Property Refinement in Linear Temporal Logic: Formal Semantics and Algorithms for Software Verification

LUCA BRODO, Hochschule Hamm-Lippstadt, Germany
GIUSEPPE SCALORA, Hochschule Hamm-Lippstadt, Germany
STEFAN HENKLER, Hochschule Hamm-Lippstadt, Germany

Model checking using temporal logic is a key aspect of formal verification of modern complex software systems. These systems are often result of distributed development processes, involving multiple teams and iterative design cycles. This complexity is mirrored in the corresponding formal specifications, which often consist of a large number of temporal logic properties that need to be verified against a system model. Many of these properties overlap semantically, yet traditional verification treats them as independent, resulting in redundant checks that waste computational resources and inflate engineering effort

We address this by introducing and operationalizing a formal theory of *property refinement* for temporal logic into a concrete methodology that automatically identifies redundant properties from a verification suite. This is achieved by first partitioning specifications into equivalence classes based on shared atomic propositions, followed by an analysis of intra-class refinement relations to construct the minimal sufficient subset. Our extensive empirical evaluation confirms the practical viability of our approach, demonstrating it can reduce the number of required verification tasks by up to 75% and accelerate model checking by up to three orders of magnitude, with a one-time associated overhead cost of 0.035% for the refinement analysis. The results of the evaluation confirm that the benefits of this approach are threefold: it reduces the computational requirements for formal verification; it allows engineers to focus on the core requirements of the system, hereby reducing engineering effort; finally, the one-time negligible investment in refinement yields compounding returns, making it particularly advantageous for agile and long-term development lifecycles.

This work thus establishes property refinement analysis as a key technique for scaling requirement engineering and software verification to modern complex software systems

CCS Concepts: • **Theory of computation** → **Logic and verification**; • **Software and its engineering** → **Model checking**.

Additional Key Words and Phrases: Formal Verification, LTL Property Refinement

ACM Reference Format:
Luca Brodo, Giuseppe Scalora, and Stefan Henkler. 2026. Property Refinement in Linear Temporal Logic: Formal Semantics and Algorithms for Software Verification. *Proc. ACM Softw. Eng.* 3, FSE, Article FSE120 (July 2026), 22 pages. https://doi.org/10.1145/3808127

1 INTRODUCTION

The design of modern software systems relies on formal methods to guarantee system correctness against a suite of behavioral properties. During the design process, a behavioral model of the system is developed and formal verification techniques, such as model checking [4, 10], are employed to verify this model against a set of formal specifications that encode desired system behaviors. These specifications are typically expressed in Linear Temporal Logic (LTL), a logic adept at capturing

Authors' addresses: Luca Brodo, luca.brodo@hshl.de, Hochschule Hamm-Lippstadt, Lippstadt, NRW, Germany; Giuseppe Scalora, giuseppe.scalora@hshl.de, Hochschule Hamm-Lippstadt, Lippstadt, NRW, Germany; Stefan Henkler, stefan.henkler@hshl.de, Hochschule Hamm-Lippstadt, Lippstadt, NRW, Germany.

Please use nonacm option or ACM Engage class to enable CC licenses
This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2026 Copyright held by the owner/author(s).
ACM 2994-970X/2026/7-ARTFSE120
https://doi.org/10.1145/3808127

the dynamic behavior, concurrency aspects, and interaction protocols inherent in modern software systems.

As the design of software systems evolves through iterative development, feature expansion, and architectural changes, often across distributed teams, both the system models and the specification suites must adapt. Consequently, these artifacts are continuously subject to revision. This evolution increases the computational and engineering demands of formal verification, as larger and more complex models and specifications require more resources to be verified and maintained. To date, the community has primarily attacked this on the model side through techniques such as abstraction and compositional reasoning [4, 22]. Yet an equally important but underexplored issue rooted in the design engineering process itself lies on the specification side.

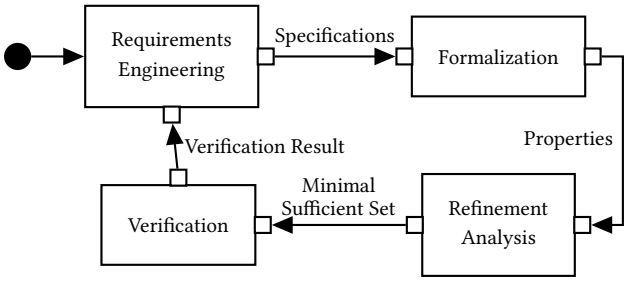


Fig. 1. The proposed enriched verification workflow, featuring an intermediate refinement analysis to compute a minimal sufficient set of properties prior to verification.

According to the traditional verification workflow in software engineering, after the specifications are elicited and are formalized into properties (the *Requirements Engineering* and *Formalization* phases in Figure 1), the latter are treated as lists and checked individually in the *Verification* phase. It is implicitly assumed that each property is independent and provides unique assurance, thereby overlooking any semantic relationships. In practice, however, specifications are not disjoint and static artifacts, but they evolve over design iterations. As such, parts of the specification are often inherited from legacy designs [26] and accumulate over years [41], frequently exhibiting intricate logical, structural, or semantic relationships reflecting the interconnectedness of software components and functionalities. At the *Requirements Engineering* phase, this complexity leads to artifacts rife with redundancy and dependencies [2, 11, 21, 33, 39]. These phenomena propagate into the *Formalization* phase, where requirements are translated into LTL properties. Reported experience from [5] confirms that LTL specifications in real-world avionics systems exhibit redundancy and non-trivial semantic dependencies. Such phenomena may lead to vacuous satisfaction, hidden inconsistencies, and inflated verification effort during the *Verification* phase. We refer to this progressive accumulation of redundant and interdependent properties as specification debt.

Structurally, a high-level property concerning an entire software module might be decomposed into, and thus refined by, several properties governing its constituent sub-modules or internal state transitions. Consider, for example, following specifications often found in mutual exclusion protocols, where a process P_0 can be in a Non-Critical Section (NCS), requesting access, or in its Critical Section (CS). The following three LTL properties are taken from a mutual exclusion protocol specification, where "$P_i = CS$" denotes that process i is in a critical section:

(1) $\varphi_1 : \neg(\ (GF(P_0 = CS) \wedge GF(P_1 = CS)) \rightarrow \neg(GF(P_0 = NCS))\ )$. This property allows P_0 to sometimes be in a non-critical state, even if both P_0 and P_1 eventually enter the critical section repeatedly. In other words, being in the critical section repeatedly does not prevent P_0 from leaving it.

(2) $\varphi_2 : \neg(\, G((P_0 = p1 \vee P_0 = p2 \vee P_0 = p3) \rightarrow F(P_0 = CS)) \rightarrow \neg(GF(P_0 = CS))\,)$, which expresses that P_0 will always eventually reach the critical section, regardless of which intermediate state (p1, p2, p3) it passes through. Essentially, this ensures that P_0's normal sequence of states doesn't block access to the CS.
(3) $\varphi_3 : G(\neg(P_0 = CS) \rightarrow F(P_0 = CS))$, which ensures that whenever P_0 is not in the critical section, it is guaranteed to eventually enter it. This is a liveness property, ensuring that P_0 cannot starve or stay out of the CS indefinitely.

The property φ_1 emerges as the essential specification, leaving the other two as redundant. This situation can naturally arise during iterative development, where new requirements are introduced to strengthen or clarify existing ones—such as φ_2, which may have been previously specified. Semantically, the original properties can be replaced by this single, more precise property, as it fully captures their constraints. Consequently, the satisfaction of a carefully selected, stronger subset of refined properties would be sufficient to rigorously demonstrate the system's overall correctness against its original specification while reducing the verification workload. Furthermore, by tracing these semantic relations back to the original requirements, engineers can identify where redundancies originate, how requirements have implicitly evolved over time and which requirements are the most critical to the system's correctness. This provides qualitative insight into the structure and health of the requirements themselves—revealing overlaps that stem from poor elicitation practices, inconsistencies introduced across design iterations, or unnecessary strengthening added by independent teams.

However, due to the complexity of LTL specifications and the potential presence of thousands of properties in a system, identifying such redundancies remains challenging. Furthermore, a formal, suite-level, and correctness-preserving framework for understanding what "refined" means in the context of temporal logic properties is currently lacking. While the field has seen extensive research in model refinement [8, 15, 20] –allowing engineers to rigorously transition from abstract to concrete implementations while preserving critical properties as well as simplify models through quotienting– the analogous concept of property refinement at the property level has remained significantly underdeveloped. While semantic implication, subsumption, vacuity detection, and refinement notions for temporal properties have been studied extensively, these techniques typically focus on reasoning about individual formulas, model-dependent analysis, or intra-formula simplification.

This paper addresses this critical gap by introducing a formal, semantic definition of *property refinement* for LTL properties. We define what it means for one property to refine another based on language inclusion. By establishing the algebraic properties of this refinement relation (e.g., reflexivity, transitivity, composability), we prove that this refinement relation forms a compositional preorder, enabling modular reasoning about specifications' strength. This is then leveraged to develop a technique to identify a minimal, sufficient and most-refined subset of properties whose verification guarantees the satisfaction of the entire original specification set.

Through the identification of these refinement relationships, we propose enriching the workflow with the additional *Refinement Analysis* step (c.f. Figure 1). This integrated approach aims to provide a more systematic way of understanding and managing the relationships between different properties instead of simply verifying all formalized properties in isolation. The objective is to leverage refinement to reduce verification effort without compromising assurance and enhancing the manageability of specification suites. This promises to significantly alleviate the verification burden, leading to more efficient verification and engineering workflows without sacrificing rigor.

Our work, therefore, aims to elevate property refinement to the same level of formal understanding as model refinement and leverage it for efficient verification, compositional reasoning, and better management of complex specification suites.

The remainder of this paper is structured as follows. Section 2 provides essential background on LTL. In Section 3, we formally define LTL property refinement, establish its key structural properties such as reflexivity, transitivity, anti-symmetry, and composability, and discuss its automata-theoretic characterization. Section 4 details our approach for leveraging property refinement to achieve more efficient model checking. Section 5 presents an empirical evaluation of our approach, demonstrating its effectiveness in reducing the number of properties requiring verification with modest computation overhead. Section 6 reviews existing literature on model and specification refinement, highlighting the gap our work addresses. Finally, Section 7 concludes the paper and outlines potential avenues for future research.

2 PRELIMINARIES

Linear Temporal Logic (LTL) was first introduced by Pnueli in his seminal work [36], where he proposed temporal logic as a formalism for specifying properties of concurrent systems. Since then, LTL has become a cornerstone in the field of formal methods, particularly in model checking and formal verification. We base the following section on [4], but make the common alphabet explicit to compare formulas over different atomic proposition sets.

We write $\mathcal{L}_A(\varphi)$ for the *trace language* of an LTL property φ over alphabet A, i.e., the set of infinite words $\sigma \in (2^A)^\omega$ that satisfy φ. When comparing a set of formulas Φ, we interpret all languages over the common alphabet $A := \bigcup_{\psi\in\Phi} AP(\psi)$(unused atoms are ignored by satisfaction) where $AP(\psi)$ denotes the set of atomic propositions occurring in ψ. The semantics of LTL over words is defined as follows.

Definition 2.1 (LTL semantics over words). Let A be a finite set of atomic propositions. A (infinite) word over A is a sequence $\sigma = A_0A_1A_2\cdots$ with each $A_i \subseteq A$. We write $\sigma, i \models \varphi$ for satisfaction at position $i \in \mathbb{N}$, defined by:

$$
\begin{aligned}
&\sigma, i \models \top \\
&\sigma, i \models a \iff a \in A_i \quad (a \in A) \\
&\sigma, i \models \neg\varphi \iff \sigma, i \not\models \varphi \\
&\sigma, i \models \varphi \wedge \psi \iff \sigma, i \models \varphi \text{ and } \sigma, i \models \psi \\
&\sigma, i \models \mathbf{X}\varphi \iff \sigma, i{+}1 \models \varphi \\
&\sigma, i \models \varphi\,\mathbf{U}\,\psi \iff \exists j \geq i.\ \sigma, j \models \psi \ \wedge\ \forall k\,(i \leq k < j \Rightarrow \sigma, k \models \varphi).
\end{aligned}
$$

Derived operators: $\mathbf{F}\varphi \equiv \top\,\mathbf{U}\,\varphi$, $\mathbf{G}\varphi \equiv \neg\mathbf{F}\neg\varphi$. We abbreviate $\sigma \models \varphi$ for $\sigma, 0 \models \varphi$ and define the language $\mathcal{L}_A(\varphi) := \{\sigma \in (2^A)^\omega \mid \sigma \models \varphi\}$.

The concept of transition systems as the underlying semantic model for reactive systems is standard in formal verification. Transition systems are formally defined in foundational texts like [4, 10], and are widely used to represent the behavior of systems under verification.

Definition 2.2 (Transition system). A transition system is a tuple $\mathcal{T} = (S, Act, \rightarrow, I, A, L)$ where S is a set of states, Act a set of actions, $\rightarrow \subseteq S \times Act \times S$, $I \subseteq S$ the initial states, A a set of atomic propositions, and $L : S \to 2^A$ a labeling. An (infinite) path is a sequence $\pi = s_0s_1s_2\cdots$ with $s_0 \in I$ and $\exists a_i \in Act.\ (s_i, a_i, s_{i+1}) \in \rightarrow$ for all i. Its *trace* is $\mathrm{trace}(\pi) := L(s_0)L(s_1)L(s_2)\cdots \in (2^A)^\omega$. We write $\mathrm{Traces}_A(\mathcal{T}) := \{\mathrm{trace}(\pi) \mid \pi \text{ path of } \mathcal{T}\}$.

The definition of satisfaction of an LTL formula by a transition system, $\mathcal{T} \models \varphi$, as trace inclusion $\mathrm{Traces}_A(\mathcal{T}) \subseteq \mathcal{L}_A(\varphi)$, aligns with the semantics presented in [4, 10].

Proposition 2.3 (TS vs. word semantics). *For any transition system $\mathcal{T}$ over alphabet A and any LTL formula φ,*

$$\mathcal{T} \models \varphi \quad \Longleftrightarrow \quad \mathit{Traces}_A(\mathcal{T}) \subseteq \mathcal{L}_A(\varphi) \quad \Longleftrightarrow \quad \forall \pi \; of \mathcal{T},\ \mathrm{trace}(\pi) \models \varphi.$$

A *partially ordered set* (poset) $(L, \sqsubseteq)$ consists of a set L and a binary relation $\sqsubseteq$ that is reflexive, transitive, and anti-symmetric. Elements of L are said to be ordered under $\sqsubseteq$. For a subset $B \subseteq L$, an element $y \in L$ is an *upper bound* of B if $s \sqsubseteq y$ for all $s \in B$. It is a *least upper bound* (lub) if $y \sqsubseteq y'$ for every upper bound y' of B. *Lower bounds* and the *greatest lower bound* (glb) are defined dually. A *lattice* is a poset in which every non-empty finite subset has both a least upper bound and a greatest lower bound. A *complete lattice* is one in which every subset (finite or infinite) has both a least upper bound and a greatest lower bound. All finite lattices are complete.

3 LTL PROPERTY REFINEMENT AND ITS STRUCTURAL PROPERTIES

In this section, we formalize a notion of refinement between LTL properties, which captures when one property strengthens or constrains another. This concept is foundational in compositional reasoning, system specification, and abstraction refinement. The refinement relation allows us to understand how a more concrete or detailed property (e.g., arising from an implementation or a strengthened specification) 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 language of infinite words that satisfy an LTL property, enabling a precise, model-theoretic interpretation. Following this, we establish fundamental structural properties of this refinement relation, such as reflexivity, transitivity and anti-simmetry. 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 fixed set of properties $\Phi = \{\varphi_1, \ldots, \varphi_n\}$ over a common set of atomic propositions $A = \bigcup_{i=1}^{n} AP(\varphi_i)$.

Definition 3.1 (LTL Property Refinement). Let φ' and φ be LTL properties. We say φ' refines φ, iff

(a) $\mathcal{L}_A(\varphi') \subseteq \mathcal{L}_A(\varphi)$ denoted as $\varphi' \preceq \varphi$, or
(b) $\mathcal{L}_A(\varphi') \subseteq \mathcal{L}_A(\varphi) \quad$ and $\quad \mathcal{L}_A(\varphi') \neq \emptyset$ denoted as $\varphi' \preceq^{+} \varphi$

Definition 3.1(a) captures the traditional notion of refinement: φ' is stronger than φ whenever satisfaction of φ' guarantees satisfaction of φ. Definition 3.1(b) additionally requires non-emptiness, ensuring that refinement is non-trivial. This excludes contradictions such as $p \wedge \neg p$, which would otherwise refine every property via the empty language. The non-emptiness guard is crucial in practice (cf. Section 4), since only non-trivial refinements impose meaningful constraints that can be substituted for the abstract property without loss of correctness

While simple examples of refinement, such as $Gp \preceq Fp$, are intuitive, the relationships within complex specification suites derived from industrial practice are often far from obvious. Consider again the example in Section 1 with the three properties $\varphi_1, \varphi_2, \varphi_3$. In order to establish the refinement relationships between these properties, we analyze their languages and the behaviors they enforce on the processes. Formula φ_1, which can be rewritten as $GF(P_0 = \mathrm{CS}) \wedge GF(P_1 = \mathrm{CS}) \wedge GF(P_0 = \mathrm{NCS})$, requires that both processes P_0 and P_1 enter the critical section infinitely often, while P_0 also spends an infinite amount of time outside the critical section. Formula φ_2, which simplifies to $GF(P_0 = \mathrm{CS})$, requires only that P_0 enters the critical section infinitely often, without imposing any constraints on P_1 or on P_0 being outside the critical section. Similarly, φ_3, which states $G(\neg(P_0 = \mathrm{CS}) \rightarrow F(P_0 = \mathrm{CS}))$, ensures that whenever P_0 is outside the critical section, it eventually returns, repeating indefinitely. Consequently, any run satisfying φ_1 also satisfies both φ_2 and φ_3,

whereas the converse does not hold φ_2 and φ_3 do not impose the additional constraints required by φ_1. In other words, φ_1 refines the other two properties.

We now analyze the algebraic structure induced by the refinement relation $\preceq$ over LTL properties. These structural insights are foundational for reasoning about specification hierarchies, property transformations, and correctness preservation.

Proposition 3.2 (Algebraic Properties of $\preceq$). *The refinement relation $\preceq$ is a pre-order (reflexive and transitive) and is anti-symmetric modulo logical equivalence:*

- ***Reflexive:*** *For any LTL 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 (modulo $\equiv$):*** *$\varphi_1 \preceq \varphi_2$ and $\varphi_2 \preceq \varphi_1 \iff \varphi_1 \equiv \varphi_2$.*

Proof. **Reflexive:** For any φ, $\mathcal{L}_A(\varphi) \subseteq \mathcal{L}_A(\varphi)$ holds trivially. **Transitive:** Assume $\varphi_1 \preceq \varphi_2$ and $\varphi_2 \preceq \varphi_3$. By definition, $\mathcal{L}_A(\varphi_1) \subseteq \mathcal{L}_A(\varphi_2)$ and $\mathcal{L}_A(\varphi_2) \subseteq \mathcal{L}_A(\varphi_3)$, implying $\mathcal{L}_A(\varphi_1) \subseteq \mathcal{L}_A(\varphi_3)$. **Anti-symmetric:** Given $\varphi_1 \preceq \varphi_2$ and $\varphi_2 \preceq \varphi_1$, then by Definition 3.1, $\mathcal{L}_A(\varphi_1) \subseteq \mathcal{L}_A(\varphi_2)$ and $\mathcal{L}_A(\varphi_2) \subseteq \mathcal{L}_A(\varphi_1)$. This implies $\mathcal{L}_A(\varphi_1) = \mathcal{L}_A(\varphi_2)$, meaning $\varphi_1 \equiv \varphi_2$. Conversely, if $\varphi_1 \equiv \varphi_2$, then $\mathcal{L}_A(\varphi_1) = \mathcal{L}_A(\varphi_2)$, and both refinement directions hold by Definition 3.1. □

This result establishes that (LTL/$\equiv$, $\preceq$) forms a partially ordered set (poset). Furthermore, since for any two properties φ_1 and φ_2 their greatest lower bound (meet) is given by their conjunction ($\varphi_1 \wedge \varphi_2$) and their least upper bound (join) is given by their disjunction ($\varphi_1 \vee \varphi_2$), this poset is also a lattice. This structure enables organizing specifications from abstract (weaker) to concrete (stronger), facilitating stepwise refinement and precise management of specification evolution. Notably, these characteristics also allow for optimizations of the refinement check processes. Firstly, anti-symmetry ensures that if two properties are found to refine each other, they can be treated as equivalent, avoiding redundant checks. In the example from Section 1, since φ_2 and φ_3 are equivalent, any refinement checks involving either can be simplified by considering just one of them. Secondly, transitivity allows for the elimination of intermediate properties in a chain of refinements, reducing the number of direct checks needed. Considering again the example in Section 1, let's assume an additional property φ_4 is introduced. If we already know that $\varphi_1 \preceq \varphi_2$ and $\varphi_2 \preceq \varphi_4$, then by transitivity, we can conclude that $\varphi_1 \preceq \varphi_4$ without needing to check this directly.

The properties of the refinement relation also provide a clear strategy for handling *cycles* in the lattice. A cycle occurs if there exists a subset of properties $\{\varphi_1, \ldots, \varphi_n\}$ such that $\varphi_1 \preceq \varphi_2 \preceq \ldots \preceq \varphi_n \preceq \varphi_1$. By the transitivity of $\preceq$, we can deduce that $\varphi_1 \preceq \varphi_n$. Combined with the final link in the cycle, $\varphi_n \preceq \varphi_1$, the anti-symmetry property implies that $\varphi_1 \equiv \varphi_n$. This reasoning can be applied to any pair of properties within the cycle, proving that all properties are logically equivalent. Consequently, for any analysis, the entire cycle can be collapsed into a single conceptual node. We can arbitrarily select any property φ_i from the cycle as its canonical representative, since verifying any one of them is sufficient to guarantee the satisfaction of all others in the cycle.

Another important aspect of refinement is its *composability* with logical connectives, essential for modular reasoning in system design where specifications are constructed compositionally.

Proposition 3.3 (Composability with Conjunction). *Let $\varphi_1, \varphi_2, \varphi_3$ be LTL properties. If $\varphi_1 \preceq \varphi_2$, then $\varphi_1 \wedge \varphi_3 \preceq \varphi_2 \wedge \varphi_3$ as well as $\varphi_1 \wedge \varphi_3 \preceq \varphi_2$.*

Proof. By the semantics of conjunction:

$$\mathcal{L}_A(\varphi_1 \wedge \varphi_3) = \mathcal{L}_A(\varphi_1) \cap \mathcal{L}_A(\varphi_3), \quad \mathcal{L}_A(\varphi_2 \wedge \varphi_3) = \mathcal{L}_A(\varphi_2) \cap \mathcal{L}_A(\varphi_3).$$

Since $\mathcal{L}_A(\varphi_1) \subseteq \mathcal{L}_A(\varphi_2)$, intersecting both sides with $\mathcal{L}_A(\varphi_3)$ preserves inclusion:

$$\mathcal{L}_A(\varphi_1) \cap \mathcal{L}_A(\varphi_3) \subseteq \mathcal{L}_A(\varphi_2) \cap \mathcal{L}_A(\varphi_3)$$
$$\iff \mathcal{L}_A(\varphi_1 \wedge \varphi_3) \subseteq \mathcal{L}_A(\varphi_2 \wedge \varphi_3),$$

which by Definition 3.1 implies $\varphi_1 \wedge \varphi_3 \preceq \varphi_2 \wedge \varphi_3$. Since $\mathcal{L}_A(\varphi_2 \wedge \varphi_3) \subseteq \mathcal{L}_A(\varphi_2)$, we also have $\varphi_1 \wedge \varphi_3 \preceq \varphi_2$. □

Proposition 3.3 is especially important for the evolution of property suites. As properties are refined or strengthened, ensuring that these refinements compose correctly with existing specifications is crucial for maintaining overall system correctness. For instance, if a property φ_1 is proven to refine φ_2 and it is further strengthened by the conjunction with another property φ_3 as additional knowledge about the system is obtained, this still remains a refinement of the conjunction of φ_2 (or an accordingly strengthened version of it). For example, let $\varphi_1 = \mathbf{G}p$, $\varphi_2 = \mathbf{G}(p \vee q)$, and $\varphi_3 = \mathbf{F}r$. Clearly, $\varphi_1 \preceq \varphi_2$, and conjoining φ_3 yields $\varphi_1 \wedge \varphi_3 = \mathbf{G}p \wedge \mathbf{F}r$ and $\varphi_2 \wedge \varphi_3 = \mathbf{G}(p \vee q) \wedge \mathbf{F}r$, which preserves the refinement relationship. Thus, composability under conjunction allows modular verification and incremental strengthening of property suites without impacting relation correctness.

Proposition 3.4 (Composability with Disjunction). *Let $\varphi_1, \varphi_2, \varphi_3$ be LTL properties. If $\varphi_1 \preceq \varphi_2$, then $\varphi_1 \vee \varphi_3 \preceq \varphi_2 \vee \varphi_3$ as well as $\varphi_1 \preceq \varphi_2 \vee \varphi_3$*

Proof. By assumption, $\varphi_1 \preceq \varphi_2$, so $\mathcal{L}_A(\varphi_1) \subseteq \mathcal{L}_A(\varphi_2)$. Since $\mathcal{L}_A(\varphi_2) \subseteq \mathcal{L}_A(\varphi_2 \vee \varphi_3)$, we have $\varphi_1 \preceq \varphi_2 \vee \varphi_3$.

Now,

$$\mathcal{L}_A(\varphi_1 \vee \varphi_3) = \mathcal{L}_A(\varphi_1) \cup \mathcal{L}_A(\varphi_3), \quad \mathcal{L}_A(\varphi_2 \vee \varphi_3) = \mathcal{L}_A(\varphi_2) \cup \mathcal{L}_A(\varphi_3).$$

Since $\mathcal{L}_A(\varphi_1) \subseteq \mathcal{L}_A(\varphi_2)$, we have

$$\mathcal{L}_A(\varphi_1) \cup \mathcal{L}_A(\varphi_3) \subseteq \mathcal{L}_A(\varphi_2) \cup \mathcal{L}_A(\varphi_3), \text{ hence } \varphi_1 \vee \varphi_3 \preceq \varphi_2 \vee \varphi_3$$

. □

Proposition 3.4 establishes that LTL refinements are preserved under disjunction. Specifically, if a property φ_1 refines φ_2, then conjoining an additional property φ_3 with a disjunction preserves the refinement: $\varphi_1 \vee \varphi_3 \preceq \varphi_2 \vee \varphi_3$. Moreover, φ_1 also refines any weakened version of φ_2 obtained via disjunction, $\varphi_2 \vee \varphi_3$. This is useful in practice for modular and incremental specification evolution: when extending a system specification with alternative acceptable behaviors, previously established refinements remain valid. In particular, it ensures that adding optional behaviors or relaxing constraints through disjunction does not break correctness relationships already proven, allowing flexible property management and incremental verification of complex systems.

4 MINIMAL SPECIFICATION SUBSETS VIA PROPERTY REFINEMENT

This section introduces our methodology for the identification of a minimal subset of properties whose verification guarantees the entire suite's correctness. Traditional independent verification of each formula is computationally burdensome for complex properties and requires a high engineering effort to be maintained; our approach, based on property refinement (c.f. Sections 3), aims at mitigating this. The key insight is that properties often exhibit refinement relationships. This refinement relation allows to replace the weaker properties with the refining one, while guaranteeing that any system satisfying the latter also satisfies the former.

The core of our approach is to solve the following formally defined problem, which aims to identify a minimal, sufficient subset of properties that preserves the full verification strength of the original suite.

Definition 4.1 (The Minimal Sufficient Subset Problem). Let $\Phi = \{\varphi_1, \ldots, \varphi_n\}$ be a finite set of LTL properties, referred to as the *property suite*. The problem is to find a subset $\Phi' \subseteq \Phi$ that satisfies two conditions:

(1) Sufficiency (Correctness): The set Φ' must be a *sufficient subset* of Φ. A subset Φ' is sufficient if whenever all properties in Φ' are satisfied, then the conditions in the full set Φ are also satisfied. In other words, $\Phi' \models \Phi$.
(2) Minimality (Optimality): The sufficient subset Φ' must be minimal. This means no formula can be removed from Φ' without violating the sufficiency condition. Formally, $\forall \psi \in \Phi'$, the set $\Phi' \setminus \{\psi\}$ is not a sufficient subset of Φ

Condition (1) requires the use of Definition 3.1(b) to ensure that the satisfaction of the properties in Φ' guarantees the satisfaction of all properties in Φ. If trivially false properties (e.g., `false`) were to be present in Φ, they would refine all other properties, leading to a trivial solution of $\Phi' = \{\texttt{false}\}$, which would not impose meaningful constraints on the system nor be sufficient for guaranteeing its satisfaction.

The remainder of this section details our two stage methodology. We first describe how we partition the property suite into equivalence classes based on shared atomic propositions and how to filter contradictions and tautologies (Algorithm 1). Then, in Algorithm 2, we detail how we determine refinement relations within each class and leverage these to identify a minimal sufficient subset of properties for verification. We also provide a comprehensive complexity analysis and discuss practical heuristics for ensuring scalability in complex verification scenarios.

4.1 Partitioning Properties by Shared Atomic Propositions and Filtering of contradictions and tautologies

Algorithm 1 Partitioning by Shared Atomic Propositions with Automata Filtering

Require: A set of LTL formulas $\Phi = \{\varphi_1, \ldots, \varphi_n\}$.
Ensure: A partition C of Φ, A map $\mathcal{A}$ of formula to automata

```
Initialize the Union-Find and the maps L : A → 2^{1,...,n}, 𝒜 : Φ → Automata.
for all φ ∈ Φ do
    A_φ ← ConstructAutomata(φ)
    if L(A_φ) ≠ ∅ and L(A_φ) ≠ (2^A)^ω then
        𝒜[φ] ← A_φ
        for all p ∈ AP(φ) do
            L[p] ← φ.
        end for
    end if
end for
for all S ∈ range(L) do
    Union all indices in S using the Union-Find.
end for
Let C be the partition of Φ induced by the Union-Find.
return (C, 𝒜).
```

The first stage of our approach has three main purposes: (1) to construct the equivalent automata, (2) to filter out trivially false or tautological properties, and (3) to partition the property suite Φ into equivalence classes based on shared atomic propositions.

The first two steps are straightforward. For each formula $\varphi \in \Phi$, we construct its corresponding Büchi automaton A_φ using the `ConstructAutomata` function (line 3). We then check whether the language of A_φ is empty (a contradiction) or universal, i.e., it accepts all words over the common alphabet A (a tautology). If neither is the case, we store A_φ in the map $\mathcal{A}$ that associates each formula with its automaton. This step particularly ensures that any trivially false properties, which would otherwise refine all other properties and lead to a trivial solution, are excluded from further consideration. The remaining properties into are then partitioned into equivalence classes based on shared atomic propositions. We initialize a Union-Find data structure to manage the merging of indices (line 1) and a map L that associates each atomic proposition with the list of formula indices containing it. For each formula, we update L by associating its atomic propositions with its index (lines 7-10). After processing all formulas, we iterate over the values in L and union their indices in the Union-Find structure (lines 11-13). Finally, we derive the partition C of Φ from the Union-Find structure and return it along with the map of automata (lines 14-15). This step is used to prune the search space for the subsequent refinement analysis, as it restricts pairwise checks to properties that share atomic propositions. This restriction is *complete* for all $\varphi' \preceq^+ \varphi$ with φ' satisfiable and $\varphi \not\equiv \top$: if $\varphi' \preceq^+ \varphi$ and $\varphi \not\equiv \top$, then $AP(\varphi') \cap AP(\varphi) \neq \emptyset$, hence φ', φ lie in the same AP-connected component. We remove contradictions ($\bot$) up front through emptiness checking and treat $\top$ as a universal sink (any satisfiable formula refines $\top$, hence can be removed as well), so cross-component comparisons are unnecessary and do not affect source selection or sufficiency

4.2 Determining Refinement Relations

To determine refinement relations between the equivalence classes, we adopt a structured approach based on the properties identified in Section 3. For the rest of this section, we focus on a single equivalence class C_i, as the analysis can be performed independently and then combined.

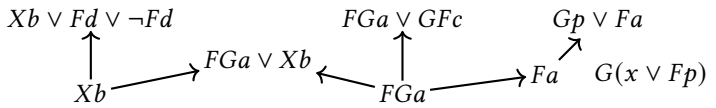


Fig. 2. Example refinement relations among constituents of an equivalence classe. Arrows indicate refinement relations pointing towards the weaker property.

Consider the example in Figure 2. A property like FGa (eventually, always a) is a refinement of the subformula Fa (eventually a), then verifying FGa suffices to ensure the satisfaction of both the FGa itself and Fa due to language inclusion. The goal of the algorithm is to find all such properties.

To formalize the selection of the minimal set, let $\Phi(C_i) = \{\varphi_1, \ldots, \varphi_{|C_i|}\}$ denote the set of all LTL formulas in the class C_i. The refinement relation $\preceq^+$ (Definition 3.1) establishes a preorder on $\Phi(C_i)$. The quotient $L = (\Phi(C_i))/\equiv$, forms a poset. Given that L is finite and equipped with meet (conjunction) and join (disjunction), it constitutes a complete lattice (cf. Proposition 3.2 and standard lattice theory results). Let Φ_i' represent the set of most-refined properties of the class i.

The algorithm, outlined in Algorithm 2, constructs a directed graph where nodes represent properties and edges denote refinement relations. It iteratively checks each pair of properties (ϕ, ψ) in the class to determine if ϕ refines ψ. This is achieved by retrieving the automata A_ϕ and A_ψ from the previously obtained map (Algorithm 1) and checking for language inclusion $L(A_\phi) \subseteq L(A_\psi)$. Note that, by discarding trivially false properties in the previous step, emptiness checks are not required. If a refinement relation is established, it is recorded in the graph. As optimization, the transitivity of refinement is leveraged to expand the refinement nodes to upper properties. Finally, the algorithm identifies the minimal set of most-refined properties Φ_i' by selecting nodes with no

Algorithm 2 Minimal Refined Subset Finder for a Class

Require: The set $\Phi(C_i)$ of all properties in the class C_i, the map $\mathcal{A}$ of formula to automata
Ensure: Minimal set Φ'_i

$refinement \leftarrow \emptyset$ ▷ Refinement relation graph
for all $\phi \in \Phi(C_i)$ **do**
  for all $\psi \in \Phi(C_i)$ **do**
    if $\psi \in refinement[\phi]$ **then**
      continue ▷ Already known refinement
    end if
    $A_\phi, A_\psi \leftarrow \mathcal{A}(\phi), \mathcal{A}(\psi)$
    if $L(A_\phi) \subseteq L(A_\psi)$ **then** ▷ $\phi \preceq^+ \psi$ check
      $refinement[\phi] \leftarrow \psi$
      Expand($refinement[\phi], refinement[\psi]$) ▷ Transitive closure
    end if
  end for
end for
$\Phi'_i \leftarrow$ GetSourceNodes($refinement$)
return Φ'_i

incoming edges (source nodes), as they are the most refined properties, as exemplified in Figure 2. GetSourceNodes also deals with cycles which can arise when properties refine each other. In this case they are equivalent by anti-symmetry and one of them is arbitrarily chosen (c.f. Section 3).

Once $\Phi' = \bigcup_{C_i \in C} \Phi'_{C_i}$ is established, it defines a minimal sufficient subset of properties whose verification guarantees satisfaction of the entire specification suite. Importantly, refinement does not remove specifications: all properties in Φ are retained for documentation, traceability, and certification purposes, while Φ' is used solely to optimize verification, preserving the full requirements context. Consider again the example in Figure 2, where FGa is identified as the most refined property. The original properties Fa and $G(x \vee Fp)$ remain part of the specification suite, although only FGa needs to be verified. Similarly, in the example discussed in the introduction, only ϕ_3 needs to be verified, while ϕ_1 and ϕ_2 are retained for traceability. The refinement relations provide a semantic explanation for these reductions by making explicit which requirements are effectively enforced (e.g., $\Phi' = \{Xb, FGa, G(x \vee Fp)\}$ and $\Phi' = \{\phi_3\}$) and which are already guaranteed by stronger properties, ensured by the corresponding refinement relations. This approach allows engineers to focus verification effort on core requirements, while maintaining a comprehensive specification suite for documentation and traceability.

4.3 Sufficiency and Minimality

We prove that Algorithm 2 always returns a sufficient and minimal subset.

Sufficiency. Fix an equivalence class C with formulas $\Phi(C_i)$. Algorithm 2 builds the refinement graph $G = (V, E)$ where $u \to v$ iff $u \preceq^+ v$. Only source nodes are selected, thus $\forall \varphi \in \Phi'_i$ either φ has no relationships to any other properties, hence it's necessary to be verified, or is refining other properties in Φ'_i. In this case, by Definition 3.1(b) and Proposition 3.2, φ guarantees satisfaction of all properties reachable from it. Therefore, the satisfaction of Φ'_i guarantees the satisfaction of $\Phi(C_i)$, showing that Φ'_i is sufficient.

Minimality. For any $\psi \in \Phi'_i$, by definition of source there is no $\psi' \in \Phi'_i$ with $\psi' \preceq^+ \psi$. Hence removing ψ leaves some it uncovered, violating sufficiency. Therefore no proper subset of Φ'_i is sufficient, making it minimal.

4.4 Complexity of the algorithm

The computational complexity of Algorithm 2, excluding the language inclusion checks, is $O(|\Phi(C_i)|^2)$, where $|\Phi(C_i)|$ is the number of unique properties in $\Phi(C_i)$. The quadratic term arises from constructing the refinement graph (lines 2-13). The subroutine ConstructAutomata, however, dominates the complexity. Verifying whether $\phi \preceq^+ \psi$ involves language inclusion checks, which are PSPACE-complete for LTL. This is typically done by constructing equivalent Non Deterministic Büchi Automata (NBA), A_ϕ and A_ψ, and checking for language inclusion $L(A_\phi) \subseteq L(A_\psi)$. While NBA complementation can be avoided by checking $L(A_\phi \cap A_{\neg\psi}) = \emptyset$, the construction of A_ψ (or $A_{\neg\phi}$) from an LTL formula is exponential in the formula length in the worst case, i.e., $O(2^{|\varphi|})$, where $|\varphi|$ is the length of the LTL formula. Consequently, the overall complexity of the refinement analysis is $O(|C| \cdot |\Phi(C_i)|^2 \cdot 2^{|\phi|})$.

5 EVALUATION

The main goal of this section is to demonstrate that our approach is not only effective in theory but also provides significant, tangible benefits on the kinds of complex specification suites found in practice. Our evaluation addresses two fundamental research questions (**RQs**):

RQ1: To what extent can property refinement reduce the verification burden in practice?

RQ2: What is the computational overhead of refinement analysis, and how does it compare to potential verification savings?

Methodology. To answer the abovementioned **RQs**, we developed a tool implementing our refinement analysis pipeline, following the workflow illustrated in Figure 1. Given a specification suite consisting of a finite set of LTL properties, these are first partitioned into equivalence classes and the corresponding automata are constructed (Algorithm 1). We rely on the state-of-the-art tool Spot [13] for LTL-to-Büchi automata translation and use its built-in automata simplification techniques to reduce the size of the resulting automata. Algorithm 2 is then applied independently to each class to identify a minimal sufficient subset of properties, using Spot to perform language inclusion checks. This step corresponds to the *Refinement Analysis* phase shown in Figure 1. The reduced property set resulting from combining all the class-independent ones, as well as the original specification suite, are then evaluated by verifying each property against the system model using the Spot–LTSmin–DVE backend, as described in [38]. All verification runs use identical models, tool configurations, and computational settings, differing only in the set of properties being verified. This enables a direct comparison of verification effort before and after refinement. While our prototype relies on Spot for all stages of the pipeline, the proposed approach is fundamentally tool-agnostic and can be implemented using any automata-based model checking framework. We selected Spot due to its state-of-the-art performance and widespread adoption in the formal verification community, which supports reproducibility and facilitates usage, however, the automata-theoretic capabilities required by our approach—namely LTL-to-automata translation and language inclusion checking (with automata minimization being optional)—are also available in other tools, such as SPIN [19], which can therefore replace Spot. Furthermore, translation between different LTL syntaxes is straightforward, therefore, the model checking phase of the evaluation is not tied to a specific tool as well and it can be generalized to any model checkers supporting LTL specifications.

Evaluation Metrics. We measure refinement effectiveness across multiple dimensions:

- **Property Reduction Percentage (PRP)**: $(|\Phi_{original}| - |\Phi_{minimal}|)/|\Phi_{original}| \times 100\%$. Measures the percentage decrease in the number of properties requiring verification after refinement, hence assessing the proportion of the suite that can be considered redundant and can be omitted as it provides no further assurance.
- **Net efficiency Gain (NEG)** $T_{baseline}/(T_{optimized} + T_{refinement})$: Accounts for both the time saved in model checking and the overhead introduced by refinement analysis, providing a holistic view of workflow improvement. We account for a 3% tolerance when considering NEG to account for variability in measurement. We consider cases where the optimization leads to NEG > 1- tolerance as PNEG (positive NEG) or 'beneficial', otherwise they are referred to as NNEG (negative NEG).
- **Model Checking Speedup (MCS)** $T_{baseline}/T_{optimized}$: Quantifies the acceleration in verification time, comparing the baseline (all properties) to the optimized (minimal subset) suite. Differently from the NEG, this metric is particularly useful when a new iteration of the system model needs to be verified, as it shows how much faster the verification of the minimal set is compared to the baseline.
- **Refinement Overhead** $T_{refinement}/T_{baseline} \times 100\%$: Evaluates the computational cost of performing refinement analysis relative to the baseline verification time. This measures how much of the traditional model checking time is invested for the refinement process.
- **Net Benefit Percentage (NBP)** $(T_{baseline} - T_{optimized} - T_{refinement})/T_{baseline} \times 100\%$: Measures the overall verification workflow time after accounting for optimization costs. This is the immediate productivity gain on the first iteration.
- **Net Verification Reduction (NVR)** $(T_{baseline} - T_{optimized})/T_{baseline} \times 100\%$: Measure how much faster the optimized model checking is compared to the traditional one. This doesn't include the time spent doing the optimization itself, hence it represents the speedup achieved once the refinement check is complete and the model is iterated.
- **Brake even Factor (BEF)** $T_{refinement}/(T_{refinement} - T_{optimized})$: A metric indicating the point at which the time saved by optimizing the verification process exceeds the time spent on the refinement analysis. This factor helps determine how many iterations of model checking are needed to recoup the initial investment in refinement analysis.
- **Property Complexity Score (PCS)**: A composite metric quantifying the complexity of a property. The score is a weighted sum of structural complexity, which considers the formula's length, total operator count, number of unique atomic propositions, and parenthesis nesting depth; temporal complexity, based on a weighted count of temporal operators scaled by nesting depth; and Automaton Complexity, derived from the number of states and edges in the corresponding Büchi automaton.

These metrics collectively demonstrate not only the technical effectiveness of our approach, but also its practical value in real-world engineering scenarios. For instance, a high property reduction rate means engineers spend less time specifying and maintaining redundant properties, while a high NEG reflects a high model checking saving time even when accounting for its own computational cost. Table 1 presents a detailed summary of our findings.

Benchmark Selection. The evaluation has been conducted on a comprehensive set of 141 benchmarks derived from established verification literature [35], covering a wide range of domains and complexities. This benchmark suite has been chosen for two main reasons. First, it is particularly suitable for our study because each entry consists of a system model accompanied by a non-trivial

Table 1. Comprehensive Performance Analysis Across 141 Benchmarks

(a) Aggregate outcomes of the refinement process.

Metric	Value
Benchmarks with PRP > 0	109 (77.3%)
Gross Verification reduction	74164.28 (~20.6h)
Net Verification reduction	73445.29 (~20.4h)
Analysis Time	718.99 (~12m)
Net Benefit Percentage	26.70%
Net Verification Reduction	26.96%

(b) Profile of PNEG vs. NNEG Cases.

Metric	PNEG	NNEG
Total Cases	137 (97%)	4 (3%)
Average metrics		
Suite Size	21.7	160.8
PCS	12.6	18.8
Model size $\|TS\|$	1.47×10^{10}	672
PRP	36.5%	6.3%

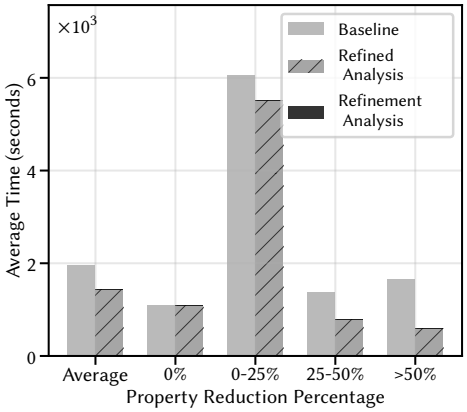
(c) Statistical Summary of Performance Metrics

Metric	Mean	Median	Best	Worst
PRP	35.95%	50.0%	75%	0%
MCS	1.63×	1.75×	4.2×	0.96×
NEG	1.52×	1.58×	3.4×	0.68×
Refinement Overhead	2.22%	0.035%	0.003%	54.5%

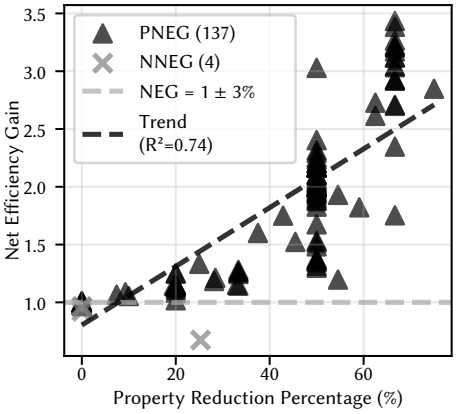
suite of LTL specifications. This characteristic reflects realistic verification scenarios in which multiple properties must be analyzed jointly. These benchmarks were not explicitly designed to study specification redundancy and their structure closely mirrors realistic verification workflows. Second, this set has been chosen to reflect a variety of real-world verification scenarios, ensuring that our evaluation captures the practical applicability and effectiveness of our approach across different contexts. These scenarios include:

- Mutual Exclusion Protocols (n=45): A collection of canonical algorithms for ensuring exclusive resource access, including well-known examples like Peterson's and Lamport's algorithm. These systems are ideal for testing our approach's ability to identify refinements among tightly coupled safety and liveness properties that define correct concurrent behavior
- Communication Protocols (n=27): Models focused on reliable data transmission and interaction, exemplified by the Bounded Retransmission Protocol (BRP) and the Rether protocol. This category challenges our analysis with properties rich in temporal sequencing, where high-level requirements (e.g., "a message is eventually delivered") are often implicitly refined by properties governing intermediate protocol stages.
- Distributed Algorithms (n=20): Systems designed for coordination in distributed environments, such as various leader election and publish-subscribe algorithms. The specifications for these systems are frequently hierarchical, testing our ability to uncover refinement relationships between general fairness properties and more specific, constrained behaviors.
- Scheduling and Resource Allocation (n=24): Problems involving the coordination of concurrent processes and the management of shared resources, including the Dining Philosophers problem, elevator schedulers, and automated driving controllers.
- General Model Checking Examples (n=25): A varied set of well-established verification problems that serve as standard case studies in formal methods, including Anderson's algorithm and Loyd's sliding puzzle. This group ensures the generality of our approach, demonstrating its effectiveness outside of specialized domains.

(a) Average verification time comparison across PRP categories.



(b) Overall analysis of property refinement effectiveness and efficiency across benchmarks.



The diversity is evident in both the scale and the intrinsic difficulty of the specifications. In terms of scale, the suite ranges from minimal systems defined by 2-10 properties to large-scale specifications with over 633 distinct formulas. Furthermore, our analysis of the properties themselves reveals a vast spectrum of PCS. This analysis confirmed a more than tenfold variation in average PCS across the benchmarks, from simple safety properties with an average score of 3.7 to intricate, deeply nested liveness properties with average scores exceeding 47. This multi-dimensional and quantitatively confirmed complexity range provides a robust and comprehensive basis for evaluating our method's performance across a realistic spectrum of verification scenarios, from lightweight checks to deeply challenging analyses.

Given their structure, diversity, and complexity, these benchmarks provide a meaningful and representative basis for evaluating the effectiveness and robustness of our approach. The presence of non-trivial specification suites, together with the fact that the benchmarks were originally designed for general verification rather than for studying specification redundancy, allows our evaluation to reflect practical challenges encountered in realistic verification workflows. As a result, the evaluation offers an informed assessment of the benefits and limitations of our property refinement methodology in applied verification settings.

5.1 RQ1: To what extent can property refinement reduce the verification burden in practice?

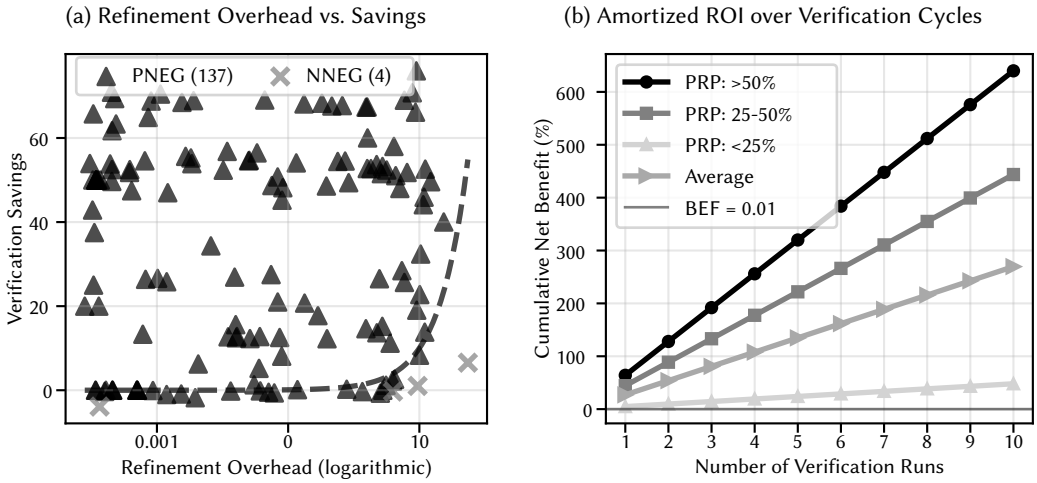
We provide an answer to **RQ1** through an analysis under two perspectives: quantitatively and qualitatively. The quantitative results presented in Tables 1a and 1c show that refinement is highly effective: about 97% of all benchmarks achieve a PNEG, i.e. reductions in verification effort and time. On average, the PRP was around 36%, showing that a significant portion of the original property suite (more than a third) is redundant and can be safely omitted without sacrificing verification quality. In the best cases, engineers can omit nearly 75% of properties, with an average and a median PRP of 36% and 50%, respectively. Figure 3a confirms that this reduction translates directly into faster verification: Figure 3a further illustrates that the time taken for the refined verification is consistently lower than the baseline, while the refinement overhead remains negligible across all PRP categories. Benchmarks with PRP ≈ 0 typically involve small models with little redundancy;

even there, the refinement overhead is negligible (~0.001s), making the analysis effectively cost-free. A typical benchmark achieves 1.75× MCS and 1.52× NEG, with more than half of all cases performing even better. Although a few challenging cases slightly lower the geometric means (1.62× and 1.52×), 97% of benchmarks still show PNEG, and the best cases exceed 4.2× speedup. Altogether, refinement saved nearly 21 hours of model-checking time at a cost of only 12 minutes. Figure 3b further illustrates the relationship between PRP and NEG. The positive correlation (R^2=0.74) shows that higher redundancy leads to greater efficiency gains, leading to peaks up to 3.4×. Notably, a significant number of benchmarks with low, or even zero, property reduction still result in a net benefit. This indicates that the computational overhead of our refinement analysis is often modest enough to not negate the overall workflow efficiency, making it a low-risk, high-reward optimization strategy in most practical scenarios. In the rare NNEG cases, three yielded only minor slowdowns (NEG = 0.962, 0.958, 0.922), and the remaining case had high PCS and a relatively small model, making the overhead difficult to amortize.

In addition to these quantitative benefits, refinement also provides qualitative advantages, even in NNEG cases. The identification of redundant properties helps engineers focus on the core requirements, reducing cognitive load and simplifying maintenance. While redundant ones are omitted during verification, all properties remain available and refinement relations can then be traced back to requirements to help engineers in understanding whether redundancy arises from poor elicitation, incorrect formalization, or design evolution, further reducing engineering and maintenance effort. Even in 0-PRP cases, refinement confirms the minimality of the specification suite, providing assurance that all properties are necessary, which is valuable for system certification and compliance.

In summary, the evaluation demonstrates that property refinement is a powerful technique for reducing the complexity and cost of formal verification in practice. It achieves significant reductions in verification time while also enhancing the manageability and clarity of specification suites, making it a valuable tool for engineers working on complex verification scenarios.

5.2 RQ2: What is the computational overhead of refinement analysis, and how does it compare to potential verification savings?



Tackling **RQ2** involves investigating the trade-off between the computational cost of the refinement analysis and the resulting verification savings.

Overall, the technique demonstrates a compelling cost-benefit profile. It achieves a 26.9% NVR, a significant acceleration of the most computationally intensive part of the verification process, while the one-time refinement overhead remains extremely low at 0.26% of the baseline verification time (almost imperceptible as shown in Figure 3a). As shown in Figure 4a and Table 1c, the distribution of overhead is highly skewed: the median overhead is only 0.035%, indicating that refinement is effectively negligible for most benchmarks. The mean value (2.2%) is inflated by a single outlier NNEG case with a 55% overhead, although measuring a 25% PRP. This means that for engineers, refinement can be treated as an extremely lightweight preprocessing step with a very low expectation of significant upfront cost, which, in addition to the benefits discussed in Section 5.1, makes it a highly attractive option. The economic viability of this trade-off is further underscored by a BEF of 0.01. This indicates that, on average, the verification time saved by running the refinement analysis is greater than the time spent on the refinement analysis itself, delivering a positive return on investment (ROI) on the very first verification run. In other words, the process not only pays for itself immediately but also provides a net productivity gain from the outset.

The most significant benefits, however, are realized when applied to modern, iterative development workflows. The analysis overhead is a sunk cost, incurred only once for a given property suite. In contrast, the verification savings are realized with every subsequent run. Therefore, for every follow-up task, such as regression testing or verifying a modified model, engineers benefit from an average ∼27% pure reduction in verification time. This trend is visualized in Figure 4b. Suites with large PRP (≥25%) experience substantial savings on the first run, reaching a cumulative time benefit of 100% within two verification cycles depending on the PRP, which is equivalent to the cost of one entire baseline run—effectively providing a full verification for free. Problems with lower PRP show a more gradual improvement, but still achieve a positive ROI within the first iteration reaching a 50% benefit within 10 runs. Nevertheless, this shows how even for problems with moderate PRP, the refinement analysis is a net positive investment, compounding its value over the software lifecycle.

5.3 Discussion

Some comments are in order. First, regarding the complexity of the analysis. While our evaluation demonstrates that the overhead is generally negligible, it is important to acknowledge that the worst-case complexity of the refinement analysis remains exponential, as highlighted in Section 4.4.

This complexity can be mitigated through several heuristics. Firstly, caching and parallelization offer the most potential. Algorithm 1 provides already ground for parallelization, as each class can be processed independently. Alternatively, within each class, the pairwise refinement checks (lines 4-11 of Algorithm 2) can also be parallelized, as they are independent. This process, however, might increase complexity if optimizations based on transitivity are used, as they require inter-class knowledge. Additionally, caching results of previous refinement checks can prevent redundant computations. Finally, the of use syntactic implication rules (e.g. $F\phi \sqsubseteq_{\text{syn}} F\psi$ if $\phi \sqsubseteq_{\text{syn}} \psi$) provide a fast, sound, but incomplete method for establishing language inclusion, hence refinement. Various rules can be found in the literature [6, 13, 43, 45].

Considering this complexity, our refinement-based approach becomes particularly advantageous under the following conditions.

First, it is beneficial in cases of high redundancy among properties. This commonly occurs when specifications originate from multiple sources (e.g., different teams or departments). When many formulas refine others, i.e., $|\Phi'| \ll |\Phi|$, the minimal set Φ' becomes much smaller than the original

suite, reducing the number of required model-checking runs. Figure 3b demonstrates the strong correlation between redundancy and verification efficiency.

Second, the approach excels in large system models. For systems with large $|TS|$, the model-checking cost $O(|TS| \cdot 2^{|\phi|})$ per property dominates the workflow. The preprocessing cost $O(|\Phi|^2 \cdot 2^{|\phi|})$ is quickly amortized if it avoids multiple expensive verification runs. This trend is evident in Table 1b and Figure 4b. Such scenarios are typical in iterative development and regression testing, where refinement is performed once while verification is repeated, producing cumulative savings over successive iterations. Table 1b and Figure 3a further support these observations. NNEG cases generally feature high PCS and large property sets combined with small system models, leading to low baseline verification time and limited opportunity for amortization. Conversely, PNEG cases exhibit the opposite characteristics, yielding a more favorable cost-benefit balance and highlighting when refinement is most impactful.

Table 1b and Figure 3a provide evidences for these claims. NNEG cases are defined by the very factors that dominate the analysis cost. In these cases, both the PCS and the number of properties are high. Simultaneously, these cases are associated with smaller system models where the baseline verification time is relatively low. This explains why, despite even when achieving significant PRP, these scenarios yield a low NVR. For example, in the case with the lowest measured NNEG (NEG ≈ 0.68), the specification comprises 633 properties with an average PCS of 47.1, while the corresponding $|TS| = 1.35 \times 10^5$. Compared to the average benchmarks, both the PCS and the specification suite size are substantially higher, whereas the transition system is several orders of magnitude smaller than the average model size in our benchmark set. This imbalance, characterized by a very large and complex specification suite applied to a relatively small model, makes the overhead difficult to amortize, even when the PRP is around 25%. On the contrary, PNNEG cases are characterized by an opposite combination of those factors, leading to a more favorable cost-benefit trade-off for our approach and confirming the conditions under which our approach is quantitatively most beneficial.

This efficiency extends to the evolution of the property suite itself. When a new property ψ is added, a full re-analysis is not required. Instead, the system only needs to check if the new property refines or is refined by the existing properties, an operation with complexity $O(|\Phi| \cdot 2^{|\psi|})$. This incremental approach leverages the initial investment, ensuring that the analysis remains computationally inexpensive throughout the software lifecycle. This combination of a one-time amortizable cost and efficient incremental updates solidifies the methodology's value as a compounding efficiency gain over a project's entire lifecycle.

An important aspect to consider is the impact of equivalence classes on the refinement analysis (Algorithm 1). If property sets operate on disjoint APs, comparing them is unnecessary, as already highlighted in Section 4.1.In our evaluation, most benchmarks were monolithic, so we observed only 1-3 classes; however, for larger industrial systems with distinct architectural components, it is natural to expect many more disjoint classes, potentially making this heuristic optimization more relevant.

Overall, refinement provides additional value as a property-suite validation tool. It is broadly applicable, reduces verification effort (**RQ1**), and introduces only minimal overhead (**RQ2**). Even in NNEG or PRP=0 cases, refinement clarifies semantic dependencies and confirms minimality, helping engineers focus on essential properties. These characteristics make refinement easy to integrate into verification workflows while offering sustained benefits at a negligible cost.

6 RELATED WORK

The concept of refinement in system modeling is well established through automata-theoretic notions such as simulation and bisimulation [10, 18, 25, 30, 34]. While model-level refinement is mature and widely used in compositional verification, its extension to *specifications* remains comparatively underexplored in the contex of verification effort minimization and engineering guidance in the form of redundancy identification, which is often dealt with at a natural language level [2, 21, 32, 33, 39]. Early work by Pnueli [37] and by Mokkedem and Méry [29, 31] introduced preliminary notions of specification refinement, but do not explore it further in the same context.

Other related efforts study property preservation across model evolution. For example, Event-B refinement [40] focuses on conditions ensuring that a property verified on an abstract model remains valid on its concrete refinement. This is orthogonal to our goal: they analyze how properties evolve with the model, whereas we analyze semantic relations *among* properties to identify a minimal sufficient subset. Gallardo et al. [17] refine LTL formulas to mitigate precision loss in abstract model checking by producing multiple variants of a single property. Unlike this work, which identifies stronger redundant properties in a concrete specification suite, their approach refines a property to adapt to abstraction levels, not to reduce verification effort.

Research on *minimal assumption refinement* [9, 42] targets synthesis, where refinement introduces environment assumptions to repair unrealizable specifications. In contrast, we study semantic refinement among valid LTL properties to detect logical containment and redundancy. Additional techniques such as vacuity detection [3, 16, 23, 27, 28], consistency checking [14], and specification mining [1, 24] address specification quality but do not capture semantic entailment or redundancy across a property suite. Vacuity detection analyzes a *single* property with respect to a model (or inherently) to determine whether parts of the formula are irrelevant or trivially satisfied. In contrast, our approach is model-independent and operates at the *suite level*, identifying semantic entailment relations between distinct properties and eliminating entire formulas that are redundant with respect to stronger ones. Specification mining techniques focus on discovering missing properties from system executions, whereas our goal is to analyze and structure an existing property suite. Formula simplification and clause-elimination techniques, such as [44] or the one implemented by SPOT [13], aim to rewrite an individual formula into an equivalent but structurally simpler one, without reasoning about redundancy across multiple specifications. These approaches are orthogonal to our method and can be applied in combination with it; indeed, our implementation leverages the simplification techniques provided by Spot as a preprocessing step.

Furthermore, the approach in [7] focuses on improving the efficiency of verifying multiple properties by reusing information across verification runs, such as learned invariants, reachable-state over-approximations, and circuit simplifications, while still checking all properties individually or in grouped form. In contrast, the property refinement approach operates at the specification level and reduces verification effort by identifying semantic refinement relations among properties, allowing a minimal set of stronger properties to be verified while guaranteeing the satisfaction of all weaker, redundant ones. Delta-debugging-based approaches reduce specification sets by iteratively removing properties and re-running verification to preserve a given outcome [46]. Adapted from test-case minimization, these techniques treat the specification suite largely as a black box and rely on repeated model-checking invocations to identify redundancy. While effective, their search-based nature makes them sensitive to the cost of verification. In contrast, our refinement-based reduction exploits semantic relationships between specifications, identifying refinement and entailment directly at the specification level. This enables a principled, model-agnostic and explanation-driven reduction of the specification suite without repeated verification runs, making our approach particularly suitable for large property sets where verification costs dominate.

Property subsumption is the closest related concept, particularly as explored in [5, 12]. In [12], implications and equivalences among LTL properties are identified via SAT-based reasoning in order to schedule verification tasks across structurally related models. However, subsumption is defined purely as logical implication. This notion is too weak to justify omitting properties from verification, since implication alone does not exclude vacuous entailments (e.g., an unsatisfiable property trivially implies all others), and therefore does not guarantee preservation of correctness. To mitigate this issue, the approach in [12] relies on additional model-checking runs. While effective, this makes the technique inherently dependent on the availability of models and introduces substantial computational overhead due to the model-checking runs, which tend to be expensive, as we empirically show in Section 5. Moreover, because the analysis is tied to specific models, it is sensitive to model evolution, requiring repeated analysis for iterative design processes. In contrast, our work is fundamentally based on *refinement*. By requiring meaningful containment as formalized in Definition 3.1(b), refinement guarantees both completeness and correctness while excluding vacuous implications. As a result, our method is model-independent, hence significantly less expensive than verification-based approaches, and robust under model evolution. These combined advantages are reflected in the evaluation results.

Similarly to our approach, the work in [5] also uses subsumption and it is model-independent, but it suffers from two key limitations: *poor computational scalability* and *limited semantic insight*. Determining whether a property is subsumed requires checking all 2^{n-1} subsets of the remaining $n-1$ properties, resulting in exponential blowup. This cost is further amplified by repeated LTL-to-automata translations needed for each implication check, which are already exponential in the size of the formula. Since each subset check constructs increasingly complex conjunctions of properties, the automata generation step becomes even more expensive. In practice, the authors report that even for suites of only ten properties, the analysis can take hours and quickly becomes infeasible for more properties, thus not viable for real-world use cases. Our implementation of the approach confirms the statement from the authors, as it could not finish the benchmarks from Section 5 with more than 20 properties as automata became too complex. In contrast, our refinement-based approach avoids this combinatorial explosion. By exploiting the transitivity of refinement and its composition with conjunction (Proposition 3.3), we do not need to explore all subsets and can infer indirect refinement relations, drastically reducing the necessary checks and making our method scale quadratically in the number of properties. Comparing the benchmarks finished by both implementations in the same environment, our approach is on average two orders of magnitude faster.

A second limitation of the framework in [5] is the lack of semantic structure: because it operates on subsets, it does not provide insight into how individual properties relate to one another beyond subsumption. Consequently, it offers little guidance for understanding dependency, hierarchy, or equivalence within a specification suite. Our refinement-based method, by contrast, exposes a lattice of semantic relations, enabling engineers to identify equivalences (via antisymmetry), understand logical dependencies, and reason about specification structure. Together, these features provide both a scalable redundancy analysis and actionable insight for requirements engineering and specification maintenance.

7 CONCLUSION

Model checking using temporal logic is a key aspect of formal verification of modern complex software systems. The traditional “checklist” approach to model checking, however, is inadapt to the growing complexity of modern software systems, as it ignores semantic overlaps resulting from factors such distributed development over various teams and iterative design cycles. We address this by introducing and operationalizing a formal theory of property refinement for temporal logic

into a concrete methodology that automatically identifies redundant properties from a verification suite. We first establish the fundamental algebraic groundwork necessary for principled reasoning about specification strength and redundancy, then use it to develop an algorithm to identify the minimal, most-refined subset of specifications, the verification of which guarantees the satisfaction of the entire original suite. The extensive empirical evaluation demonstrated the practical efficacy of this methodology. The practical viability of this approach is not speculative; our evaluation demonstrates it can reduce the number of required verification tasks by up to 75% and accelerate model checking by up to three orders of magnitude, with a negligible associated computational overhead for the refinement analysis. Furthermore, it establishes how the one-time investment in the refinement analysis yields compounding returns over development lifecycles.

Considering the limitations identified in Sections 4 and 5, future work will concentrate on exploring more sophisticated heuristics to alleviate the exponential cost of the NBA construction. Further investigation into handling highly skewed formula distributions, heuristic to select relevant properties, and integrating our techniques into industrial-strength model checking toolchains also represent important next steps towards realizing the full potential of property refinement in practice.

Ultimately, this work establishes property refinement as a key technique for tackling the problem of specification debt that plagues large-scale software systems, reducing both the computational cost of formal verification and the cognitive load on engineers.

ACKNOWLEDGMENTS

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.

8 DATA AVAILABILITY

A replication package is available at https://github.com/PizzeriaAurora/LTLFSE.git. The authors are committed to making all data and code used in this study publicly available.

REFERENCES

[1] Glenn Ammons, Rastislav Bodík, and James R Larus. 2002. Mining specifications. *ACM Sigplan Notices* 37, 1 (2002), 4–16.

[2] Aayush Anand, Sushank Pandey, Kirtan Shah, and Chetan Mohnot. 2024. Optimizing Redundancy Detection in Software Requirement Specifications Using BERT Embeddings. SSRN preprint. https://doi.org/10.2139/ssrn.5059250 13 pages. Date written: 2 Dec 2024.

[3] Roy Armoni, Limor Fix, Alon Flaisher, Orna Grumberg, Nir Piterman, Andreas Tiemeyer, and Moshe Y. Vardi. 2003. Enhanced Vacuity Detection in Linear Temporal Logic. In *Computer Aided Verification*, Warren A. Hunt and Fabio Somenzi (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 368–380.

[4] Christel Baier and Joost-Pieter Katoen. 2008. *Principles of Model Checking*. MIT Press.

[5] Jiří Barnat, Petr Bauch, Nikola Beneš, Luboš Brim, Jan Beran, and Tomáš Kratochvíla. 2016. Analysing sanity of requirements for avionics systems. *Formal Aspects of Computing* 28, 1 (March 2016), 45–63. https://doi.org/10.1007/s00165-015-0348-9

[6] Ilan Beer, Shoham Ben-David, Cindy Eisner, Dana Fisman, Anna Gringauze, and Yoav Rodeh. 2001. The Temporal Logic Sugar. In *Proceedings of the 13th International Conference on Computer Aided Verification (CAV'01) (Lecture Notes in Computer Science, Vol. 2102)*, Gérard Berry, Hubert Comon, and Alain Finkel (Eds.). Springer, 363–367. https://doi.org/10.1007/3-540-44585-4_33

[7] G. Cabodi and S. Nocco. 2011. Optimized model checking of multiple properties. In *2011 Design, Automation & Test in Europe*. 1–4. https://doi.org/10.1109/DATE.2011.5763279

[8] Edward Ralph Carroll and Robert Joseph Malins. 2016. Systematic literature review: How is model-based systems engineering justified? (2016).

[9] Davide G. Cavezza, Dalal Alrajeh, and Andás György. 2020. Minimal Assumptions Refinement for Realizable Specifications. In *Proceedings of the 8th International Conference on Formal Methods in Software Engineering (FormaliSE '20).*

Association for Computing Machinery, New York, NY, USA, 66–76. https://doi.org/10.1145/3372020.3391557
[10] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. 1999. *Model Checking*. MIT Press.
[11] Joaquim Motger de la Encarnación. 2020. *Automated Similarity Detection: Identifying Duplicated Requirements*. MSc Thesis. Universitat Oberta de Catalunya. https://hdl.handle.net/10609/105807
[12] Rohit Dureja and Kristin Yvonne Rozier. 2018. More Scalable LTL Model Checking via Discovering Design-Space Dependencies (D^3). In *Tools and Algorithms for the Construction and Analysis of Systems*, Dirk Beyer and Marieke Huisman (Eds.). Springer International Publishing, Cham, 309–327. https://doi.org/10.1007/978-3-319-89960-2_17
[13] Alexandre Duret-Lutz, Etienne Renault, Maximilien Colange, Florian Renkin, Alexandre Gbaguidi Aisse, Philipp Schlehuber-Caissier, Thomas Medioni, Antoine Martin, Jérôme Dubois, Clément Gillard, and Henrich Lauko. 2022. From Spot 2.0 to Spot 2.10: What's New?. In *Proceedings of the 34th International Conference on Computer Aided Verification (CAV'22) (Lecture Notes in Computer Science, Vol. 13372)*. Springer, 174–187. https://doi.org/10.1007/978-3-031-13188-2_9
[14] Steve Easterbrook and Marsha Chechik. 2001. A framework for multi-valued reasoning over inconsistent viewpoints. In *Proceedings of the 23rd International Conference on Software Engineering* (Toronto, Ontario, Canada) *(ICSE '01)*. IEEE Computer Society, USA, 411–420.
[15] Jeff Estefan. 2008. Survey of Model-Based Systems Engineering (MBSE) Methodologies. *INCOSE MBSE Focus Group* 25 (01 2008).
[16] Dana Fisman, Orna Kupferman, Sarai Sheinvald-Faragy, and Moshe Y. Vardi. 2009. A Framework for Inherent Vacuity. In *Hardware and Software: Verification and Testing*, Hana Chockler and Alan J. Hu (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 7–22.
[17] María-del-Mar Gallardo, Pedro Merino, and Ernesto Pimentel. 2002. Refinement of LTL Formulas for Abstract Model Checking, Vol. 2477. 395–410. https://doi.org/10.1007/3-540-45789-5_28
[18] Matthew Hennessy and Robin Milner. 1985. Algebraic Laws for Nondeterminism and Concurrency. *J. ACM* 32, 1 (1985), 137–161.
[19] Gerard J. Holzmann. 1997. The Model Checker SPIN. In *Proceedings of the 23rd International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 5–14. https://doi.org/10.1109/ICSE.1997.597796
[20] Muhammed Basheer Jasser, Mar Yah Said, Jamilah Din, and Abdul azim abdul ghani. 2019. A Survey on Refinement in Formal Methods and Software Engineering. *International Journal of Advanced Trends in Computer Science and Engineering* 8 (09 2019), 105–112. https://doi.org/10.30534/ijatcse/2019/1681.42019
[21] Elmar Juergens, Florian Deissenboeck, Martin Feilkas, Benjamin Hummel, Bernhard Schaetz, Stefan Wagner, Christoph Domann, and Jonathan Streit. 2010. Can clone detection support quality assessments of requirements specifications?. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE 2010)*, Vol. 2. ACM, Cape Town, South Africa, 79–88. https://doi.org/10.1145/1810295.1810308
[22] F. Kordon, P. Bouvier, H. Garavel, L. M. Hillah, F. Hulin-Hubard, N. Amat., E. Amparore, B. Berthomieu, S. Biswal, D. Donatelli, F. Galla, , S. Dal Zilio, P. G. Jensen, C. He, D. Le Botlan, S. Li, , J. Srba, Y. Thierry-Mieg, A. Walner, and K. Wolf. 2021. Complete Results for the 2021 Edition of the Model Checking Contest. http://mcc.lip6.fr/2021/results.php.
[23] Orna Kupferman and Moshe Vardi. 2000. Vacuity Detection in Temporal Model Checking. *International Journal on Software Tools for Technology Transfer (STTT)* 4. https://doi.org/10.1007/3-540-48153-2_8
[24] Davide Lorenzoli, Leonardo Mariani, and Mauro Pezzè. 2008. Automatic generation of software behavioral models. In *Proceedings of the 30th International Conference on Software Engineering* (Leipzig, Germany) *(ICSE '08)*. Association for Computing Machinery, New York, NY, USA, 501–510. https://doi.org/10.1145/1368088.1368157
[25] Nancy Lynch and Frits Vaandrager. 1995. Forward and Backward Simulations Part I: Untimed Systems. *Information and Computation* 121, 2 (1995), 214–233.
[26] Srinivas Malladi, G. Ramakrishna, K. Rao, and E Babu. 2016. Analysis of Legacy System in Software Application Development: A Comparative Survey. *International Journal of Electrical and Computer Engineering (IJECE)* 6 (02 2016), 292–297. https://doi.org/10.11591/ijece.v6i1.8367
[27] Shahar Maoz and Rafi Shalom. 2020. Inherent vacuity for GR(1) specifications. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 99–110. https://doi.org/10.1145/3368089.3409669
[28] Dor Ma'ayan, Shahar Maoz, and Jan Oliver Ringert. 2023. Anti-Patterns (Smells) in Temporal Specifications. In *2023 IEEE/ACM 45th International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*. 13–18. https://doi.org/10.1109/ICSE-NIER58687.2023.00009
[29] Dominique Mery and Dominique Roegel. 1994. *Refining formal specifications to get efficient, structured and correct concurrent programs*. Technical Report. Centre de recherche en informatique de Nancy, CNRS / Université de Nancy 1 / Université de Nancy 2 / Institut national polytechnique de Lorraine, Vandoeuvre-lès-Nancy, France.
[30] Robin Milner. 1989. *Communication and Concurrency*. Prentice Hall, Englewood Cliffs, NJ.

[31] Abdelillah Mokkedem and Dominique Méry. 1995. On using temporal logic for refinement and compositional verification of concurrent systems. *Theoretical Computer Science* 140, 1 (1995), 95–138. https://doi.org/10.1016/0304-3975(94)00206-X Third International Conference on Algebraic Methodology of Software Technology.

[32] Q. Motger de la Encarnación. 2019. *Automated similarity detection: identifying duplicated requirements.* Master's thesis. Universitat Oberta de Catalunya (UOC).

[33] E. D. Oktaviyani and Licantik. 2020. Redundancy Detection of Sentence Pairs in the Software Requirements Specification Documents with Semantic Approach. *IOP Conference Series: Materials Science and Engineering* 879, 1 (2020), 012078. https://doi.org/10.1088/1757-899X/879/1/012078

[34] David Park. 1981. Concurrency and Automata on Infinite Sequences. *Theoretical Computer Science* 25, 1 (1981), 167–183.

[35] Radek Pelánek. 2007. BEEM: Benchmarks for Explicit Model Checkers. In *Model Checking Software*, Dragan Bošnački and Stefan Edelkamp (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 263–267.

[36] Amir Pnueli. 1977. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*. 46–57. https://doi.org/10.1109/SFCS.1977.32

[37] Amir Pnueli. 1992. System specification and refinement in temporal logic. In *Foundations of Software Technology and Theoretical Computer Science*, Rudrapatna Shyamasundar (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1–38.

[38] Gaël L. Poncelet, Akos Kovács, Jérôme Ferreyra, Bernhard Krüger, Jacques Ferro, Christoph Heinrich, Axel Legay, and Thomas Wauters. 2025. ltsmin-dve Notebook. https://spot.lre.epita.fr/ipynb/ltsmin-dve.html. Accessed: 2025-11-25.

[39] Summra Saleem, Muhammad Nabeel Asim, and Andreas Dengel. 2025. PassionNet: An Innovative Framework for Duplicate and Conflicting Requirements Identification. *Expert Systems with Applications* 293 (2025), 128684. https://doi.org/10.1016/j.eswa.2025.128684

[40] Steve Schneider, Helen Treharne, Heike Wehrheim, and David M. Williams. 2014. Managing LTL Properties in Event-B Refinement. In *Integrated Formal Methods*, Elvira Albert and Emil Sekerinski (Eds.). Springer International Publishing, Cham, 221–237.

[41] Stephan Schneider, Jan Wollersheim, Helmut Krcmar, and Ali Sunyaev. 2018. 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 (06 2018), 151–170. https://doi.org/10.1057/s41265-016-0001-y

[42] R. Shalom and S. Maoz. 2023. Which of My Assumptions are Unnecessary for Realizability and Why Should I Care?. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, Melbourne, Australia, 221–232. https://doi.org/10.1109/ICSE48619.2023.00030

[43] Fabio Somenzi and Roderick Bloem. 2000. Efficient Büchi Automata for LTL Formulae. In *Proceedings of the 12th International Conference on Computer Aided Verification (CAV'00) (Lecture Notes in Computer Science, Vol. 1855)*. Springer-Verlag, Chicago, Illinois, USA, 247–263. https://doi.org/10.1007/10722167_21

[44] Martin Suda. 2015. Variable and Clause Elimination for LTL Satisfiability Checking. *Mathematics in Computer Science* 9, 3 (Oct. 2015), 327–344. https://doi.org/10.1007/s11786-015-0240-2

[45] Heikki Tauriainen. 2003. *On translating linear temporal logic into alternating and nondeterministic automata.* Technical Report A83. Helsinki University of Technology, Laboratory for Theoretical Computer Science, Espoo, Finland. http://www.tcs.hut.fi/Publications/A83.shtml Reprint of Licentiate's thesis.

[46] A. Zeller and R. Hildebrandt. 2002. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering* 28, 2 (2002), 183–200. https://doi.org/10.1109/32.988498