SigmaKEE-rs: An Embedded Ontological Reasoning System with Vampire as a Reusable Library Component

Theodore Kim and Adam Pease

Naval Postgraduate School, Monterey, CA, USA
theodore.kim@nps.edu, adam.pease@nps.edu

Abstract

We present `SigmaKEE-rs`, a re-implementation of the Sigma Knowledge Engineering Environment (SigmaKEE) in the Rust programming language, designed to serve as an ontological datastore backed by embedded theorem proving. A key contribution of this work is the integration of the Vampire theorem prover as an embedded library rather than as an external process, via extended Rust–C++ Foreign Function Interface (FFI) bindings built on the `vampire-rs` Rust library. We extend these bindings with support for Vampire's type system and clausification machinery. Using this strategy, the entire baseline ontology, the Suggested Upper Merged Ontology (SUMO)[1], is eagerly normalized, deduplicated, and held in an in-memory datastore alongside an incrementally maintained SUMO Inference Engine (SInE) relevance index. Subsequent queries bypass Vampire's parsing and the bulk of its preprocessing, invoking only clausification and the saturation engine on a relevance-filtered axiom set. We describe the engineering challenges encountered in adapting Vampire into a reusable component suitable for real-time, multi-query workloads, and report preliminary performance results comparing the embedded approach with the traditional process-based invocation model used in the original Java-based SigmaKEE.

Keywords: theorem proving, Vampire, SUMO, ontology, Rust, embedded reasoning

1 Introduction

The Suggested Upper Merged Ontology (SUMO) [1] is one of the most comprehensive formal, open-source ontologies available, comprising more than 20,000 terms and 80,000 axioms expressed in the higher-order SUO-KIF language [2]. SUMO has been mapped [3] to approximately 117,000 word senses in the WordNet lexicon [4], and has found application in domains ranging from natural language processing and information retrieval to legal reasoning and digital media [5].

The primary development tool for SUMO is the Sigma Knowledge Engineering Environment (SigmaKEE) [6], a Java-based system that provides browsing, editing, type-checking, and theorem proving capabilities over the ontology. SigmaKEE interfaces with first- and higher-order automated theorem provers (ATPs), mainly Vampire [7] and E [8], to answer queries against SUMO. In the standard SigmaKEE workflow, SUMO axioms are first translated from SUO-KIF into TPTP format [9] (either untyped first-order form (FOF) [10], typed first-order form with arithmetic (TFF) [11], or typed higher-order form (THF) [12]) and then passed to an external prover process for each query.

This process-based architecture currently works well as a server hosted application, where hardware accelerates the building and dispatching of queries to a latency of several seconds.

[1] https://www.ontologyportal.org

However, it presents a bottleneck for applications that require higher query throughput or are limited to less powerful deployment targets. Each invocation of the prover entails spawning a new operating system process, re-parsing the (potentially very large) axiom set, and communicating results via standard I/O or temporary files. For the full SUMO theory, the TPTP translation alone produces more than two million formulas [13], making the re-parsing a significant cost.

Two application domains motivate the need for a faster, more portable architecture. First, there is growing interest in using formal ontologies as grounded knowledge stores for large language models (LLMs), providing verified structured facts that can anchor LLM output and reduce hallucinations [14]. In such a setting, `SigmaKEE-rs` would serve as a verification backend: candidate assertions produced by an LLM are translated to SUO-KIF conjectures and checked against SUMO. The prover confirms or refutes them with formal guarantees that statistical models cannot provide. This use case demands sub-second query latency across potentially thousands of verification calls per session. The current process-based implementation cannot sustain this rate. Additionally, the use of theorem provers on low-powered edge devices is being further explored to provide concrete validations to algorithmic decision making [15]. The grounding of such systems in a formal ontology such as SUMO would improve their general applicability and correctness.

In this paper, we present `SigmaKEE-rs`, a from-scratch reimplementation of the core SigmaKEE functionality in Rust, with the Vampire theorem prover embedded as a native library component. Our approach differs from the standard architecture in three principal ways:

1. **Language choice.** We use Rust rather than Java, gaining predictable memory performance and safety without garbage collection pauses. This is particularly relevant for a system that holds large portions of its ontology directly in memory and necessitates aggressive memory management on resource-constrained systems.

2. **Embedded prover.** Rather than invoking Vampire as an external process, we link against it as a shared library via a Rust–C++ FFI layer, building on the `vampire-rs` Rust bindings [16] which we extended with support for typed logic and CNF-level access. This allows the axiom set to be loaded once and retained in memory across multiple queries, amortizing the substantial cost of parsing and preprocessing the SUMO theory.

3. **Eager normalization and incremental relevance indexing.** Rather than clausifying the SUMO axiom set up front, we eagerly rewrite it into conjunctive antecedent form (CAF) at startup and deduplicate formulas via a hash over their abstract syntax. Clausification is invoked only as a final deduplication pass over formulas that survive the fast AST-hash check (although this is disabled by default and is reserved for situations in which deep logical equivalence should be used to save on-disk cache footprint). In parallel, we maintain an incrementally updated SInE relevance index [17] that caches the per-symbol occurrence set and the per-formula minimum occurrence count, allowing queries to evaluate the D-relation at any tolerance without a rebuild. Subsequent queries bypass Vampire's parsing and most of its preprocessing, operating on the cached CAF formulas augmented only with the clausified conjecture.

The remainder of this paper is organized as follows. Section 2 provides background on SUMO, SigmaKEE, and the SUMO-to-TFF translation pipeline. Section 3 describes the architecture of `SigmaKEE-rs` and the `vampire-rs` FFI bindings. Section 4 discusses the engineering challenges of adapting Vampire for embedded, reusable operation. Section 5 describes the tar-

get application domains in detail. Section 6 presents preliminary evaluation results. Section 7 discusses related work, and Section 8 concludes with planned future directions.

2 Background

2.1 SUMO and SUO-KIF

SUMO originated in the year 2000 from the IEEE Standard Upper Ontology Working Group [1]. It is expressed in SUO-KIF (Standard Upper Ontology Knowledge Interchange Format)[2], a variant of KIF [18] that supports higher-order constructs including quantification over predicates, variable arity predicates, and modal operators. While this expressiveness is valuable for knowledge representation, it exceeded the capabilities of many automated theorem provers at the time SUMO was created, when few fully automated provers could handle logics beyond first-order logic.

2.2 Translation to First-Order Logic

To enable automated reasoning grounded in SUMO, a substantial translation pipeline has been developed [5, 11]. The translation from SUO-KIF to TPTP proceeds through several stages: flattening of higher-order constructs via predicate-argument expansion, axiomatization of the SUMO type hierarchy as explicit guard predicates, handling of arithmetic via the TFF theory of integers and reals, and various normalizations.

2.3 SigmaKEE

SigmaKEE [6] is a Java web application built on Apache Tomcat that provides a browser-based interface for SUMO development. SigmaKEE translates queries from SUO-KIF to TPTP, writes the translated theory plus the query conjecture to a temporary file, invokes an ATP (currently Vampire or E-Prover) as a subprocess, parses the prover's text output, and presents the result to the user.

2.4 Vampire

Vampire [7] is an automated theorem prover for first-order logic with equality. It supports TPTP (FOF, TFF, and THF) input formats, as well as SMT-LIB. Recent versions support portfolio mode, in which multiple proving strategies are tried in sequence with configurable time slices. Vampire is implemented in C++ for performance.

3 Architecture

The `SigmaKEE-rs` system consists of three principal layers, depicted in Figure 1: (1) a Rust core that handles input parsing, ontology management, and the TPTP translation pipeline; (2) the `vampire-rs` FFI bindings that expose Vampire's proving capabilities as a Rust library; and (3) a query interface layer. We describe each in turn.

[2] https://github.com/ontologyportal/sigmakee/blob/master/suo-kif.pdf

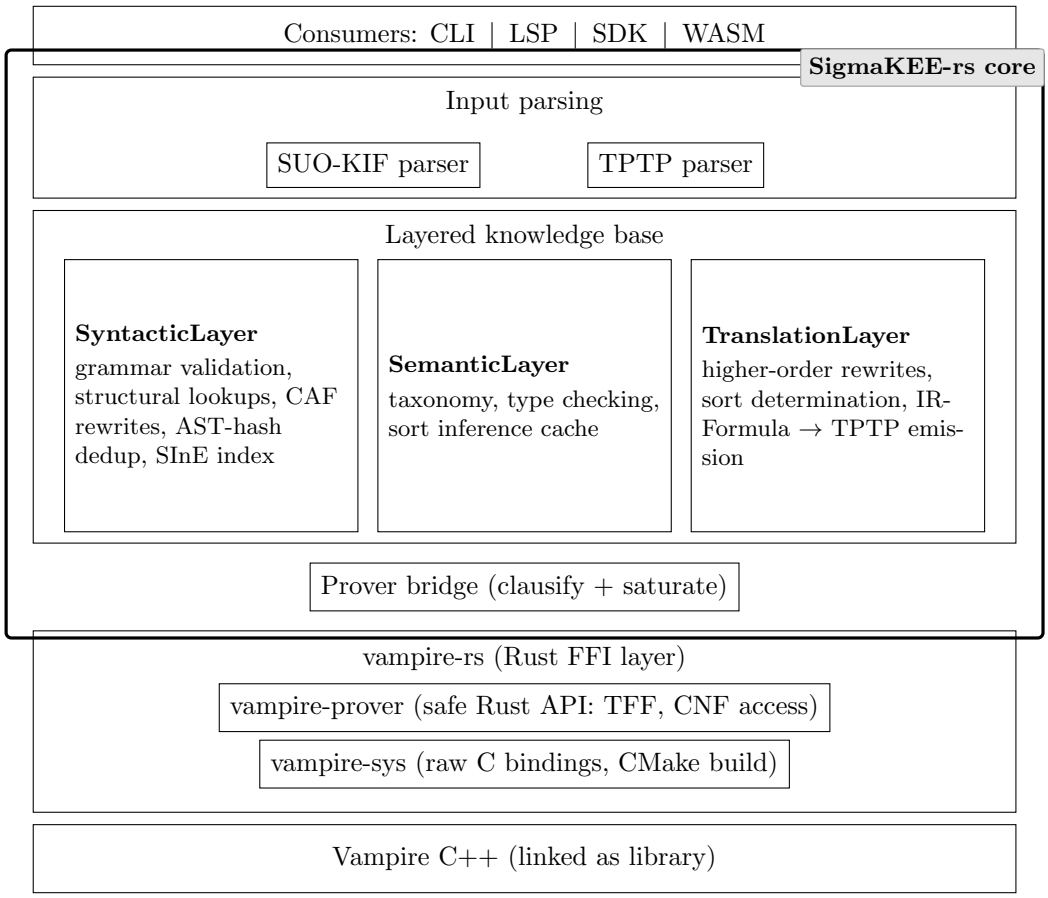

Figure 1: Architecture of `SigmaKEE-rs`. The Rust core is structured as a three-layer knowledge base sitting above a shared parse layer; CAF normalization, formula deduplication, and the incrementally maintained SInE index live in the `SyntacticLayer`. The prover bridge invokes Vampire through the extended `vampire-rs` FFI rather than as an external process.

3.1 The Rust Core

The Rust core of `SigmaKEE-rs` reimplements the essential functionality of SigmaKEE's Java codebase. This includes:

SUO-KIF parser. A recursive-descent parser for the SUO-KIF language, producing an abstract syntax tree (AST) representation. The parser handles the full SUO-KIF grammar including higher-order constructs, though not all constructs are supported in the downstream TFF translation (consistent with the limitations of first-order reasoning).

Ontology index. An in-memory data structure that indexes SUMO terms, relations, and axioms for efficient lookup. This supports operations such as retrieving all axioms mentioning a given term, computing the transitive closure of the subclass hierarchy, and finding type signatures for relations.

TFF translator. A translation pipeline from the internal SUO-KIF AST to TPTP (FOF/TFF) syntax, following the approach described by Pease [11]. The translator produces TFF type declarations, axioms, and conjectures suitable for input to Vampire. A key design decision was to structure this as an incremental pipeline: the base theory is translated once at startup and cached, while individual query conjectures are translated on demand and appended to the cached theory.

3.2 The `vampire-rs` FFI Bindings

Our work builds on the open-source `vampire-rs` project [16], which provides safe Rust bindings to the Vampire theorem prover structured as two sub-crates (a crate is the Rust-specific term for code libraries). The original `vampire-rs` exposes Vampire's core proving functionality through a clean Rust API: users construct problems using Rust's type system, and the library handles marshaling across the FFI boundary to invoke Vampire's C++ engine.

We forked `vampire-rs` and extended it in two important directions required for ontological reasoning with SUMO:

Type system bindings. The original `vampire-rs` exposes only untyped first-order logic (FOF). Since our TFF translation of SUMO relies heavily on Vampire's native type system, we extended the FFI layer to expose Vampire's internal type and sort infrastructure.

CNF-level access. We additionally exposed Vampire's clausification (conjunctive normal form conversion) machinery through the FFI boundary, enabling `SigmaKEE-rs` to invoke Vampire's preprocessing and clausification pipeline independently of the full proving loop. As described in Section 3.4, this capability is a component of our clause caching optimization.

The fork preserves the original two-crate split. The low-level `vampire-sys` crate drives the CMake-based Vampire build, compiles it as a static library linked into the final Rust binary, and generates C headers for the exposed subset of Vampire's API; `vampire-prover` is the safe Rust wrapper over those bindings. Our type-system and CNF extensions span both layers: new header declarations in `vampire-sys` and the typed-construction and `clausify` methods in `vampire-prover`.

3.3 Ingest Pipeline

When `SigmaKEE-rs` receives a new axiom, hypothesis (a query-specific assertion), or query (conjecture), the following steps occur:

1. The input is parsed from SUO-KIF into the internal AST representation.
2. In the case of a new axiom, the following additional steps occur:
   (a) The axiom is deduplicated against existing axioms in the ontology with a configurable level of granularity.
   (b) If the axiom includes a new symbol or taxonomy relation, it is inserted into the global ontology taxonomy, used later for typed first-order sort generation and type checking.
   (c) The formula's symbols' occurrence indices are updated and the formula added to the eagerly calculated SInE index cache.

3. The input is converted into an intermediate representation which has an $O(1)$ emission to either TPTP or Vampire's native formula representation. If being done ahead of a conjecture, the processing concludes here.

4. In the case of a conjecture, the following additional steps occur:

   (a) The conjecture's symbols seed a SInE BFS over the cached relevance index, selecting a tolerance-scaled subset of the cached CAF axiom set.

   (b) The selected axioms are passed to the theorem prover in a single pass.

   (c) The result (proved, refuted, or unknown) is returned to the caller, along with any proof object if one was found.

3.4 Eager Normalization and Incremental SInE Indexing

A central optimization in `SigmaKEE-rs` is the *eager normalization* of the SUMO axiom set together with an *incrementally maintained* SInE relevance index. In the standard Vampire workflow, each invocation begins with parsing, type checking, formula simplification, Skolemization, and clausification. For a theory the size of SUMO, these preprocessing stages dominate the time cost per query, yet their output is largely identical across queries that share the same axiom set.

Normalization. At system initialization, the full SUMO axiom set is rewritten into conjunctive antecedent form (CAF). CAF is a structural normal form for implications produced by four local transformations: biconditional splitting, hoisting of nested implications into conjunctive antecedents, distribution over disjunctive antecedents, and flattening of nested conjunctions in the antecedent. The result is a flat collection of implication formulas with conjunctive premises that is significantly more amenable to indexing than the original SUO-KIF surface form, while remaining a strict first-order rewriting. Choosing CAF rather than CNF avoids paying the full Skolemization and clause-splitting cost at startup, and keeps formulas in a shape that the relevance index can summarize symbolically.

SInE index caching. In parallel with normalization, we build a SInE relevance index [17] over the CAF axiom set. Traditional implementations recompute the relevant axiom subset per query; we maintain it eagerly. For each symbol s the index caches $\mathrm{occ}(s)$, the set of axioms containing s, and for each axiom A the minimum occurrence count $g_{\min}(A) = \min_{s \in \mathrm{syms}(A)} \mathrm{occ}(s)$. A per-symbol trigger list, sorted descending by $g_{\min}$, lets a query at tolerance t evaluate the D-relation $\mathrm{occ}(s) \leq t \cdot g_{\min}(A)$ by a single walk that short-circuits once $g_{\min}$ drops below $\lceil \mathrm{occ}(s)/t \rceil$. The cached structure is tolerance-independent: queries at different tolerances share one index, so t becomes a per-query parameter rather than a build-time constant. Insertion and removal are local to the portion of the ontology affected by the modification as only axioms whose $g_{\min}$ owner is among the affected symbols can change.

Complexity. Given R for the axiom set a query selects and k for the average number of symbols per axiom, the cached index performs a selection in $O(|R| \cdot k)$ time, proportional to the chosen subset rather than the whole ontology, while a tolerance change costs $O(1)$ and a single-axiom edit updates only the locally affected portion of the index; a traditional SInE instead rebuilds in time linear to the total number of symbol–axiom incidences on every tolerance change or edit. The cache pays for this with a heavier one-time cold build, quadratic in the

generality of the most frequently occurring symbols such as `instance` and `subclass`; however, this cost is amortized away in `SigmaKEE-rs`'s regime of a fixed base theory queried at varying tolerance.

Tolerance initialization and autoscaling. The tolerance t is one of SInE's free parameters. A poor choice of t under- or over-selects axioms for a given conjecture; because our cached index makes a tolerance change $O(1)$, we search for t rather than fix it. At query time we pick the largest t whose selected set, R, stays within a target axiom budget (configurable, ≈ 2000 by default) via a bounded binary search over the index's *tolerance breakpoints*—the discrete values of t at which R changes, read directly off the sorted trigger lists, each probe costing one $O(|R|\cdot k)$ selection. Since a single tolerance can still mis-size an individual conjecture, we use the prover to inform adjustments: saturation or counter-satisfiability (the conjecture is not entailed by the selected subset, *under*-selection) widens the budget, a timeout or resource-out (*over*-selection) narrows it. Once both directions have fired the optimum is bracketed and subsequent attempts bisect, terminating either after a small fixed number of consecutive non-proofs or when the caller's time budget is exhausted. The loop is cheap because every re-selection reuses the cached index ($O(1)$ tolerance change plus one $O(|R| \cdot k)$ walk) and never re-parses or re-clausifies the axiom set.

4 Engineering Challenges

Adapting Vampire from a standalone command-line tool to an embedded, reusable library component presented several significant engineering challenges. We discuss the most important ones here.

4.1 Global State and Reusability

Vampire, like many theorem provers, was designed under the assumption that it would be invoked once per problem and then terminate. This design is reflected in its use of global and static variables for configuration, symbol tables, and internal data structures. To employ Vampire as a reusable component, it is necessary either to reset this global state between invocations or to refactor it into instance-local storage.

Our current approach uses Vampire's existing library entry point, which provides a C-callable function that accepts a TPTP problem as input and returns a result status code. Between invocations, we must ensure that Vampire's internal state is properly cleaned up. This remains an area of active work, as not all of Vampire's internal allocators and caches are designed for reset. Rather than rewrite the internals of Vampire, we implemented a fork-and-wait methodology which spawns a new copy of the Vampire process, its memory preloaded with the problem in its native data structure. Upon completion, the memory cleanup is handled by the OS. While this is not an ideal implementation, it is a short-term solution to Vampire's inconsistent performance over subsequent reuses.

4.2 Thread Safety

Vampire is not thread-safe. Its global data structures are not protected by synchronization primitives, and concurrent invocations would lead to data races and undefined behavior. The `vampire-rs` crate addresses this by protecting all prover operations with a global mutex, serializing access to the Vampire engine. Although this prevents concurrent proofs, it is sufficient

for the single-query-at-a-time model that characterizes interactive ontology use. Currently, to accommodate higher-throughput scenarios, a pool of independent Vampire instances (each in its own process and address space) is employed. While this ensures the safety of Vampire's internal memory, it is slowed down by the nontrivial overhead of process spawning and means that the prover cannot operate over a shared state. Future work is planned to further push the boundary between `SigmaKEE-rs` and the Vampire library.

4.3 Memory Management Across the FFI Boundary

Rust and C++ have fundamentally different memory management models. Rust enforces ownership and borrowing at compile time, while Vampire uses a mix of manual memory management and custom allocators. To avoid utilizing the Vampire data structures directly, as implemented in its C++ source, `SigmaKEE-rs` implements its own `Formula` and `Problem` representations, which are serialized and cached, and are lowered to Vampire's representation just prior to transport across the language FFI boundary. This ensures that Rust can maintain ownership of the data in its native, optimized, and memory safe data structure up until the time that Vampire will consume it. It also lends itself to future prover integrations which may have different programming interfaces.

5 Target Applications

The two domains introduced in Section 1 share a requirement that optimizations presented by `SigmaKEE-rs` make possible to fulfill: reasoning against the complete ontology at low, predictable per-query cost. We describe the system-level implications of each.

5.1 Grounded Knowledge Store for Large Language Models

In the verification pipeline of Lupu et al. [14], natural-language claims are translated into SUO-KIF, aligned with SUMO concepts, and checked by Vampire, which classifies each claim as *True*, *False*, or *Unknown*. SUO-KIF is a practical target for this step because its s-expression syntax is more amenable to LLM generation than the flatter TPTP dialects. `SigmaKEE-rs` is intended as the verification backend for such a pipeline: an LLM emits candidate assertions, drawn for example from a document or a reasoning trace, and the system checks each against the full axiomatization of SUMO. A proof certifies the assertion as a consequence of the ontology; a refutation flags a contradiction.

The binding constraint is throughput. A single interaction can generate many facts to verify, and a retrieval-augmented pipeline may issue queries on every conversational turn. The embedded, cached architecture is what makes verification cheap enough to sit on this path: Section 6 shows it answers the full SUMO test suite at roughly two seconds per query where the process-based pipeline times out.

5.2 Ontology-Grounded Proving on Edge Devices

Edge deployment is attractive where network access is constrained, such as autonomous vessels and remote sensor networks, or where real-time decisions outpace the latency of remote computation. Haddou-Oumouloud et al. [15] survey formal-methods approaches to dependability in such systems and identify theorem proving as a route to sound, universally quantified guarantees; it scales more gracefully than enumerative methods like model checking when the property

under verification is logically rich. The limitation they note is that the driving axiomatizations are usually static and narrow, which constrains their use in dynamic settings. A broad, managed ontology gives a deployed system a stable vocabulary against which novel concepts can be classified at runtime rather than hand-patched into the axiom set.

Portability follows from the implementation. The Rust toolchain produces statically linked native binaries for common edge targets, and the same source compiles to WebAssembly for sandboxed runtimes such as Wasmtime, increasingly used at the edge for isolation and portable update distribution [19]. In every case the prover and the ontology ship as a single artifact, with no Java runtime or external Vampire binary to provision separately. Combined with the deduplicated CAF cache and SInE-filtered selection, which keep the resident working set small and predictable, this brings formally grounded reasoning into decision loops that have so far relied on hand-coded rules or opaque statistical models.

6 Performance Analysis

We present a performance analysis comparing `SigmaKEE-rs` against the traditional Java SigmaKEE workflow (which utilizes a standalone Vampire prover) over the standard SUMO inference test suite. The comparison is structured around two usage scenarios, a single query (cold start) and a repeated query session (warm start). Additionally, the analysis tests how each system scales as the underlying ontology grows from a small core to the complete SUMO knowledge base. These capture the two dominant modes of interaction with an ontological reasoning system and expose the qualitatively different behavior of the two pipelines at various scale.

6.1 Experimental Setup

All experiments were run on a single workstation (10 cores, 32 GB RAM) under `macOS 26.4`, with Vampire 5.0.1 as the prover in every configuration. The Java pipeline ran on OpenJDK; `SigmaKEE-rs` was a release build (`aarch64-apple-darwin`) with Vampire linked as an in-process library. Importantly, the two pipelines invoke Vampire in deliberately different modes. The Java pipeline runs Vampire in its `--mode casc` portfolio scheduler, which executes a sequence of pretuned strategies over the full emitted problem. `SigmaKEE-rs`, by contrast, invokes Vampire in its basic single-strategy mode (`--mode vampire`) with the prover's internal relevance filtering disabled (`--sine_selection off`): because axiom selection is already performed upstream by the cached index, the embedded prover is handed only the pre-selected, problem-relevant clauses and need not re-run relevance filtering itself.

The benchmark consists of the FOF-provable subset of the standard SUMO inference tests (the `plain` set, 39 problems). Each problem was run on three nested axiom-set tiers: (1) **core**, SUMO's foundational `Merge.kif` (5,477 axioms, 1,247 unique symbols); (2) **mid**, core plus the mid-level ontology (15,586 axioms, 3,780 symbols); and (3) **full**, the complete SUMO axiom set (106,633 axioms, 16,541 symbols). A query was included in a tier only when every ontology file it declares as a dependency was present in that tier, yielding 35, 37, and 39 runnable queries for core, mid, and full respectively. Each (query, tier) pair was executed four times to separate one-time setup cost from steady-state per-query cost, and Vampire was given a hard 60 s timeout per invocation. In total, 888 timed runs were recorded.

We measure two configurations:

C1. Java SigmaKEE + Vampire. SigmaKEE translates the active knowledge base to a TPTP representation (`SUMO.tptp`), emits the combined problem (the full KB plus the

Table 1: Scaling across the three axiom-set tiers. "Build" is the one-time cost of translating the KB to its prover image (C1) or building the cached representation (C2), paid once per session. "Solve/query" is the median per-query proof-search time once the KB is built. "Solved" is the number of queries answered correctly. All times are in seconds.

Tier (axioms)	Config.	Build (1×)	Solve/query	Solved
core	C1: Java	12.2	**0.23**	21/35
(5,477)	C2: **rs**	**1.8**	0.60	**28/35**
mid	C1: Java	12.7	**0.29**	20/37
(15,586)	C2: **rs**	**2.8**	0.88	**27/37**
full	C1: Java	32.9	**0.80**	18/39
(106,633)	C2: **rs**	**6.1**	2.16	**39/39**

test's hypotheses and conjecture), and Vampire is then run on that file. The entire translated KB is presented to the prover.

C2. SigmaKEE-rs (embedded, cached). A one-time `load` step parses, canonicalizes, rewrites, and persists the KB into a cached representation; each subsequent query then performs relevance selection over that cache and presents only the conjecture and the selected axioms before invoking the embedded prover.

We decompose each run into a setup phase (C1: KB-image translation/emit; C2: cache build) and a solve phase. The setup phase is paid only on the first run of each (query, tier) pair and reused thereafter, so the cold start (run 1) bears the full setup cost while the warm runs (runs 2–4) measure steady-state per-query cost. All figures are medians over the relevant runs, in seconds.

6.2 Results: Scaling Across Ontology Tiers

Table 1 reports, for each tier and configuration, the one-time KB build cost, the median per-query solve time, and the number of queries solved. The scaling behavior is the central result.

Two trends dominate. First, **build cost**: C2 constructs its cached representation 4–5× faster than C1 translates its prover image at every tier, and the gap widens with KB size (from 12.2 s vs. 1.8 s on core to 32.9 s vs. 6.1 s on full). Second, **per-query behavior**: C1's per-query time stays low on the queries it can solve, but the fraction it solves falls as the ontology grows (21/35 to 18/39); C2 instead answers a growing share, reaching 39/39 at full scale, at a modest and bounded per-solve cost.

The full tier is the clearest illustration. Handed all 106,633 axioms at once, Vampire spends its entire 60 s budget searching the full undifferentiated KB and fails to find a proof for 21 of 39 queries. C2 avoids this outcome: its SInE relevance selection serves Vampire only the axioms pertinent to each conjecture, so the problem the prover actually sees stays small and is solved in at most 2.42 s. Notably, C2 improves with KB size: queries unanswerable on the core and mid tiers (where the required axioms are simply absent) become provable once the full ontology is present, and SInE keeps each resulting problem tractable.

A complete per-query breakdown (the correctness and median solve time of every test across both configurations and all three tiers) is provided in Appendix A. It makes the trend visible at the level of individual problems: C1's column-by-column degradation (e.g. `TQG30` at 0.37 →

3.29 → 24.04 s, and the cluster of small-tier passes that flip to timeouts at full) against C2's uniformly solved full-tier column.

6.3 Cold and Warm Start

For a single query (cold start), C2 is faster end-to-end at every tier. The cold-start total (builds + first solve) has median 2.40 s vs. 12.43 s on core, 3.68 s vs. 12.99 s on mid, and 8.17 s vs. 33.70 s on full.

For repeated queries (warm start), the build is amortized away and the per-query cost is the solve phase alone. Here, the comparison is more subtle. On the queries that both systems solve, standalone Vampire is competitive or faster per solve (0.23 vs. 0.60 s on core, 0.29 vs. 0.88 s on mid, 0.80 vs. 2.16 s on full), because C2's solve time additionally includes its per-query relevance selection and has invoked Vampire sans its most aggressive optimizations. The difference is not raw per-proof speed on the shared problems but which problems each system can solve at all: at full scale C2's bounded ≈ 2 s per-query overhead buys a 39/39 solve rate, whereas C1's per-query cost is dominated by repeated 60 s timeouts on the 21 queries it cannot answer.

6.4 Discussion

`SigmaKEE-rs` demonstrates *capability under scale* rather than raw per-proof speed: as the ontology scales, the conventional workflow of presenting the universal axiom set to the theorem prover grows slower to build and progressively less able to answer queries. On the other hand, the `SigmaKEE-rs` caching-and-selection pipeline builds several times faster and answers every query at full scale, at the cost of a small, bounded per-query selection overhead. For the LLM-verification and edge deployment scenarios of Section 5, where reasoning must run against the complete ontology and per-query latency and reliability are both critical, this is the regime where `SigmaKEE-rs` offers an advantage.

7 Related Work

SUMO-to-TFF translation. Pease [11] described the translation of SUMO to TFF with level 0 polymorphism, which our TFF translator in `SigmaKEE-rs` reimplements. The TFF translation exploits Vampire's native type system and arithmetic, and produces a more compact theory than the earlier FOF translations.

SUMO-K to higher-order set theory. Brown, Pease, and Urban [20] extended the scope of SUMO reasoning to higher-order fragments via a translation to set theory, enabling interactive proof in systems like Megalodon. While this work pushes toward greater expressiveness, our focus is on making existing first-order reasoning faster and more practical for runtime use.

Vampire as a library. The idea of using Vampire as an embedded component has precedent in the RAPID software verification tool [21], which links against a specialized branch of Vampire for verification of loop programs. Our work differs in targeting a general-purpose ontological reasoning workload rather than program verification, and in providing general-purpose Rust bindings usable beyond a single application.

Ontology-grounded LLM verification. Lupu et al. [14] demonstrated that SUMO and Vampire can be used to fact-check LLM outputs by translating natural language claims into SUO-KIF and attempting proofs. Their pipeline uses the traditional process-based SigmaKEE architecture. Our work provides the infrastructure to make such verification fast enough for real-time, in-the-loop use, as discussed in Section 5.

8 Conclusion and Future Work

We have presented `SigmaKEE-rs`, a Rust-based reimplementation of the SigmaKEE ontological reasoning environment that embeds the Vampire theorem prover as a native library component. The key technical contributions are:

- Extensions to the `vampire-rs` crate [16] that expose Vampire's type system and clausification machinery through the Rust FFI, enabling typed (TFF) problem construction and CNF-level access directly from Rust without round-tripping through textual TPTP.
- An eager normalization-and-deduplication datastore: the SUMO axiom set is rewritten once at startup into Conjunctive Antecedent Form (CAF, all implications rewritten to have only a single conjunction in the antecedent), deduplicated via a hash over its abstract syntax (with an optional, deeper clause-level dedup pass), and held in memory in an intermediate representation with $O(1)$ emission to TPTP or Vampire's native formula structures.
- A cached, incrementally maintained SInE relevance index that makes axiom selection tolerance-independent: per-symbol occurrence sets and per-axiom minimum generality are held eagerly, so a query selects in time proportional to the chosen subset rather than the whole ontology, changing tolerance is $O(1)$, and ontology edits update only the locally affected portion of the index.
- An architecture for embedded, multi-query ontological reasoning that links Vampire as a library despite the engineering challenges of adapting a batch-oriented prover into a reusable component.

Several directions for future work are planned:

Higher-order logic (THF) support. The current system targets the first-order (TFF) fragment of SUMO. A significant portion of SUMO's expressiveness resides in higher-order constructs (e.g. quantification over formulas, `KappaFn`). We plan to extend `SigmaKEE-rs` with a translation pipeline targeting TPTP THF (Typed Higher-order Form), exploiting Vampire's growing support for higher-order reasoning [20]. This would bring `SigmaKEE-rs` closer to supporting the full semantics of SUMO, including constructs that have previously required interactive theorem provers such as Megalodon.

Incremental axiom loading. We plan to investigate whether Vampire's internal preprocessing can be made incremental, so that adding or removing axioms does not require reprocessing the entire theory. Combined with the clause cache, this would support dynamic ontology editing workflows where individual axiom changes invalidate only a subset of the cached clauses.

Parallel proving. The current architecture serializes all proof attempts through a global mutex. We plan to explore a multi-instance architecture where several Vampire instances run in parallel, potentially with different proving strategies (analogous to Vampire's portfolio mode, but with true parallelism).

LLM integration toolkit. Building on the grounded knowledge store concept, we plan to develop a toolkit that provides standardized interfaces between `SigmaKEE-rs` and popular LLM frameworks, including function-calling APIs for real-time verification and batch verification endpoints for offline grounding of generated corpora.

The source code for `SigmaKEE-rs` is available at https://github.com/ontologyportal/sigma-rs. The extended `vampire-rs` fork is available as a git submodule within that repository and independently at https://github.com/ontologyportal/vampire-rs; the original `vampire-rs` project from which our contributions were forked is at https://github.com/Dragon-Hatcher/vampire-rs.

References

[1] Ian Niles and Adam Pease. Toward a standard upper ontology. In Chris Welty and Barry Smith, editors, *Proceedings of the 2nd International Conference on Formal Ontology in Information Systems (FOIS-2001)*, pages 2–9, 2001.

[2] Adam Pease. *Ontology: A Practical Guide*. Articulate Software Press, Angwin, CA, 2011.

[3] Ian Niles and Adam Pease. Linking lexicons and ontologies: Mapping WordNet to the Suggested Upper Merged Ontology. In *Proceedings of the IEEE International Conference on Information and Knowledge Engineering*, pages 412–416, 2003.

[4] Christiane Fellbaum, editor. *WordNet: An Electronic Lexical Database*. MIT Press, Cambridge, MA, 1998.

[5] Adam Pease and Stephan Schulz. Knowledge engineering for large ontologies with Sigma KEE 3.0. In *Proceedings of the International Joint Conference on Automated Reasoning (IJCAR 2014)*, pages 519–525, 2014.

[6] Adam Pease and Christoph Benzmüller. Sigma: An integrated development environment for formal ontology. *AI Communications*, 26(1):79–97, 2013.

[7] Laura Kovács and Andrei Voronkov. First-order theorem proving and Vampire. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification (CAV 2013)*, volume 8044 of *LNCS*, pages 1–35. Springer, 2013.

[8] Stephan Schulz. E—a brainiac theorem prover. *AI Communications*, 15(2–3):111–126, 2002.

[9] Geoff Sutcliffe. TPTP, TSTP, CASC, etc. In *Computer Science – Theory and Applications (CSR'07)*, volume 4649 of *LNCS*, pages 6–22. Springer, 2007.

[10] Adam Pease, Geoff Sutcliffe, Nick Siegel, and Steven Trac. Large theory reasoning with SUMO at CASC. *AI Communications*, 23(2–3):137–144, 2010. Special issue on Practical Aspects of Automated Reasoning, IOS Press, ISSN 0921-7126.

[11] Adam Pease. Converting the Suggested Upper Merged Ontology to typed first-order form. *CoRR*, abs/2303.04148, 2023.

[12] Christoph Benzmüller and Adam Pease. Progress in automating higher-order ontology reasoning. In Boris Konev, Renate Schmidt, and Stephan Schulz, editors, *Workshop on Practical Aspects of Automated Reasoning (PAAR-2010)*, CEUR Workshop Proceedings, Edinburgh, UK, 2010.

[13] Adam Pease and Stephan Schulz. Contradiction detection and repair in a large theory. In *Proceedings of the 35th International FLAIRS Conference*, 2022.

[14] Daniela Lupu, Adrian Groza, and Adam Pease. Cross-validation of answers with SUMO and GPT. In *Proceedings of the 1st Workshop on Knowledge Base Construction from Pre-Trained Language Models (KBC-LM@ISWC 2023)*, 2023.

[15] Imane Haddou-Oumouloud, Abderahman Kriouile, Soukaina Hamida, and Ahmed Ettalbi. Toward secure and reliable IoT systems: A comprehensive review of formal methods applications. *IEEE Access*, 12:171853–171875, 2024.

[16] Dragon-Hatcher. `vampire-rs`: Safe Rust bindings to the Vampire theorem prover for first-order logic with equality. https://github.com/Dragon-Hatcher/vampire-rs, 2024. Version 0.5.1, available at https://crates.io/crates/vampire-prover.

[17] Kryštof Hoder and Andrei Voronkov. Sine qua non for large theory reasoning. In Nikolaj Bjørner and Viorica Sofronie-Stokkermans, editors, *Automated Deduction (CADE-23)*, volume 6803 of *LNCS*, pages 299–314. Springer, 2011.

[18] Michael Genesereth. Knowledge interchange format. In J. Allen et al., editors, *Proceedings of the Second International Conference on the Principles of Knowledge Representation and Reasoning (KR-91)*, pages 238–249. Morgan Kaufmann, 1991.

[19] Adam Hall and Umakishore Ramachandran. An execution model for serverless functions at the edge. In *Proceedings of the International Conference on Internet of Things Design and Implementation (IoTDI '19)*, pages 225–236. ACM, 2019.

[20] Chad E. Brown, Adam Pease, and Josef Urban. Translating SUMO-K to higher-order set theory. In Uli Sattler and Martin Suda, editors, *Frontiers of Combining Systems (FroCoS 2023)*, volume 14279 of *LNAI*, pages 255–274. Springer, 2023.

[21] Vampire Project. RAPID: Software verification with first-order reasoning. https://github.com/vprover/rapid.

A Per-Test Results

Table 2 gives the complete breakdown of the aggregated figures of Section 6: for every test, both configurations, and all three ontology tiers, it reports the median solve time (in seconds), or the failure mode otherwise. The data are the same 888 timed runs summarized in Table 1.

Table 2: Per-test outcomes. A number is the median solve time (seconds) over the repeat runs on a *correct* answer; TO denotes a timeout at the prover budget (60 s); × denotes a disproved result; and `--` denotes a query not runnable on that tier because a required axiom dependency is absent. C1 is the Java SigmaKEE+Vampire pipeline; C2 is embedded `SigmaKEE-rs`.

	C1: Java			**C2: SigmaKEE-rs**		
Test	core	mid	full	core	mid	full
SP01	0.19	×	0.78	0.69	0.89	2.08
SP02	0.19	0.34	0.81	0.63	0.90	2.06
SP03	–	TO	TO	–	0.88	2.07
SP04	–	TO	TO	–	×	2.06
SP26	–	–	TO	–	–	2.17
SP37	–	–	0.86	–	–	2.16
TQG1	TO	TO	TO	×	×	2.18
TQG2	TO	TO	TO	TO	TO	2.21
TQG5	TO	TO	TO	1.00	0.87	2.18
TQG6	TO	TO	TO	0.40	0.86	2.15
TQG7	TO	TO	TO	TO	0.92	2.16
TQG9	0.17	0.26	0.71	×	×	2.08
TQG10	TO	TO	TO	0.61	0.97	2.22
TQG12	0.19	0.28	0.79	0.60	0.90	2.21
TQG13	0.19	0.27	0.85	0.59	1.01	2.20
TQG14	TO	TO	TO	0.61	0.93	2.10
TQG15	0.36	0.60	1.47	0.61	0.94	2.11
TQG17	0.35	0.62	TO	0.62	×	2.18
TQG18	TO	TO	TO	0.59	0.93	2.20
TQG22	TO	TO	TO	0.58	0.85	2.22
TQG23	TO	TO	TO	0.61	1.25	2.42
TQG24	TO	TO	TO	0.65	0.88	2.26
TQG26	0.19	0.27	0.75	0.60	0.87	2.12
TQG29	0.11	0.23	0.78	0.60	0.86	2.18
TQG30	0.37	3.29	24.04	0.60	0.84	2.15
TQG31	0.39	0.40	1.22	0.59	0.87	2.14
TQG32	0.39	0.40	0.97	0.59	0.87	2.16
TQG34	0.13	0.23	0.80	0.58	0.86	2.16
TQG35	0.50	0.61	TO	0.60	×	2.12
TQG37	0.47	0.63	1.47	0.59	×	2.12
TQG38	TO	TO	TO	0.61	TO	2.10
TQG44	TO	TO	TO	0.64	0.85	2.09
TQG45	0.22	0.29	TO	0.60	0.85	2.12
TQG46	0.23	0.26	0.80	0.60	0.88	2.11
TQG47	0.23	0.26	0.78	0.60	0.87	2.16
TQG48	0.26	0.27	0.75	TO	TO	2.16
TQG49	0.23	0.27	0.71	TO	TO	2.17
TQG50	TO	TO	TO	TO	1.00	2.14
TQG51	0.20	0.61	TO	0.60	0.85	2.11