangle$. The abstract trace segment cover $\mathcal{T}(a)$ for this abstraction is the labels of the set $children(a)$.

Abstract hierarchical traces. We fix an abstract bound pair domain $\mathcal{L} \times \mathcal{P}_\mathcal{D}$. An *abstract hierarchical trace* $\psi = \langle (\phi_0 \wedge P_0)(\phi_1 \wedge P_1) \ldots, sub_\psi \rangle$ consists of (a) a finite or infinite sequence of abstract bound pairs; and (b) a partial function sub_π from $\mathbb{N}$ to hierarchical traces.

The concrete traces $\gamma(\psi)$ corresponding to a given abstract trace $\psi = \langle (\phi_0 \wedge P_0)(\phi_1 \wedge P_1) \ldots, sub_\psi \rangle$ are defined as $\{\sigma_0 \sigma_1 \ldots | \forall i : \sigma_i \in \gamma(\phi_i \wedge P_i) \wedge sub_\psi(i) \neq \bot \implies \sigma_i \in \gamma(sub_\psi(i))\}$. Intuitively, a concrete trace of an abstract hierarchical trace ψ is made of segments σ_i from $\phi_i \wedge P_i$, with the additional condition that $\sigma_i \in sub_\psi(i)$ if $sub_\psi(i)$ is defined.

Given a property set $\mathcal{P}$, an effective abstract bound pair domain, and an inductive trace segment cover $\mathcal{C}$ of system S, we inductively compute property bounds (and hence, abstract system values) using Algorithm 1. The algorithm is based on the inductive proof method presented in [7]. It can be rewritten as a fixed-point computation in the lattice $\mathcal{C} \to \mathcal{L} \times \mathcal{D}_\mathcal{P}$ of maps from $nodes(\mathcal{C})$ to $\mathcal{L} \times \mathcal{D}_\mathcal{P}$ (the lattice ordered point-wise by $(\preceq, \sqsubseteq_\mathcal{P})$). At the final line in Algorithm 1, the value of $\mathcal{C}$ will be the least fixed-point of the function which replaces each node in $\mathcal{C}$ with a best approximation obtained from among the current value of the node, and (a) $ReduceBound$ applied on the node if it is a leaf node, or (b) $PropComp$ applied on the node and its children if it is an internal node.

Algorithm 1 Inductive Property Computation (InductiveCompute)

Input: WTS S,
Effective abstract bound pair domain $\mathcal{L} \times \mathcal{D}_\mathcal{P}$,
Abstract Inductive Segment Cover $\mathcal{C}$ (labelled from $\mathcal{L} \times \mathcal{D}_\mathcal{P}$),
Output: abstract bound pair $\phi^* \wedge P^*$ such that $(\phi^* \wedge P^*) \geq \mathcal{P}(label(root(\mathcal{C})))$

1: $(\phi \wedge P) \leftarrow label(root(\mathcal{C}))$
2: **if** $root(\mathcal{C})$ has no children **then**
3: **return** $(\phi \wedge P) \wedge (ReduceBound(\phi \wedge P))$
4: **else**
5: $subTrees \leftarrow$ top level sub-trees of $\mathcal{C}$
6: **return** $(\phi \wedge P) \wedge PropComp(\phi \wedge P, \{InductiveCompute(\mathcal{C}') \mid \mathcal{C}' \in subTrees\})$

Monotonicity. We define refinements of $HPathBound$ abstractions using refinement steps. Let $\mathcal{C}$ be an inductive trace segment cover and let a be a non-root node in $\mathcal{C}$, b be its parent, $\mathcal{C}(a)$ be the sub-tree of $\mathcal{C}$ rooted at b, and $\phi \wedge P$ and $\phi^b \wedge P^b$ the labels of a and b. A *one-step refinement* of $\mathcal{C}$ is one of the following:

- Horizontal refinement. Let ϕ_1 and ϕ_2 be such that $\gamma(\phi \wedge P) = \gamma(\phi_1 \wedge P_1) \cup \gamma(\phi_2 \wedge P_2)$. Let $\mathcal{C}(a)[label(a) \leftarrow \phi_1 \wedge P_1]$ and $\mathcal{C}(a)[label(a) \leftarrow \phi_2 \wedge P_2]$ be the tree $\mathcal{C}(a)$ with $\phi \wedge P$ replaced by $\phi_1 \wedge P_1$ and $\phi_2 \wedge P_2$, respectively. Let the tree $\mathcal{C}'$ be obtained by detaching $\mathcal{C}(a)$ from p and then attaching $\mathcal{C}(a)[label(a) \leftarrow \phi_1 \wedge P_1]$ and $\mathcal{C}(a)[label(a) \leftarrow \phi_2 \wedge P_2]$ to p. Then $\mathcal{C}'$ is a *one-step horizontal refinement* of $\mathcal{C}$.
- Vertical splitting refinement. Suppose $\{\gamma(\phi_1 \wedge P_1), \ldots \gamma(\phi_n \wedge P_n)\}$ cover $\phi \wedge P$ and that a does not have any children. Suppose $\mathcal{C}'$ is obtained from $\mathcal{C}$ by adding the children with labels $\phi_1 \wedge P_1, \ldots, \phi_n \wedge P_n$ to a. Then, $\mathcal{C}'$ is a *one-step vertical splitting refinement* of $\mathcal{C}$.
- Vertical joining refinement. Suppose that b has no grandchildren, and let $(\phi^b \wedge P^b) \leq PropComp(\phi \wedge P, children(b), \delta)$ and $\forall c \in children(b) : c) \leq ReduceBound(c))$. If $\mathcal{C}'$ is the tree obtained by removing all the children of b in $\mathcal{C}$, then $\mathcal{C}'$ is a *one-step vertical joining refinement* of $\mathcal{C}$.
- Downward strengthening refinement. Suppose $\gamma(\phi') \subseteq \gamma(\phi)$ and $\phi^b \wedge P^b \subseteq \biguplus^\delta \{\gamma(\psi) \mid \psi \in ((children(b) \setminus \{\phi \wedge P\}) \cup \{\phi' \wedge P'\})\}$. Let $\mathcal{C}'$ be the tree obtained by replacing $\mathcal{C}(a)$ by $\mathcal{C}(a)[label(a) \leftarrow \phi' \wedge P']$. Then, $\mathcal{C}'$ is a *one-step downward strengthening refinement* of $\mathcal{C}$.
- Upward strengthening refinement. Suppose that $\phi' \wedge P'$ is such that $\biguplus^\delta \{\gamma(\psi) \mid \psi \in children(b)\} \subseteq \gamma(\phi' \wedge P') \subseteq \gamma(\phi^b \wedge P^b)$. If $\mathcal{C}'$ is obtained from $\mathcal{C}$ by replacing $\phi^b \wedge P^b$ with $\phi' \wedge P'$, then $\mathcal{C}'$ is a *one-step upward strengthening refinement* of $\mathcal{C}$.

A $\mathcal{C}_n$ is a *refinement* of $\mathcal{C}_0$ if there exists a sequence $\mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_{n-1}$ such that $\mathcal{C}_i$ is a one-step refinement of $\mathcal{C}_{i-1}$ for all i such that $n \geq i > 0$.

$HPathBound[\mathcal{L} \times \mathcal{D}_\mathcal{P}]$ is *monotonic* if for all systems S and abstract inductive trace segment covers $\mathcal{C}$ and $\mathcal{C}'$ of S, if $\mathcal{C}'$ is a refinement of $\mathcal{C}$, then abstract value $f^{hpb}[\mathcal{L} \times \mathcal{D}_\mathcal{P}](\alpha^{hpb}[\mathcal{L} \times \mathcal{D}_\mathcal{P}]_{\mathcal{C}'}(S)) \leq f^{hpb}[\mathcal{L} \times \mathcal{D}_\mathcal{P}](\alpha^{hpb}[\mathcal{L} \times \mathcal{D}_\mathcal{P}]_\mathcal{C}(S))$.

Theorem 18. *If f is a property such that $f_s = \sup$, and the inductive property set $\mathcal{P}$ contains f, and $\mathcal{L} \times \mathcal{D}_\mathcal{P}$ is effectively inductive, $HPathBound[\mathcal{L} \times \mathcal{D}_\mathcal{P}]$ is monotonic for f.*

Intuitively, the theorem holds as every one-step refinement preserves the information about the property-bounds of nodes from the previous iteration. For all one-step refinements other than the vertical joining refinements, the monotonicity follows from the monotonicity of $PropComp$ and $ReduceBound$. For vertical splitting one-step refinements, the monotonicity holds as all the information that can be generated from the deleted children is already present in the parent node.

5.3 $HPathBound$ abstractions for control-flow graphs

A control-flow graph of a program written in a high-level language is contains many structures like loops and function calls. The traces of transition systems produced from such programs are structured. This opens the possibility of using hierarchical segment-based abstractions to analyze them.

Control-flow graphs and Programs. Following, e.g., [4], we abstract away from the syntax of a concrete programming language, and model programs as graphs whose nodes correspond to program locations. Here we assume simple programs with inlined function calls (this implies that there are no recursive functions).

Let V be a set of variables and let $D(V)$ be the combined range of variables in V. A *control-flow graph* (CFG) is a tuple of $\langle L, V, \Delta, \psi : \Delta \to 2^{D(V) \times D(V)} \rangle$ where L is a finite set of control locations, Δ is a transition relation, V is a set of variables, and ψ is a function from Δ to assertions over program variables V and their primed versions V'. The primed variables V' refer to the values of the variables after executing the transition. We assume that V contains a special variable wt ranging over $\mathbb{R}$ which denotes the weight of a particular state.

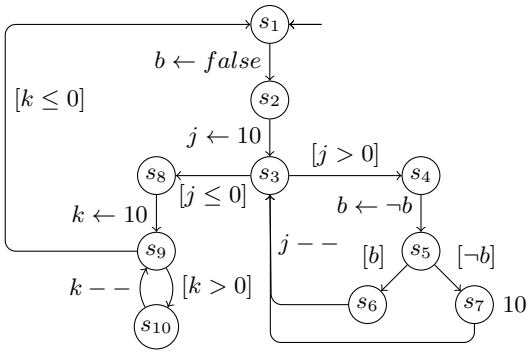


Figure 9: An example CFG

Given a CFG C, a corresponding transition system can be generated (with state-space $L \times D(V)$) in a standard way. We add the weight function $\varrho(l, d) = w$, where w is the value of the special variable wt in d, to turn the transition system into a WTS.

We assume the following about the CFG (these assumptions are valid for CFGs generated for programs in most programming languages): (a) *Reducibility.* In the graph (L, Δ), every maximal strongly connected component has a single entry and exit point, i.e., if $G \subseteq L$ is a maximal strongly connected component, there exists a node l_G such that $(l, l') \in \Delta \cap ((L \setminus G) \times G) \implies l' = l_G \wedge (l, l') \in \Delta \cap (G \times (L \setminus G)) \implies l = l_G$; (b) *Recursive reducibility.* Suppose $G \subseteq L$ is a maximal strongly connected component. The graph $(G, \Delta \cap (G \times (G \setminus \{l_G\}))$ is also reducible. Intuitively, the above conditions imply that loops in the CFG are single-entry and single-exit, and that they are nested.

```
while(true)
 b = false;
 j = 10;
 while j > 0
   b = not b;
   if b costlyOp;
   j--
 k = 10;
 while (k>0) k--
```

Figure 8: Prg

A *hierarchical control-flow graph* (HCFG) is a graph $\langle \mathcal{H}, \Delta \subseteq \mathcal{H} \times \mathcal{H}, h_i \rangle$ where each node $h \in \mathcal{H}$ is either a control-flow location, or a hierarchical control-flow graph. We call the first kind of nodes, *abstract state nodes* and the second kind of nodes *subgraph nodes*.

Any recursively reducible CFG C can be converted into a HCFG H of a special form using a standard algorithm on reducible graphs. In this special form, H and all sub-HCFGs occurring in H have the following property: either they are acyclic, or they have a single node with a self-loop on itself. Note that each loop of a program will correspond to such sub-HCFG. In what follows, we assume that all our HCFGs are of this special form.

Example 19. *Consider the example CFG shown in Figure 9 (generated from program in Figure 8). The equivalent HCFG of the special form to this CFG is shown in Figure 10. The statements on the transitions in Figure 10 are omitted for the sake of clarity. Each of the dotted boxes represent a HCFG. Intuitively, each loop has been separated out into an acyclic CFG containing the loop body and a single self-loop CFG.*

Inductive trace segment covers can be derived from HCFGs. In particular, every HCFG H represents a unique inductive trace segment cover $\mathcal{C}(H)$. Intuitively, the root of the inductive trace segment cover is the set of all traces of H, and the children of the root node are either (a) $\mathcal{C}(H')$ if H' is a subgraph node of H, and (b) $\{S\}$ if S is a abstract state node of H where $\{S\}$ is the SegmentSet containing all segments of length 1 with states in S. From now on, we use HCFGs and abstract inductive trace segment covers interchangeably.

Example 20. *Consider the program in Figure 8. Its CFG is in Figure 9 and the corresponding HCFG is in Figure 10. We use regular expressions over control locations as our abstract SegmentSet domain $\mathcal{L}$. The regular expressions have their intuitive meaning. For example, (a) the expression $s_1 s_2$ represents all segments $q_1 q_2$ such that the control location of q_1 (resp. q_2) is s_1 (resp. s_2); and (b) the expression $(s_1 s_2)^*$ represents the set of segments obtained by concatenation of segments from $s_1 s_2$.*

The HCFG corresponds to the following inductive trace segment cover $\mathcal{C}$ of the traces generated by the CFG. The root of $\mathcal{C}$ is the expression $H = (s_1 s_2 s_3 (s_4 s_5 (s_6 + s_7) s_3)^* s_8 s_9 (s_{10} s_9)^*)^*$. Its only child is $F = s_1 s_2 s_3 (s_4 s_5 (s_6 + s_7) s_3)^* s_8 s_9 (s_{10} s_9)^*$. The children of F are $\{C_B, L_1, C_M, L_2\}$ where $C_B = s_1 s_2 s_3$, $L_1 = (s_4 s_5 (s_6 + s_7) s_3)^*$, $C_M = s_8 s_9$, and $L_2 = (s_{10} s_9)^*$. Of these, only L_1 and L_2 are non-leaf nodes having one child each $C_{L_1} = s_4 s_5 (s_6 + s_7) s_3$ and $C_{L_2} = s_{10} s_9$, respectively.

Evaluating $limavg$ **for** $HPathBound$ **abstractions induced by HCFGs.** Let H be a HCFG of a special form. We compute the values $maxp$, $minp$, val^{pba}, and $hasInfPath$ using the technique of computing loop bounds. We consider a the domain of abstract bound pairs with elements of the form $\phi \wedge P$ as before.

Suppose H is a HCFG with a single node and a self-loop. Furthermore, let $\phi \wedge P$ be the corresponding abstract SegmentSet property bound in $\mathcal{C}(H)$, and let $\{\phi_1 \wedge P_1, \ldots, \phi_n \wedge P_n\}$ be its children. Let segment σ be in $\gamma(\phi \wedge P) \cap \biguplus^{\delta} \{\gamma(\phi_1 \wedge P_1), \ldots, \gamma(\phi_n \wedge P_n)\}$. Let $iters(\sigma) \subseteq \mathbb{N}$ where $n \in iters(\sigma)$ if and only if there exist $\sigma_0, \ldots, \sigma_{n-1}$ such that $\forall j < n . \exists i : \sigma_j \in \gamma(\phi_i \wedge P_i)$. Let $Iters(H) = \bigcup iters(\sigma)$ for all such σ. We define the *upper loop bound* (resp. *lower loop bound*), denoted by $ulb(H)$ (resp. $llb(H)$) as the value $\sup Iters(H)$ (resp. $\inf Iters(H)$). Note that there exists techniques to compute loop bounds using for example relational abstractions and ranking functions, see for example [10], and references therein.

Now, given the values of $maxp$, $minp$, val^{pba}, and $hasInfPath$ of subgraphs of an HCFG H, we inductively compute the value of the properties for H as follows:

- If H is a single node with no self-loop, i.e., an abstract state node, we have $maxp(H) = minp(H) = 1$, $val^{pba}(H) = \varrho(H)$, and $hasInfPath(H) = false$.
- If H is acyclic, we construct the following graph $H^\dagger$ as in Section 4.3, i.e., edges of H correspond to nodes of $H^\dagger$, and every node of H corresponds to two edges of weights (resp. lengths) $maxp(H') \cdot val^{pba}(H')$ and $maxp(H') \cdot val^{pba}(H')$ (resp. $maxp(H')$ and $minp(H')$). Now, $maxp(H)$ and $minp(H)$ are equal to the length of the longest and shortest paths in $H^\dagger$. Also, $hasInfPath(H) = true$ if and only if there is a node H' in H with $hasInfPath(H') = 1$. The value $val^{pba}(H)$ can be evaluated using Howard's policy iteration as in Section 4.3.
- If H is a single node with a self-loop, there is only one subgraph of H (say H'). We have $maxp(H) = ulb(H) \cdot maxp(H')$, $llb(H) \cdot minp(H')$, $val^{pba}(H) = val^{pba}(H')$, and $hasInfPath(H) = hasInfPath(H') \vee ulb(H) = \infty$.

Thus for such an HCFG, we can evaluate the limit-average value by using the inductive evaluation scheme from Section 5.2, i.e., using the values $minp$, $maxp$, val^{pba} and $hasInfPath$, we can inductively compute the limit-average values for an HCFG.

Example 21. *Consider the HCFG in Figure 10 and its inductive cover $\mathcal{C}$ from Example 20. We will illustrate a few steps of the inductive computation of the properties on $\mathcal{C}$. Fix $\mathcal{P} = \langle \langle minp \rangle, \langle maxp, val^{pba}, hasInfPath \rangle \rangle$.*

Let us start for the leaves of $\mathcal{C}$, i.e., $C_B = s_1 s_2 s_3$, $C_M = s_8 s_9$, $C_{L1} = s_4 s_5 (s_6 + s_7) s_3$, and $C_{L2} = s_{10} s_9$. Now, we can compute $minp$ and $maxp$ for these as the shortest and longest paths in the corresponding HCFGs. Furthermore, the $hasInfPath$ values for each of these is 0 as all of these CFGs are acyclic. The val^{pba} of

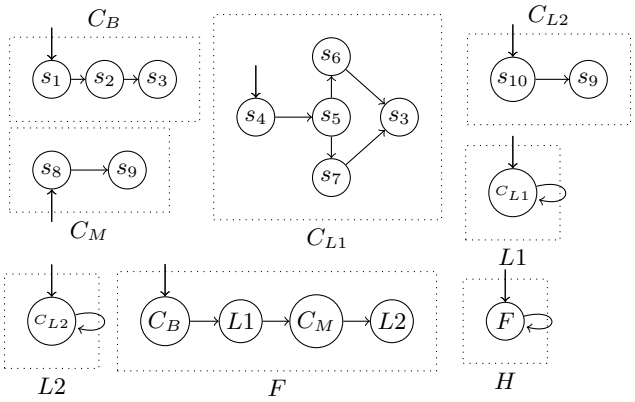


Figure 10: HCFG of a program

all the leaves except C_{L1} is 0 as the weights of all nodes except s_7 are 0. For C_{L1}, the val^{pba} can be computed to be $\frac{10}{4}$. Therefore, $ReduceBound(C_B) = ((3),(3,0,0))$, $ReduceBound(C_M) = ((2),(2,0,0))$, $P_1 = ReduceBound(C_{L1}) = ((4),(4,\frac{10}{4},0))$, and $P_2 = ReduceBound(C_{L2}) = ((2),(2,0,0))$.

Now, the nodes $L_1 = C^*_{L1}$ and $L_2 = C^*_{L2}$ have the children $\{C_{L1}\}$ and $\{C_{L2}\}$ respectively. The PropComp procedure has to now compute the properties $\mathcal{P}(L_1)$ using P_{L1}. However, note that the domain $\mathcal{L}$ does not give methods to compute loop-bounds (see Example 22 for a refinement of the domain $\mathcal{L}$ which can compute loop-bounds). Therefore, assuming ulb and llb for the loops are 0 and ∞ respectively, we get the values $\mathcal{P}(L_1) \leq P_{L1} = ((0),(\infty,\frac{10}{4},1))$ and $\mathcal{P}(L_2) \leq P_{L2} = ((0),(\infty,0,1))$.

Proceeding similarly, we get the estimates for $\mathcal{P}(F) \leq P_F = ((5),(\infty,\frac{10}{4},1))$ and $\mathcal{P}(H) \leq P_H = ((0),(\infty,\frac{10}{4},1))$. Intuitively, this corresponds to a counterexample that ends with an infinite loop in L1 that contains the costly operation.

6. Quantitative refinements

In this section, we present quantitative refinement algorithms for the state-based and segment-based abstraction schemes.

6.1 Algorithmic Refinement of $ExistMax$ abstractions

For the state-based abstraction scheme $ExistMax$, we give an algorithm for counterexample-guided abstraction refinement (CEGAR) for quantitative properties. The algorithm is based on the classical CEGAR algorithm [2], which we extend to the quantitative case. Here (as in [2]), we assume that the concrete system is finite-state, and obtain a sound and complete algorithm. In the infinite-state case, the algorithm is sound, but incomplete.

Let $S = (Q, \delta, \rho, q_\iota)$, let f be a $(\leq_p, \sqsubseteq)$-monotonic quantitative property that admits memoryless extremal counterexamples (as stated in Section 2, we restrict ourselves to these properties), and let $\approx_1$ be an equivalence relation on Q. Let us further assume that $S^\alpha = (Q^\alpha, \delta^\alpha, \rho^\alpha, q_\iota^\alpha)$ is the result of applying the $ExistMax$ abstraction parameterized by $\approx_1$ on S. Let ρ_{ext} be the memoryless counterexample of S^α which realizes the value $f(S^\alpha)$.

Algorithm 2 is a refinement procedure for $ExistMax$ abstractions. Its input consists of the concrete and abstract systems, the equivalence relation that parameterizes the abstraction, the quantitative property, and the extremal trace ρ_{ext}. As the extremal counterexample is memoryless, it is of the shape $H_1 \dots H_k(H_{k+1} \dots H_n)^\omega$. The output of the algorithm is either a concrete counterexample (if one corresponding to ρ_{ext} exists), or a refined equivalence relation (which can be used to produce a new abstract system).

Let us consider a set of traces $\gamma(\rho_{ext})$ of the system S that correspond to an abstract memoryless trace $\rho_{ext} = H_1 \dots H_k(H_{k+1} \dots H_n)^\omega$. The first observation is that checking whether a concrete counterexample exists, i.e., whether $\gamma(\rho_{ext})$ is non-empty, can be done by checking whether a finite abstract trace ρ_u corresponds to a concrete trace. The finite abstract trace ρ_u can be obtained by unwinding the loop part of ρ_{ext} m number of times, where m is the size of the smallest abstract state in the loop part of ρ_{ext}, or formally, $m \leftarrow \min\{|H_i| \mid k+1 \leq i \leq n\}$ (line 1). This result can easily be adapted from [2] to the quantitative case.

Algorithm 2 Refinement for $ExistMax$

Input: WTS $S = (Q, \delta, \rho, q_\iota)$, quant. prop. f, eq. rel. $\approx_1$,
abstract WTS $S^\alpha = (Q^\alpha, \delta^\alpha, \rho^\alpha, q_\iota^\alpha)$,
abstract counterexample $\rho_{ext} = H_1 \dots H_k(H_{k+1} \dots H_n)^\omega$
Output: refined eq. rel. $\approx_2$
or a concrete counterex. $tecx$

1: $m \leftarrow \min\{|\gamma(H_i)| \mid k+1 \leq i \leq n\}$
2: $\rho_u \leftarrow \text{unwind}(\rho_{ext}, m, k, n)$
{we have $\rho_u = G_1 \dots G_{k+(n-k)\cdot m}$}
3: $R_1 \leftarrow \gamma(\sigma_1) \cap \{q_\iota\}$
4: $i \leftarrow 1$
5: **while** $R_i \neq \emptyset \wedge i < (k + (n-k)\cdot m)$ **do**
6: $\quad R_{i+1} \leftarrow post(R_i, \rho^\alpha(G_{i+1})) \cap \gamma(G_{i+1})$
7: $\quad i \leftarrow i + 1$
8: **if** $R_{i-1} \neq \emptyset$ **then**
9: $\quad$ **return** counterEx$(R_0, \dots, R_{i-1})$
10: **else**
11: $\quad U \leftarrow \{s \in \gamma(R_{i-1}) \mid \exists s' \in \gamma(G_i) : \delta_S(s, s') \wedge \rho(s') = \rho^\alpha(G_i)\}$
12: $\quad$ **return** refine($\approx_1$,G_{i-1},U)

Lines 3 to 7 traverse the finite abstract trace ρ_u, and at each step maintain the set of states that are reachable form the initial state along a path corresponding to ρ_u. (The $post$ operator in line 6 takes as input a set of concrete states L and a weight w, and returns all the successors of states in L that have weight w.)

If the traversal finishes because at i-th step the set R_i is empty, then the algorithm refines the equivalence relation $\approx_1$ by splitting the equivalence class given by G_{i-1} into U and $G_{i-1} \setminus U$ (line 11). The set U contains those states that have a transition (corresponding to a transition in ρ_u) to a state with weight $\rho^\alpha(G_i)$. The intersection $U \cap R_{i-1}$ is empty, because R_i is empty. Thus separating U leads to eliminating the counterexample from the abstract system.

If the traversal finishes a pass over the whole trace ρ_u, it can construct a concrete counterexample using sets of states $R_0, \dots, R_{i-1}$ (line 9).

We have thus extended the classical CEGAR algorithm [2] to the quantitative case. The extension is simple, the main difference is in taking into account the weights in lines 6 and 11.

6.2 Algorithmic Refinement of $HPathBound$ abstractions

In this subsection, we describe an algorithmic technique for refinement of segment-based abstractions. We assume that the abstract bound pair domain is precisely inductive.

We assume that the extremal counter-example from the evaluation of a $HPathBound$ abstraction is returned as a abstract hierarchical trace $\psi = \langle(\phi_0 \wedge P_0)(\phi_1 \wedge P_1) \dots (\phi_k \wedge P_k)((\phi_{k+1} \wedge P_{k+1}) \dots (\phi_n \wedge P_n))^\omega, sub_\psi\rangle$. Note that we can assume a lasso-shaped counter-example due to the memoryless property of the quantitative properties we consider. Furthermore, without loss of generality we also assume that every leaf in the abstract trace segment cover is composed of segments of length 1.

The basic structure of the refinement algorithm is same as in $ExistMax$. However, the main difference is in the $post$ operator.

For hierarchical traces, we define a non-deterministic *post* operator in Algorithm 3. Intuitively, the algorithm non-deterministically chooses a level of the hierarchical trace to perform the analysis. First, given a hierarchical trace of length 1, *post* operator computes (Lines 3 and 4) the set $\gamma(\phi_0 \wedge P_0) \circ \delta = \{q\sigma \mid \sigma \in \gamma(\phi_0 \wedge P_0) \wedge (q, \mathit{first}(\sigma)) \in \delta\}$. Then, it computes the top-level post set R^* of states reachable from R using the segments from $\gamma(\phi_0 \wedge P_0) \circ \delta$. Now, non-deterministically (Line 5) it chooses whether to descend into the next level of the hierarchy. If it decides to, the set of post states $R^\dagger$ is computed from the levels below, and then the strengthening of R^* by $R^\dagger$, i.e., $R^* \cap R^\dagger$ is returned.

Assume that the *post* computation is done at a particular level, i.e., the level below is not used. Intuitively, this means that all the segments in $\phi_0 \wedge P_0$ are assumed to be valid segments, and the property bounds are assumed to be tight, i.e., the part of the counter-example corresponding to $\phi_0 \wedge P_0$ is considered non-spurious. Note that in the case where the algorithm always descends to the lowest level, the set returned is exactly the set of states reachable using segments in $\gamma(\psi) \circ \delta$. We also remark that nondeterminism in Algorithm 3 can be instantiated in a manner suitable for a particular domain.

Algorithm 3 Counterexample analysis for $\mathit{HPathBound}$

Input: Hierarchical trace $\psi = \langle(\phi_0 \wedge P_0) \dots (\phi_k \wedge P_k), \mathit{sub}_\psi\rangle$, Concrete set of states R,
Output: Over-approximation of states reachable through segments in $\gamma(\psi)$.

1: **if** $n > 1$ **then**
2: **return** $\mathit{post}((\phi_1 \wedge P_1) \dots (\phi_n \wedge P_n), \mathit{sub}_\psi\rangle, \mathit{post}(\langle\phi_0 \wedge P_0, \mathit{sub}_\psi\rangle, R))$
3: $T \leftarrow \gamma(\phi_0 \wedge P_0) \circ \delta$
4: $R^* \leftarrow \{q' \mid \exists q\sigma q' \in T : q \in R\}$
5: **if** $*$ **then**
6: **if** $\mathit{sub}_\psi(0) \neq \bot$ **then**
7: $R^\dagger \leftarrow \mathit{post}(\mathit{sub}_\psi(0), R)$
8: **else**
9: $R^\dagger \leftarrow R^*$
10: **return** $(R^* \cap R^\dagger, R^*, R^\dagger)$

Let $\mathcal{C}$ be the inductive trace segment cover and let $\psi = \langle(\phi_0 \wedge P_0)(\phi_1 \wedge P_1) \dots (\phi_k \wedge P_k)((\phi_{k+1} \wedge P_{k+1}) \dots (\phi_n \wedge P_n))^\omega, \mathit{sub}_\psi\rangle$ be the extremal abstract trace. The abstraction refinement procedure $\mathit{hAbsRefine}(\mathcal{C}, \psi, R_0)$ proceeds similarly to Algorithm 2 as follows:

- The abstract hierarchical trace ψ is unwound m number of times, where $m = \min\{|\gamma(\phi_i \wedge P_i)| \mid i \in \{k+1, \dots, n\}\}$;
- Let R_0 be the set of concrete initial states. For each abstract SegmentSet property pair $\phi_i \wedge P_i$ in the unwound trace, we compute $(R_{i+1}, R^*, R^\dagger) = \mathit{post}(\langle\phi_i \wedge P_i, \mathit{sub}_\psi\rangle, R_i)$;
- If at any step $R_{i+1} = \emptyset$, we refine the inductive trace segment cover $\mathcal{C}$ using set R_i and $\langle\phi_i, \mathit{sub}_\rho\rangle$ as $\mathit{hRefine}(R_i, \langle\phi_i, \mathit{sub}_\rho\rangle)$ (explained below). Otherwise, return any concrete counter-example constructed from the set $\{R_0, R_1, \dots\}$.

We describe the computation of $\mathit{hRefine}(R_i, \langle\phi_i \wedge P_i, \mathit{sub}_\rho\rangle)$ when $\mathit{post}(\langle\phi_i, \mathit{sub}_\rho\rangle, R_i) = \emptyset$: during the computation $\mathit{post}(\langle\phi_i, \mathit{sub}_\rho\rangle, R_i)$ (execution of Algorithm 3) we have at least one of the following cases based on the values of R^* and $R^\dagger$. First, we define $T_R = \{\sigma \mid \exists q \in R(r, \mathit{first}(\sigma)) \in \delta \wedge \sigma \in \gamma(\phi_i \wedge P_i)\}$ and $T_R^{\mathit{below}} = \{\sigma \mid \exists r \in R.(r, \mathit{first}(\sigma)) \in \delta \wedge \sigma \in \gamma(\mathit{sub}_\psi(i))\}$. Intuitively, T_R is the set of concrete segments from R in $\phi_i \wedge P_i$, and T_R^{below} is the set of concrete segments from R generated from the hierarchical levels under $\phi_i \wedge P_i$. We have one of the following options:

- $R^* = \emptyset \wedge R^\dagger = \emptyset$. In this case, the refinement returned is $\mathit{hAbsRefine}(\mathcal{C}, \langle\mathit{sub}_\rho(i), \mathit{sub}_\rho\rangle, R)$, i.e., we run the abstraction refinement procedure on the lower level, starting from the concrete set of states R.
- $R^* = \emptyset$. In this case, we perform a horizontal refinement to separate the sets T_R and $T \setminus T_R$, i.e., the node labelled $\phi_i \wedge P_i$ is split into $\phi^A \wedge P^A$ and $\phi^B \wedge P^B$ where $\gamma(\phi^A \wedge P^A) \cup \gamma(\phi^B \wedge P^B) = \gamma(\phi_i \wedge P_i)$ and $T_R \subseteq \gamma(\phi^A \wedge P^A) \wedge T_R \cap \gamma(\phi^B \wedge P_B) = \emptyset$. Intuitively, we are separating segments in $\phi_i \wedge P_i$ that are from R from those that are not from R.
- $R^* \neq \emptyset \wedge R^* \cap R^\dagger = \emptyset$. In this case, we perform multiple simultaneous refinements. The segment sets which need to be distinguished from each other are $T \setminus (T_R \cup T_R^{\mathit{below}})$, T_R and T_R^{below}. Intuitively, we are trying to separate the segments in $\gamma(\phi_i \wedge P_i)$ that (a) do not start from R (i.e., $T \setminus (T_R \cup T_R^{\mathit{below}})$), (b) those that start from R and are validly generated from the levels below (i.e., T_R^{below}); and (c) those that do start from R and are not validly generated from the levels below (i.e., T_R). Formally, let $\phi^A \wedge P^A$, $\phi^B \wedge P^B$, and $\phi^C \wedge P^C$ be such that:
  - $T \subseteq \gamma(\phi^A \wedge P^A) \cup \gamma(\phi^B \wedge P^B) \cup \gamma(\phi^C \wedge P^C)$;
  - $T_R \subseteq \gamma(\phi^A \wedge P^A) \wedge T_R^{\mathit{below}} \cap \gamma(\phi^A \wedge P^A) = \emptyset$;
  - $T_R^{\mathit{below}} \subseteq \gamma(\phi^B \wedge P^B) \wedge T_R \cap \gamma(\phi^B \wedge P^B) = \emptyset$; and
  - $(T_R \cup T_R^{\mathit{below}}) \cap \gamma(\phi^C \wedge P^C) = \emptyset$.

  First, we do a horizontal refinement splitting the node $\phi_i \wedge P_i$ into $\phi^A \wedge P^A$, $\phi^B \wedge P^B$, and $\phi^C \wedge P^C$. Second, in the subtree rooted at $\phi^A \wedge P^A$, the levels below contains the information that T_R is infeasible, but the root does not. So, we perform upward strengthening refinements till the root contains the same information. Third, in the subtree rooted at $\phi^B \wedge P^B$, the root contains the information that T_R^{below} is infeasible, but the levels below do not. So, we perform either (a) downward strengthening refinements till the levels below contain the same information; or (b) vertical joining refinements till there are no levels below. Note that if one of $\phi^A \wedge P^A$, $\phi^B \wedge P^B$, or $\phi^B \wedge P^B$ is empty, we omit it.

Example 22. *Consider the HCFG H and the corresponding abstract trace segment cover $\mathcal{C}$ from Example 21. We now show some examples of hierarchical counter-example guided refinements for computing the limit-average value.*

We work in a more powerful refined domain than in Example 21, one that allows computation of loop bounds. Let $\mathcal{L}$ be the domain of regular expressions over $HCFG$'s along with a relation between the values of the variables in the initial and final states. For example, the expression $((s_4 s_5 (s_6 + s_7) s_3, b' = \neg b)$ represents the set of segments which match $s_4 s_5 (s_6 + s_7) s_3$ and have the value of b is the last state is negation of the value of b in the first state.

Let us first start with abstract trace segment cover $\mathcal{C}$ from 21. A part of the abstract extremal trace generated from $\mathcal{C}$ will be $\langle L_1, \mathit{sub}\rangle$ where (a) $\mathit{sub}(1) = \langle(C_{L1})^\omega, \mathit{sub}'\rangle$, where (b) $\mathit{sub}'(i) = s_4 s_5 s_7 s_3$ for all $i \geq 1$. We will illustrate two refinement steps that might occur:

- *Suppose during the refinement process we are computing $\mathit{post}(\langle L_1, \mathit{sub}'\rangle, R)$ where R is the set of states at location s_3 with $j = 0$. Now, we can perform the analysis either at the top level or at the lower level:*
  - *At the top level, we get the post states to be R^* where the control location of a state is in s_3.*
  - *At the lower level, we get the post states to be $R^\dagger = \emptyset$ as the transition from s_3 to s_4 is disabled due to j being 0.*

  Therefore, we need to refine the abstract SegmentSet $s_4 s_5 (s_6 + s_7) s_3$. One possible valid refinement is to strengthen the set $s_4 s_5 (s_6 + s_7) s_3$ to $s_4 s_5 (s_6 + s_7) s_3, j > 0 \wedge j' = j - 1$. Using this strengthened set, we can compute that the upper and lower loop bounds for L_1 are 10. This leads to a improvement in the

value of the system as now, there is no infinite path in the high value segment L_1. The new value of the system is $\frac{20}{9}$.

- *Suppose during the refinement process we are computing $post(\langle L_1 \wedge P, sub' \rangle, R)$ where P bounds the limit-average of segments in L_1 to $\frac{10}{4}$, R is the set of states at location s_3 with $b = true$. Again, performing the analysis at top level produces $R^* = \{s_7\}$, but the lower level produces $R^\dagger$ where $R^\dagger = \emptyset$. Therefore, we can to refine the abstract SegmentSet $s_4 s_5 (s_6 + s_7) s_3$ and one possible refinement is a horizontal split into $s_4 s_5 s_6 s_3, b = true \wedge b' = false$ and $s_4 s_5 s_7 s_3, b = false \wedge b' = true$. Performing this refinement reduces the value of L_1 to $\frac{20}{9}$ and hence, by upward strengthening the value of the whole system to $\frac{20}{9}$.*

7. Case study: WCET analysis

We present a case study to demonstrate anytime verification, and to evaluate $ExistMax$ and hierarchical $PathBound$ abstractions. Worst-case execution time (WCET) estimation is an important part of verifying real-time systems. We only study one aspect, i.e., cache behavior prediction. In a processor, clock cycles needed for an instruction can vary by factors of 10-100 based on cache hits vs misses. Assuming the worst-case for all instructions leads to bad WCET estimates. Abstract domains for cache behavior prediction are well studied (e.g., [9, 18]). However, we know of no work on automated refinement for these abstractions. Note that this case study and the accompanying implementation is not a complete WCET analysis tool, but a prototype used to illustrate the quantitative abstraction refinement approach. Our intention is just to evaluate the anytime aspect of our approach.

We estimate WCET using the limit-average property. Intuitively, we put the whole program in a nonterminating loop. The limit-average then corresponds to the average cost of an instruction in the worst-case execution of a loop. (For a terminating program, it is the execution of the artificial outer loop.) WCET is estimated by multiplying the limit-average values with (an over-approximation of) the length of the longest trace. We report limit-average values instead of WCETs.

The cache model and the abstract domain used are from [9]. Multiple abstract domains are possible based on the tracked locations. If no set is tracked, every memory access is deemed a cache-miss; whereas the invariant computation is very expensive in the domain which tracks all cache-sets. Here, we start from the empty cache and refine by tracking more cache-sets as necessary.

7.1 Implementation details

We implemented a WCET analyzer based on the presented techniques in a tool QUART that analyzes x86 binaries.

Static analysis. We analyze the binary and produce the control flow graph. Instructions in the program may operate on non-constant addresses (for example, array indexing leads to variable offsets from a fixed address). However, if the exact addresses cannot be computed statically, we perform no further analysis and assume that the memory access is a cache miss. This restriction comes from the cache abstract domain we use from [9].

Worst-case computation. In the resulting graph, we annotate states with invariants from the current cache abstract domain. From the invariants, we compute costs of each transition (we use costs of 1 and 25 cycles for cache-hits and cache-misses, respectively). We then find the worst-case using techniques of Section 3 and Section 4.3 to find a the worst-case limit-average value. Furthermore, we implemented the extension of the algorithm to graphs with both edge weights and edge lengths [3].

Refinement. We analyze worst-case counter-example ext:

Feasibility analysis. We first check if ext is a valid program execution (ignoring the cache) using a custom symbolic execution computation. If ext is not a valid execution, we refine the abstract graph using standard CEGAR techniques.

Cache Analysis. If ext is valid, we compute the concrete cache states for it. If the concrete value obtained is the same as that of ext, we return the concrete trace.

Refinement heuristic. Otherwise, of all locations accessed in the loop of ext, we find the one with most abstract cache misses which are concrete cache hits. The current cache abstract domain is refined by additionally tracking this location.

Fall-back refinement. If all the locations accessed in ext are already being tracked, we use Algorithm 2 and the algorithm given by $hAbsRefine$ to do the refinement.

Example	Step	Value	Time (ms)	Tracked
Basic Example	0	14.14	1240	
	1	6.50	2102	i
	2	4.87	2675	a
	3	4.75	3275	b
	4	1.27	3864	c
	5	1.03	4631	v
Binary search	0	15.77	908	
	1	11.15	1130	m
	2	8.23	1369	r
	3	5.0	1707	l
	4	3.76	1895	s
	5	3.0	2211	a[$\frac{(N-1)}{2}$]
	6	2.97	2527	a[$\frac{(N-3)}{4}$]
	7	2.85	3071	a[$\frac{(3N-1)}{4}$]
Polynomial Eval.	0	15.49	524	
	1	8.13	759	i
	2	4.45	1025	val
	3	2.95	1237	x
GCD	0	13.76	289	
	1	9.47	399	inp2
	2	6.65	472	inp1
	3	6.33	536	temp

Table 1: $ExistMax$ abstraction results

7.2 Evaluation of $ExistMax$ abstraction

For evaluating the $ExistMax$ abstraction and refinement methods, we consider binaries for five (small) C programs, including the example from the introduction (called Basic example in the table). The results are in Table 1. For each example program, the table contains lines, with each corresponding to a refinement step. For each refinement step we report the current estimate for the limit-average value, the running time for the analysis (cumulative; in milliseconds) and in case the refinement enlarged the abstract cache, we also show what new memory locations correspond to the entries in the abstract cache. In each case, the over-approximated limit-average value decreases monotonically as the tool is given more time.

```
while(true)
 input(s);
 l = 0; r = N - 1;
 do {
  m = l + r / 2;
  if(s > a[m])
   l = m + 1;
  else
   r = m - 1;
 } while(l <= r
     ∧ a[m] != s)
```

Figure 11: Bin. Search

Binary search. We analyze a procedure (Figure 11) that repeatedly performs binary search for different inputs on a given array. We start with the empty abstract cache domain and all behaviors have high values (with worst-case value 15.77). In the ext-trace, variable `m`, accessed 4 times every iteration of the inner loop, causes most spurious cache misses.

Using the *Refinement heuristic* we heuristically choose the location of `m` is additionally tracked in the cache abstract domain reducing the value to 11.15. Indices `l`, `r` and the input `s` are the next most frequently used, and are added subsequently to the cache abstract domain. More importantly, the most used array elements are added in order. During binary search, the element at position $N/2$ is accessed always, and the elements at $N/4$ and $3N/4$ half the times, and so on. The refinements obtained add these array elements in order. This illustrates the anytime aspect: refinement can be stopped at any point to obtain an over-approximation of the value.

7.3 Evaluation of the hierarchical $PathBound$ abstraction

For evaluating the $PathBound$ abstraction refinement procedure, we picked 4 benchmarks from the collection of WCET benchmarks in [11]. These benchmarks were larger than the ones for the $ExistMax$ evaluation with around 150-400 lines of code each. The benchmarks we picked included a simple program which scanned a matrix and counted elements, matrix multiplication, and two versions of discrete-cosine transformations.

Bench mark	Step	Value	Time (ms)
cnt	0	8.74	1810
	3	8.64	6349
	4	4.08	8298
matmult	0	8.73	4669
	2	8.71	15660
	5	8.71	30408
	6	4.17	35676
fdct	0	6.88	2142
	1	1.94	4274
	2	1.76	6685
jfdctint	0	6.95	3095
	1	3.35	5759
	2	1.89	8674
	3	1.57	11809

Table 2: $PathBound$ abstraction results

We used the hierarchical $PathBound$ abstraction-refinement algorithm, i.e., the algorithm given by $hRefine$. We note that we do not perform any cache refinements. Nevertheless, the hierarchical aspect of hierarchical $PathBound$ abstraction was evaluated, as three of the benchmarks contained a number of nested loops. The challenge addressed was to obtain good (and monotonically decreasing) estimates on WCET, as the abstraction is refined.

We summarize the results in Table 2. For each example program, the table contains a number of lines, with each line corresponding to a refinement step. For each refinement step we show the current estimate for the limit-average value, and the running time for the analysis (cumulative; in milliseconds). As it can be seen, the limit-average values monotonically decrease with longer execution time. It should be noted that for most of these programs, to obtain similar values with the $ExistMax$ approach, one would need to perform a large number (in thousands) of counter-example guided refinements (as the nested loops would have to be unrolled).

8. Conclusion

Summary. This paper makes four main contributions. First, we present a general framework for abstraction and refinement with respect to quantitative system properties. Refinements for quantitative abstractions have not been studied before.Second, we propose both state-based and segment-based quantitative abstraction schemes. Quantitative segment-based abstractions are entirely novel, to the best of our knowledge. Third, we present algorithms for the automated refinement of quantitative abstractions, achieving the monotonic over-approximation property that enables anytime verification. Fourth, we implement refinement algorithms for WCET analysis of executables, in order to demonstrate the anytime verification property of our analysis, and to investigate trade-offs between the proposed abstractions.

Future work. There are several directions for future work. The first is to perform larger-scale case studies, for instance for worst-case execution time analysis, with more realistic architecture models. Second, quantitative abstraction can aid partial-program synthesis, as quantitative reasoning is necessary if the goal is not to synthesize any program, but rather the best performing program according to quantitative measures such as performance or robustness. Furthermore, the anytime verification property of the refinements we proposed can lead to *anytime synthesis* methods, that is, methods that would synthesize correct programs, and refine these into more optimized versions if given more time.

References

[1] M. Boddy. Anytime problem solving using dynamic programming. In *AAAI*, pages 738–743, 1991.

[2] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5):752–794, 2003.

[3] J. Cochet-Terrasson, G. Cohen, S. Gaubert, M. McGettrick, and J.-P. Quadrat. Numerical computation of spectral elements in max-plus algebra, 1998.

[4] B. Cook, A. Podelski, and A. Rybalchenko. Abstraction refinement for termination. In *SAS*, pages 87–101, 2005.

[5] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252, 1977.

[6] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *POPL*, pages 269–282, 1979.

[7] P. Cousot and R. Cousot. An abstract interpretation framework for termination. In *POPL*, pages 245–258, 2012.

[8] L. de Alfaro and P. Roy. Magnifying-lens abstraction for Markov decision processes. In *CAV*, pages 325–338, 2007.

[9] C. Ferdinand, F. Martin, R. Wilhelm, and M. Alt. Cache behavior prediction by abstract interpretation. *Sci. Comput. Program.*, 35(2):163–189, 1999.

[10] S. Gulwani and F. Zuleger. The reachability-bound problem. In *PLDI*, pages 292–304, 2010.

[11] J. Gustafsson, A. Betts, A. Ermedahl, and B. Lisper. The mälardalen WCET benchmarks: Past, present and future. In *WCET*, pages 136–146, 2010.

[12] H. Hermanns, B. Wachter, and L. Zhang. Probabilistic CEGAR. In *CAV*, pages 162–175, 2008.

[13] M. Kattenbelt, M. Kwiatkowska, G. Norman, and D. Parker. Abstraction refinement for probabilistic software. In *VMCAI*, pages 182–197, 2009.

[14] A. Podelski and A. Rybalchenko. Transition predicate abstraction and fair termination. In *POPL*, pages 132–144, 2005.

[15] A. Prantl, M. Schordan, and J. Knoop. TuBound - a conceptually new tool for worst-case execution time analysis. In *WCET*, 2008.

[16] N. Shankar. A tool bus for anytime verification. Usable Verification, 2010.

[17] V. Tiwari, S. Malik, and A. Wolfe. Power analysis of embedded software: a first step towards software power minimization. In *ICCAD*, pages 384–390, 1994.

[18] R. Wilhelm, S. Altmeyer, C. Burguière, D. Grund, J. Herter, J. Reineke, B. Wachter, and S. Wilhelm. Static timing analysis for hard real-time systems. In *VMCAI*, pages 3–22, 2010.

[19] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The worst-case execution-time problem - overview of methods and survey of tools. *ACM Trans. Embedded Comput. Syst.*, 7(3), 2008.