

Property Refinement in Computation Tree Logic

Luca Brodo, Giuseppe Scalora, Stefan Henkler

University of Applied Sciences Hamm-Lippstadt
name.last_name@hshl.de

Abstract. Safety-critical systems rely on formal methods to verify behavioral properties, often expressed in Computation Tree Logic (CTL). As these systems evolve, their specifications accumulate redundant or subsumed properties, a phenomenon known as specification debt, inflating verification effort without improving assurance. Property refinement, which captures when one property is stronger than another, offers a principled way to eliminate such redundancy. A prior extension to CTL relied on a simulation preorder over tree automata, yielding a sound but incomplete check that only over-approximates the minimal sufficient set of properties. We close this gap with a complete theory of property refinement for CTL. We show that refinement is decidable, forms a pre-order, whose characteristics we leverage to propose a complete algorithm for computing the minimal sufficient set. Evaluated on more than 3,000 suites comprising over 52,000 properties, our approach proves both effective (up to 27% average reduction) and efficient (consistently lower 70 ms per refinement check), reducing the number of expensive satisfiability checks by up to a factor of six compared to prior work.

Keywords: Property Refinement · Computation Tree Logic · Specification Debt

1 Introduction

Safety-critical software systems rely on formal methods to guarantee correctness against behavioral properties, frequently expressed in Computation Tree Logic (CTL) [6,9] for its branching-time expressiveness and its efficient model checking. As such systems evolve through iterative development across distributed teams, feature expansion, and changing regulations, their specifications tend to accumulate redundant or subsumed properties, a phenomenon often referred to as *specification debt* [2,19]. This redundancy inflates verification and maintenance effort without improving assurance.

Recently, the notion of *property refinement* was introduced for Linear Temporal Logic (LTL) [3] to face this challenge. This relation captures the intuitive idea that one property is stronger than another, i.e., that verifying the stronger property guarantees the weaker one. Building on this relation, [3] enriches the verification pipeline with an additional automated step, the *Refinement Analysis* (see Figure 1), which computes a minimal sufficient subset of properties whose verification entails the entire suite. This is shown to yield measurable reductions

in verification and management effort, together with qualitative insight into the health of a requirements suite [3].

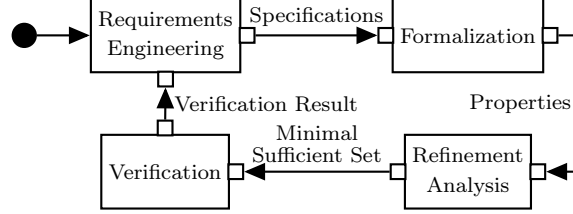


Fig. 1: The Verification Workflow Enriched with Property Refinement Analysis

An earlier extension of this concept to CTL was proposed in [4], which decides refinement via a *simulation* preorder on the tree automata characterizing CTL properties. A simulation between two automata implies language inclusion, but the converse does not hold; the simulation-based check is therefore *sound*—it never reports a refinement that does not hold—but *incomplete*, missing genuine refinements, in exchange for running in polynomial time. Because it under-approximates the refinement relation, the minimal sufficient set it produces is only an over-approximation: redundant properties may be retained and verified unnecessarily.

In this paper we close this gap by developing a complete theory of property refinement for CTL. We first show that the refinement relation remains a pre-order also for CTL, and that it is a lattice modulo logical equivalence. Leveraging its properties, we then develop a complete algorithm for computing the minimal sufficient set of properties based on tableau-based CTL satisfiability checking.

The remainder of this paper is structured as follows. Section 3 reviews related work. Section 4 introduces our formal framework for CTL refinement. Section 5 details our algorithm for computing the minimal sufficient set. Section 6 presents our experimental evaluation, and Section 7 concludes the paper.

2 Motivating Example

Consider an agricultural monitoring system in which a fleet of autonomous drones surveys crop fields to collect imagery and sensor data. Each drone D_i operates under a scheduling policy ensuring periodic coverage of assigned zones while avoiding collisions and maintaining battery safety. During system design, two requirements are identified. The first, φ_1 , is a liveness property ensuring that every assigned zone z_i is eventually scanned by its designated drone ($\varphi_1 = \text{AG}(z_i = \neg\text{scanned} \rightarrow \text{AF}(z_i = \text{scanned}))$). The second, φ_2 , ensures that drones scan zones only when they have sufficient battery ($\varphi_2 = \text{AG}(z_i = \neg\text{scanned} \rightarrow \text{AF}(\text{battery}_i > \theta \wedge z_i = \text{scanned}))$). These properties have been introduced independently, e.g. by different teams focusing on coverage and safety,

or through incremental requirements gathering. In a later iteration, a third property φ_3 ensures that drones also retain enough energy to return to base b_i safely afterwards: $\varphi_3 = \text{AG}((\text{battery}_i > \theta) \rightarrow \text{AF}(z_i = \text{scanned} \wedge \text{drone}_i = b_i))$. By combining coverage, safety, and return-to-base into a single specification, φ_3 refines the previous properties. This relation, however, is not immediately obvious and could be overlooked in large specification suites, leading to redundancy and motivating a systematic approach to identify and reason about such relationships.

3 Related Work and Background

CTL [6] is a branching-time logic for reactive systems; for its syntax and semantics we refer to [6,7,1]. We write $\text{SAT}(\varphi)$ when a CTL formula φ is satisfiable and $\text{UNSAT}(\varphi)$ otherwise. We decide refinement through a tableau-based CTL satisfiability procedure [9,18,17]. When models exist, refinement could be checked via model checking with tools such as NuSMV [5], but our tableau approach is model-independent and applies to properties in isolation.

Early ideas regarding the concept of property refinement can be found in the work of Pnueli [16] and Mokkedem and Méry [14,15], although they lacked a rigorous temporal-logic formalization. Brodo et al. [3] define a refinement relation over LTL and give a complete algorithm for computing a minimal sufficient set. For CTL, a subsequent extension [4] uses a simulation preorder to compute, in polynomial time, only an over-approximation of that set; the resulting check is sound but incomplete. We close this gap with a complete theory of CTL property refinement and a complete algorithm for the minimal sufficient set. This lineage grows out of automata-theoretic refinement via simulation and bisimulation [7]: model-level refinement is mature in compositional verification, but its lifting to specifications remains underexplored.

A related notion is property subsumption, studied mainly for LTL [2,8]. Dureja and Rozier [8] build dependency graphs via LTL-SAT implication to schedule which properties need explicit checks on which models at the Verification stage (cf. Figure 1); Barnat et al. [2] identify redundancy but scale exponentially in the number of properties, becoming impractical beyond about ten. In contrast, refinement is stronger than implication: it captures behavioral containment and excludes vacuous entailments, so removing a refined property preserves correctness. We exploit transitivity and equivalence classes for soundness and scalability, unaddressed in [8], and our analysis is model-agnostic, enabling early-stage reasoning without a system model, whereas [8] relies on model-checking runs to handle vacuous and trivially-true properties.

In requirements engineering, redundancy is more commonly addressed at the natural-language level through clone detection, similarity analysis, and learning-based techniques [?, ?, ?, ?, ?]. These approaches are useful for identifying duplicate or semantically similar requirements, but they do not establish that verifying one formal property, or a conjunction of properties, is sufficient to guarantee another. Other lines of work target related specification-quality problems, in-

cluding vacuity detection [?, ?, ?, ?, ?], consistency checking [?], specification mining [?, ?], and formula simplification [?, ?]. These techniques respectively expose irrelevant subformulas, inconsistent requirements, missing properties, or unnecessarily complex formulas. Our setting is different: we reason over an existing LTL suite and exploit semantic containment across distinct properties to compute sufficient verification subsets. This makes refinement a mechanism for both redundancy elimination and verification planning, rather than only a relation between development artefacts or a measure of textual similarity.

4 CTL Property Refinement and its Properties

The refinement relation allows us to understand how a more concrete or detailed property (e.g., arising from a subsequent design iteration) relates to a more abstract one (e.g., from an initial design or high-level requirement). We establish the refinement relation in terms of the satisfaction relation over states in transition systems. Following this, we establish fundamental structural properties of this refinement relation, such as reflexivity, transitivity and anti-symmetry. These results lay the groundwork for reasoning about system behavior under refinement, as well as for modular verification where properties evolve or are composed incrementally. We will assume a transition system $\mathcal{T}$, a fixed set of CTL properties $\Phi = \{\varphi_1, \dots, \varphi_n\}$ over the set of atomic propositions AP of $\mathcal{T}$.

Definition 1 (CTL Property Refinement). *Let φ' and φ be CTL properties. We say φ' refines φ , written as $\varphi' \preceq \varphi$, iff*

- (a) $\llbracket \varphi' \rrbracket \subseteq \llbracket \varphi \rrbracket$, denoted as $\varphi' \preceq \varphi$, or
- (b) $\llbracket \varphi' \rrbracket \subseteq \llbracket \varphi \rrbracket$ and $\llbracket \varphi' \rrbracket \neq \emptyset$, denoted as $\varphi' \preceq^+ \varphi$.

Definition 1(a) captures the traditional notion of refinement: φ' is stronger than φ whenever satisfaction of φ' guarantees satisfaction of φ . In the example of Section 2, φ_1 guarantees eventual coverage of each field zone (liveness but not safety); φ_2 strengthens it by requiring that coverage occur only with sufficient remaining battery (progress and safe operation); and φ_3 additionally requires returning to base before battery depletion. Hence $\varphi_3 \preceq \varphi_2 \preceq \varphi_1$, capturing all their guarantees in a single, stronger specification. Definition 1(b) additionally requires non-emptiness, ensuring that refinement is non-trivial: it excludes contradictions, which would otherwise refine every property via the empty language. This guard is crucial in practice, since only non-trivial refinements impose meaningful constraints that can be substituted for the abstract property without loss of correctness.

This next result will help us establish that $(\text{CTL}/\equiv, \preceq)$, where $\equiv$ means equivalence, forms a partially ordered set. This structure enables organizing properties from abstract (weaker) to more refined (stronger), facilitating refinement and precise management of properties evolution.

Theorem 1. *The CTL refinement relation $\preceq$ is:*

- *Reflexive*: For any CTL property φ , $\varphi \preceq \varphi$.
- *Transitive*: If $\varphi_1 \preceq \varphi_2$ and $\varphi_2 \preceq \varphi_3$, then $\varphi_1 \preceq \varphi_3$.
- *Anti-symmetric*: If $\varphi_1 \preceq \varphi_2$ and $\varphi_2 \preceq \varphi_1$, then $\varphi_1 \equiv \varphi_2$.

Proof. For every state s of $\mathcal{T}$,

- Reflexivity: Trivially, $s \models \varphi$ implies $s \models \varphi$.
- Transitivity: If $\varphi_1 \preceq \varphi_2$ and $\varphi_2 \preceq \varphi_3$, then $s \models \varphi_1$ implies $s \models \varphi_2$, which in turn implies $s \models \varphi_3$.
- Anti-symmetry: If $\varphi_1 \preceq \varphi_2$ and $\varphi_2 \preceq \varphi_1$, then $s \models \varphi_1$ iff $s \models \varphi_2$ for all $\mathcal{T}$ and s , meaning they are equivalent.

Since for any two properties φ_1 and φ_2 their greatest lower bound (meet) is their conjunction ($\varphi_1 \wedge \varphi_2$) and their least upper bound (join) is their disjunction ($\varphi_1 \vee \varphi_2$), this poset is also a lattice, which enables three optimizations of the refinement check. Anti-symmetry lets mutually-refining properties be treated as equivalent, so checks consider only one representative (e.g., an added $\varphi_4 \equiv \varphi_1$ collapses the two). Transitivity lets intermediate properties in a refinement chain be eliminated (having $\varphi_2 \preceq \varphi_1$, testing $\varphi_3 \preceq \varphi_1$ reduces to checking $\varphi_3 \preceq \varphi_2$). Finally, a cycle $\varphi_1 \preceq \dots \preceq \varphi_n \preceq \varphi_1$ makes, via transitivity and anti-symmetry, all its properties logically equivalent, so it collapses to a single canonical representative φ_i whose verification guarantees the rest.

The next result reduces refinement checks to a satisfiability problem.

Theorem 2. *Given two properties φ' and φ , $\varphi' \preceq \varphi \iff \text{UNSAT}(\varphi' \wedge \neg\varphi)$.*

Proof. By Definition 1, $\varphi' \preceq \varphi \iff \llbracket \varphi' \rrbracket \subseteq \llbracket \varphi \rrbracket$. By the semantics of conjunction and negation, $\llbracket \varphi' \wedge \neg\varphi \rrbracket = \llbracket \varphi' \rrbracket \cap \llbracket \varphi \rrbracket^c$. Hence, $\llbracket \varphi' \rrbracket \subseteq \llbracket \varphi \rrbracket \iff \llbracket \varphi' \rrbracket \cap (\llbracket \varphi \rrbracket)^c = \emptyset \iff \llbracket \varphi' \wedge \neg\varphi \rrbracket = \emptyset$, thus $\text{UNSAT}(\varphi' \wedge \neg\varphi)$, which proves the claim.

Corollary 1. *$\varphi' \preceq^+ \varphi$ iff $\text{UNSAT}(\varphi' \wedge \neg\varphi)$ and $\text{SAT}(\varphi')$.*

Proof. The result follows immediately from Theorem 2 and the definition of $\preceq^+$.

5 Refinement Analysis

In order to compute the minimal sufficient set of properties, we propose the procedure outlined in Algorithm 1. The algorithm takes as input a set of CTL properties Φ and returns a minimal refined subset Φ' . The procedure is an extension of the one proposed in [3] for LTL, with modifications to accommodate the branching-time nature of CTL as well as additional optimizations to reduce the number of satisfiability checks required.

The procedure begins by removing unsatisfiable properties from the input set Φ (line 2) by checking the satisfiability of each property. There are two reasons to discard them. First, an unsatisfiable property is a contradiction whose semantics is the empty language, so it would vacuously refine every other property; the non-emptiness guard of Definition 1(b) excludes exactly such trivial refinements,

Algorithm 1 Computing the Minimal Refined Subset

Require: Properties Φ
Ensure: Minimal refined subset Φ'

```

1:  $\Phi' \leftarrow \emptyset$ ,  $ref \leftarrow \emptyset$   $\triangleright$  memoised refinement relation, shared for transitive inference
2:  $(\Phi, \mathcal{CE}) \leftarrow \text{RemoveUnsatisfiableWithCounterexampleModels}(\Phi)$ 
3: for all  $C \in \text{PartitionByAP}(\Phi)$  do
4:    $S \leftarrow \emptyset$ 
5:   for all  $\phi \in C$  do
6:     if  $\exists \rho \in S : \text{Refines}(\rho, \phi)$  then
7:       continue
8:     end if
9:      $S \leftarrow (S \setminus \{\rho \in S : \text{Refines}(\phi, \rho)\}) \cup \{\phi\}$ 
10:  end for
11:   $\Phi' \leftarrow \Phi' \cup S$ 
12: end for
13: return  $\Phi'$ 
14: function  $\text{REFINES}(\phi, \psi)$ 
15:   if  $(\phi, \psi) \in ref$  then
16:     return  $ref[\phi, \psi]$ 
17:   end if
18:    $ref[\phi, \psi] \leftarrow \begin{cases} \text{false}, & \text{NoCommonAP}(\phi, \psi) \vee \mathcal{CE}[\phi] \not\models \psi \\ \text{true}, & \exists \chi : ref[\phi, \chi] \wedge ref[\chi, \psi] \\ \text{UNSAT}(\phi \wedge \neg\psi), & \text{otherwise} \end{cases}$ 
19:   return  $ref[\phi, \psi]$ 
20: end function

```

and keeping them would only introduce spurious representatives. Second, an unsatisfiable property has no model at all, so it cannot participate in the model-based refutation described below. For each *retained* (i.e., satisfiable) property, on the other hand, our tableau-based satisfiability procedure yields, at no extra cost, a concrete model that *satisfies* that property: the open branch that witnesses satisfiability constitutes such a model. These witnessing models are stored in the $\mathcal{CE}$ mapping, which associates each retained (satisfiable) property ϕ with a model of ϕ . As explained below, this satisfying model is later reused as a candidate counterexample to refinement, that is, a state where ϕ holds but a candidate target property ψ may not.

The remaining properties are then partitioned into equivalence classes based on their atomic propositions (line 3), similarly to the approach in [3]. This partitioning is motivated by the observation that properties sharing no atomic propositions cannot refine one another. Conceptually, each class induces a directed graph, and equivalently a partial order, that captures the refinement relation between its properties: every property is a node, and there is an edge from ϕ to ψ exactly when $\phi \preceq^+ \psi$, i.e. when ϕ is at least as strong as ψ . The properties we wish to keep are the *source nodes* of this graph, that is, the strongest properties, those into which no other property refines (no other property in the class is stronger than them). Crucially, the algorithm never materializes this graph nor

enumerates all of its pairwise edges; instead, it computes these source nodes on the fly, as described next.

For each class, we construct on-the-fly the set $\mathcal{S}$ of source nodes, i.e. the strongest properties, those ϕ such that no other property ρ in the class satisfies $\rho \preceq^+ \phi$. This is done by iterating over each property ϕ in the class and checking whether it is refined by some property ρ already in $\mathcal{S}$. If it is (so ρ is at least as strong), ϕ is redundant and we skip it; if not, we add ϕ to $\mathcal{S}$ and remove from $\mathcal{S}$ any property that is refined by ϕ , since such a property is no longer a strongest representative. Comparing each property only against the current set $\mathcal{S}$, rather than against every other property, ensures that $\mathcal{S}$ always contains exactly the strongest representatives of the refinement relation within the class, and avoids materializing every pairwise edge of the conceptual graph.

To check refinement, we call the **Refines** function. The function first checks whether the result of the comparison has already been computed and stored in the refinement relation cache *ref*. If so, the cached result is returned immediately. Otherwise, we perform a series of preliminary checks to reduce the number of expensive satisfiability check calls. First, we prune pairs of properties that do not share any atomic propositions. Since the partitioning step of line 3 is transitive, two properties may be placed in the same class whenever a chain of intermediate properties connects them through shared atomic propositions. As a result, a class may contain two properties that share no atomic propositions directly, linked only through a third. This check ensures that we only perform the refinement check if the two properties share at least one atomic proposition. If they do not, we can immediately conclude that $\phi \not\preceq^+ \psi$. Next, we test the stored model of ϕ against ψ : recall that $\mathcal{CE}[\phi]$ is a model that satisfies ϕ . If this model does not satisfy ψ , then it is a concrete state where ϕ holds but ψ fails, which witnesses $\llbracket \phi \rrbracket \not\subseteq \llbracket \psi \rrbracket$; hence we can soundly conclude that ϕ does not refine ψ without performing a satisfiability check. Finally, we check if there exists an intermediate property χ such that $\phi \preceq^+ \chi$ and $\chi \preceq^+ \psi$. If such a property exists, we can conclude that $\phi \preceq^+ \psi$ by transitivity. Only if all these checks fail do we check $\text{UNSAT}(\phi \wedge \neg\psi)$ to determine if $\phi \preceq^+ \psi$. The result of the refinement check is then stored in the memoisation table *ref* for future reference.

The initial satisfiability checks require $O(|\Phi|)$ calls to the satisfiability procedure, while the refinement phase performs at most $O(|\Phi|^2)$ pairwise comparisons in the worst case. Since each unresolved refinement comparison requires a satisfiability check, the overall worst-case complexity is $O(|\Phi|^2 \cdot T_{\text{SAT}}(\phi \wedge \neg\psi))$, where $T_{\text{SAT}}(\phi)$ denotes the time complexity of the satisfiability procedure for formula ϕ , which is known to be EXPTIME-complete [7].

6 Evaluation

The objective of this section is to evaluate the performance of the refinement analysis procedure described in Section 5. Our evaluation is guided by the following research questions:

Table 1: Benchmark overview.

Metric	Total	MCC			Rers	
		2018	2019	2020	(Acad.)	(Ind.)
Total Properties	52011	15216	16288	19728	180	599
Specification Files	3241	951	1018	1233	9	30
Total Refinements	2973	866	912	1017	16	162
Avg. PRP (%)	5.7	5.7	5.6	5.2	8.9	27.1
Zero-% Benchmarks	2065	604	649	808	3	1
PRP (Non-Zero Cases) (%)	15.6	15.6	15.4	15.0	13.3	28.0
Avg. Formula Length	15.0	14.7	14.6	14.5	8.3	50.3
Avg. Atomic Complexity	14.6	14.3	14.2	14.2	10.2	47.7
Highest PRP (%)	–	100.0	100.0	100.0	35.0	60.0

Table 2: Per-suite performance of the refinement analysis across configurations.

Metric	MCC 2018				MCC 2019				MCC 2020				RERS (Acad.)				RERS (Ind.)			
	Baseline	+MC	(-MC)	Our	Baseline	+MC	(-MC)	Our	Baseline	+MC	(-MC)	Our	Baseline	+MC	(-MC)	Our	Baseline	+MC	(-MC)	Our
Time (s)	0.845	0.695	0.507	0.507	1.335	1.087	0.827	0.812	0.806	0.659	0.460	0.460	0.006	0.310	0.002	0.297	0.078	0.347	0.068	0.344
Mem (MB)	26.0	26.2	25.9	26.2	26.1	26.3	26.0	26.2	25.8	26.0	25.8	26.0	2.5	3.1	2.3	2.9	3.1	3.8	3.0	3.8
ms / pair	8.30	22.00	9.29	30.19	18.54	44.41	21.88	68.17	8.04	21.83	8.97	29.80	0.02	1.02	0.02	3.53	0.21	4.76	0.23	6.72
#checks	130	65	66	33	127	63	63	32	123	60	61	30	354	313	114	99	377	78	274	56

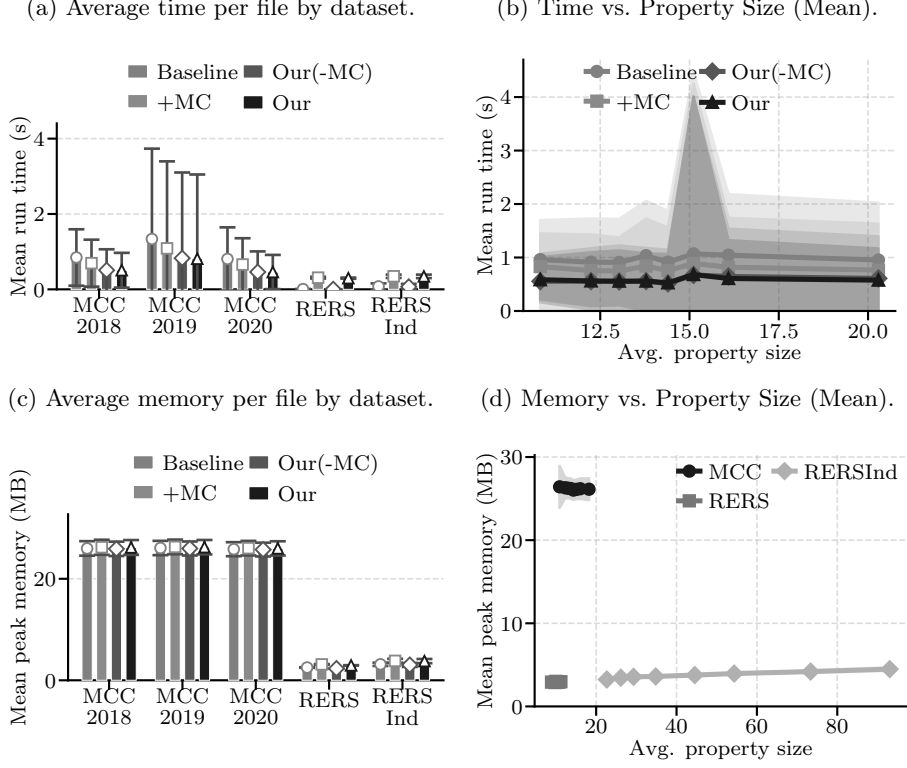
RQ1: What are the quantitative benefits of our refinement analysis in terms of property reduction?

RQ2: What is the computational and memory overhead of our refinement analysis itself and how does it compare to the one proposed in [3]?

We evaluate our approach on a set of benchmarks sourced from both academic and industrial contexts. Specifically, these benchmarks are drawn from the Model Checking Challenges (MCC) of 2018 [13] and 2019 [11] and 2020 [12], as well as the Rigorous Examination of Reactive Systems (RERS) challenge [10], which includes both an academic (Rers) and an industrial (RersInd) suite. These benchmarks consist of a diverse array of models and associated CTL properties, reflecting a variety of real-world scenarios and complexities, varying in domain (e.g., communication protocols, hardware designs, etc.), property size, structure and complexity. This ensures a comprehensive evaluation of our refinement analysis procedure across different contexts and challenges. We measure key metrics including property reduction percentage (PRP), total execution time, memory usage and property size (number of subformulas).

RQ1: To provide an answer to **RQ1**, we analyse the achieved PRP across the different benchmarks. As shown in Table 1, our refinement analysis removes 2973 redundant properties out of 52011 in total, an average PRP of 5.7%. On individual files the reduction reaches 100.0%, i.e. the analysis can eliminate all but one property: in these cases every property in the file is equivalent, so by anti-symmetry of the refinement order a single representative suffices. The effect is highly benchmark-dependent. RersInd exhibits the highest average PRP, 27.1%, and is redundant almost everywhere (only one of its files admits no reduction); the academic RERS suite is intermediate at 8.9%, while the three MCC editions are the lowest at 5.7%, 5.6% and 5.2%, with the majority of MCC files

Fig. 2: Time and Memory Overhead



irreducible. This discrepancy was expected. The MCC properties are authored for a model-checking competition: each formula is designed to probe a distinct behavioural facet of its model, so the suite is curated for diversity, thus logical overlap between properties is rare, as shown by the majority of the files achieving a PRP of 0%. The industrial RERS properties, in contrast, are derived from real-world requirement sets, which accrete over time as families of related constraints, resulting in redundancy. This is exactly what the refinement analysis exploits, yielding both near-universal coverage and the deepest per-file reduction (28.0% on reducible files).

In answer to **RQ1**: our refinement analysis yields a substantial but benchmark-dependent reduction in the number of properties that must be verified. Overall it identifies 2973 of 52011 properties as redundant—an average PRP of 5.7%—and up to 100.0% within a single file. The benefit is largest precisely where specifications are least curated: it reaches 27.1% on the industrial RERS suite (28.0% among its reducible files), eliminating more than a quarter of the verification burden, while remaining modest (5.2–5.7%) on the diversity-curated MCC competition suites.

RQ2: To answer **RQ2**, we performed an ablation study to quantify the contribution of the two main extensions to the analysis we propose: the antichain heuristics and the model-checking prune. The baseline configuration, i.e., the analysis from [3], is compared against three configurations: one where model checking is introduced to the baseline (+MC), one with only the antichain heuristics (Our(-MC)) and one with both the antichain heuristics and the model-checking prune (Our), which is our full approach. We measured wall-clock time and peak memory per file during analysis, reporting the results in Table 2 and Figure 2. Even on the slowest performing suite, MCC 2019, the analysis takes only 0.812 s per file on average, and the peak memory is 26.2 MB.

The two ablations have complementary effects. In the MCC suites, both extensions, when singularly applied, roughly halve the number of refinement checks, thus reducing the total time to below the baseline. When applied together, their reductions compound, bringing the number of checks down to a quarter of the baseline, since each prune removes roughly half of the checks the other leaves. On the RERS suites, the two reductions compound as well, cutting the checks to roughly a quarter of the baseline on the academic suite and to about a sixth on the industrial suite. On the RERS suites, whose properties are simple propositional AG-safety formulas that the tableau decides almost instantly, the model-check invocation is not amortised and adds a small *relative* overhead (0.006 s $\rightarrow$ 0.297 s on the academic suite), though the absolute cost remains negligible.

The seemingly opposite behaviour of the two extensions on the RERS suites is not a contradiction but a signature of their complementarity. The antichain heuristic is *syntactic*: it discards a candidate pair as soon as the two properties share no atomic proposition, since one then cannot refine the other. The model-check prune is *semantic*: it discards a pair by exhibiting a single counterexample state, and is indifferent to atomic overlap. Because the two levers key on unrelated features, a suite can be highly reducible by one while remaining almost opaque to the other—and the two RERS families sit at opposite ends of this spectrum. The academic suite consists of small temporal properties (on average 3.3 atomic propositions each), so roughly two-thirds of all property pairs are atom-disjoint; the antichain heuristic alone therefore eliminates 68% of the refinement checks—almost exactly the fraction of disjoint pairs—whereas the model-check, confronted with genuinely temporal formulas whose refutation demands a path witness, removes only 12%. The industrial suite is the mirror image: its properties are large AG-guarded propositional conjunctions (12 atomic propositions each) drawn from a single shared vocabulary, so 95% of pairs share an atom and the syntactic prune is starved; but because refuting a purely propositional refinement reduces to finding a single satisfying state, the model-check discharges 79% of the checks. The two optimisations thus target complementary forms of redundancy—syntactic disjointness and semantic refutability—which is why they reinforce one another on the arithmetic MCC suites, where both are abundant, while on each RERS suite the reduction is carried almost entirely by the single prune to which that suite is susceptible.

Notably, the overhead of the analysis is not driven by the size of the properties, but by their structure. As shown in Figures 2b and 2d, the time and memory per file are not primarily correlated with the average property size, although memory slightly rises for RersInd (c.f. Figure 2d). On the contrary, as shown in Figures 2c and reported in Table 2, the arithmetic-heavy MCC properties, which invoke the SMT back-end, consume ≈ 26.2 MB regardless of configuration, whereas the purely propositional RERS properties use only ≈ 3.8 MB. This suggests that the overhead is dominated by the cost of the satisfiability checks, which is driven by the presence of arithmetic in the properties, rather than by their size or number.

In answer to **RQ2**: the refinement analysis is inexpensive in absolute terms—at most 0.812s and 26.2 MB per file, on the slowest suite. Relative to the procedure of [3] (our baseline), the two extensions reduce the number of refinement checks to a quarter on the MCC suites, so that our full approach is faster than the baseline despite the added model-checking cost; on the trivial RERS suites they add only a sub-second, unamortised overhead. Peak memory is essentially unchanged by the extensions and is governed instead by whether the properties invoke arithmetic (≈ 26.2 MB with, ≈ 3.8 MB without), confirming that the cost is dominated by the satisfiability checks rather than by property size or count.

7 Summary and Outlook

We presented a complete theory of property refinement for CTL. We reduced refinement checking to satisfiability checking, and we exploited its pre-order properties to derive a complete procedure that computes the minimal sufficient set. On our evaluation, our approach proves both effective (up to 27% average reduction) and efficient (consistently lower 70 ms per check), reducing the number of expensive sat checks by up to a factor of six compared to prior work.

Several directions remain open. The satisfiability-based characterization of refinement extends naturally to richer temporal logics such as CTL*. The algorithm offers parallelization opportunities, allowing for distributed computation and further scalability. Finally, integrating refinement analysis into established model checking tools would let engineers exploit refinement directly within their existing verification workflows. More broadly, this work shifts perspective from verification as the isolated checking of properties to specifications as structured, interrelated artifacts, grounding verification not on exhaustive checking but on a precise understanding of their structure

8 Acknowledgements

Supported by the Ministry of Economic Affairs, Industry, Climate Action and Energy of the State of North Rhine-Westphalia. Co-funded by the European Union. Grant number EFRE-20800498.

References

1. Baier, C., Katoen, J.P.: Principles of Model Checking. MIT Press (2008)
2. Barnat, J., Bauch, P., Beneš, N., Brim, L., Beran, J., Kratochvíla, T.: Analysing sanity of requirements for avionics systems. *Formal Aspects of Computing* **28**(1), 45–63 (Mar 2016). <https://doi.org/10.1007/s00165-015-0348-9>, <https://doi.org/10.1007/s00165-015-0348-9>
3. Brodo, L., Scalora, G., Henkler, S.: Property refinement in linear temporal logic: Formal semantics and algorithms for software verification. In: *Proceedings of the ACM International Conference on the Foundations of Software Engineering. FSE '26*, ACM (July 2026)
4. Brodo, L., Scalora, G., Henkler, S.: What do we really need? identifying minimal sufficient temporal properties via refinement. In: *Proceedings of the IEEE/ACM 48th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion '26)*. pp. 1–2. ACM, Rio de Janeiro, Brazil (2026). <https://doi.org/10.1145/3774748.3795661>
5. Cimatti, A., Clarke, E.M., Giunchiglia, F., Roveri, M.: NuSMV: A new symbolic model verifier. In: Halbwachs, N., Peled, D.A. (eds.) *Computer Aided Verification, 11th International Conference, CAV '99. Lecture Notes in Computer Science*, vol. 1633, pp. 495–499. Springer (1999). https://doi.org/10.1007/3-540-48683-6_44
6. Clarke, E.M., Emerson, E.A.: Design and synthesis of synchronization skeletons using branching-time temporal logic. In: Kozen, D. (ed.) *Logic of Programs. Lecture Notes in Computer Science*, vol. 131, pp. 52–71. Springer (1982). <https://doi.org/10.1007/BFb0025774>
7. Clarke, E.M., Grumberg, O., Peled, D.A.: *Model Checking*. MIT Press (1999)
8. Dureja, R., Rozier, K.Y.: More scalable LTL model checking via discovering design-space dependencies (d^3). In: *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. *Lecture Notes in Computer Science*, vol. 10805, pp. 309–327. Springer (2018). https://doi.org/10.1007/978-3-319-89960-2_17
9. Emerson, E.A., Halpern, J.Y.: Decision procedures and expressiveness in the temporal logic of branching time. *Journal of Computer and System Sciences* **30**(1), 1–24 (1985). [https://doi.org/10.1016/0022-0000\(85\)90001-7](https://doi.org/10.1016/0022-0000(85)90001-7)
10. Jasper, M., Mues, M., Murtovi, A., Schlüter, M., Howar, F., Steffen, B., Schordan, M., Hendriks, D., Schiffelers, R., Kuppens, H., Vaandrager, F.W.: Rers 2019: Combining synthesis with real-world models. In: *Tools and Algorithms for the Construction and Analysis of Systems. Lecture Notes in Computer Science*, vol. 11429, pp. 101–115. Springer (2019). https://doi.org/10.1007/978-3-030-17502-3_7
11. Kordon, F., Garavel, H., Hillah, L.M., Hulin-Hubard, F., Amparore, E., Beccuti, M., Berthomieu, B., Ciardo, G., Zilio, S.D., Liebke, T., Li, S., Meijer, J., Miner, A., Srba, J., Thierry-Mieg, Y., van de Pol, J., van Dirk, T., Wolf, K.: Complete results for the 2019 edition of the model checking contest. <http://mcc.lip6.fr/2019/results.php> (April 2019)
12. Kordon, F., Garavel, H., Hillah, L.M., Hulin-Hubard, F., Amparore, E., Beccuti, M., Berthomieu, B., Ciardo, G., Zilio, S.D., Liebke, T., Li, S., Meijer, J., Miner, A., Srba, J., Thierry-Mieg, Y., van de Pol, J., van Dirk, T., Wolf, K.: Complete results for the 2020 edition of the model checking contest. <http://mcc.lip6.fr/2020/results.php> (April 2020)
13. Kordon, F., Garavel, H., Hillah, L.M., Hulin-Hubard, F., Amparore, E., Beccuti, M., Berthomieu, B., Ciardo, G., Zilio, S.D., Liebke, T., Linard, A., Meijer, J.,

- Miner, A., Srba, J., Thierry-Mieg, Y., van de Pol, J., Wolf, K.: Complete results for the 2018 edition of the model checking contest. <http://mcc.lip6.fr/2018/results.php> (June 2018)
14. Méry, D., Roegel, D.: Refining formal specifications to get efficient, structured and correct concurrent programs. Technical report, Centre de Recherche en Informatique de Nancy, CNRS / Université de Nancy, Nancy, France (1994)
 15. Mokkedem, A., Méry, D.: On using temporal logic for refinement and compositional verification of concurrent systems. *Theoretical Computer Science* **140**(1), 95–138 (1995). [https://doi.org/10.1016/0304-3975\(94\)00206-X](https://doi.org/10.1016/0304-3975(94)00206-X)
 16. Pnueli, A.: System specification and refinement in temporal logic. In: Shyam-sundar, R.K. (ed.) *Foundations of Software Technology and Theoretical Computer Science*, 12th Conference, New Delhi, India, December 18–20, 1992, Proceedings. *Lecture Notes in Computer Science*, vol. 652, pp. 1–38. Springer (1992). https://doi.org/10.1007/3-540-56287-7_92
 17. Prezza, N.: CTLSAT: A CTL satisfiability solver. <https://github.com/nicolaprezza/CTLSAT> (2015), accessed on 2025-07-16
 18. Reynolds, M.: A tableau-based decision procedure for CTL*. *Formal Aspects of Computing* **23**(6), 739–779 (2011). <https://doi.org/10.1007/s00165-011-0193-4>
 19. Schneider, S., Wollersheim, J., Krcmar, H., Sunyaev, A.: How do requirements evolve over time? a case study investigating the role of context and experiences in the evolution of enterprise software requirements. *Journal of Information Technology* **33**(2), 151–170 (2018). <https://doi.org/10.1057/s41265-016-0001-y>