Noctua: Towards Automated and Practical Fine-grained Consistency Analysis

Kai Ma
University of Science and Technology of China (USTC)

Cheng Li
University of Science and Technology of China (USTC)

Enzuo Zhu*
UC Riverside, USA

Ruichuan Chen
Nokia Bell Labs, Germany

Feng Yan
University of Houston, USA

Kang Chen
Tsinghua University, China

Abstract

Relaxing strong consistency plays a vital role in achieving scalability and availability for geo-replicated web applications. However, making relaxation correct in modern implementations, typically written in dynamic languages and utilizing high-level object-oriented database abstractions, remains a challenge, despite the existence of numerous proposed analysis tools.

Here, we present a fully automated verification framework Noctua for understanding fine-grained consistency semantics in web applications. At its core is a simple intermediate representation SOIR that bridges the semantic gaps between high-level languages, database interactions and SMT solver's verification. Noctua's lightweight program analyzer reuses the ability of the language runtime and framework to translate consistency-related effects and application invariants from codebases into SOIR code. Finally, Noctua's verification backend maps SOIR code into SMT verification conditions with a new SMT encoding that decouples order information and thus allows more database semantics to be covered.

We have implemented Noctua[1] for a popular dynamic language, Python, and evaluated its correctness and applicability with two synthetic and four existing Python web applications. The evaluation results highlight that Noctua is able to understand consistency semantics considerably fast from the real codebases with no user input. For synthetic applications that existing solutions can be applied, Noctua's delivered consistent analysis results.

*Work done during his study at USTC.
[1]Source code available at https://github.com/noctua-sys/noctua.

CCS Concepts: • **Computer systems organization → Distributed architectures**; • **Software and its engineering → Formal methods**.

Keywords: Fine-grained distributed consistency, Program analysis, Consistency verification

ACM Reference Format:
Kai Ma, Cheng Li, Enzuo Zhu, Ruichuan Chen, Feng Yan, and Kang Chen. 2024. Noctua: Towards Automated and Practical Fine-grained Consistency Analysis. In *European Conference on Computer Systems (EuroSys '24), April 22–25, 2024, Athens, Greece.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3627703.3629570

1 Introduction

The ever-growing user base complicates the design and deployment of modern web applications. These web applications often leverage geo-replication to serve data from the replica closest to users, providing high availability, scalability, and fast responses. However, there exists a fundamental tension between maintaining strong consistency and achieving good system performance since strong consistency (e.g., View-stamped replication [29], Paxos [21], and Raft [31]) incurs high cross-replica coordination for serializing operations. To close this gap, recently, various weak consistency models [7, 8, 20, 23, 24] have been proposed to eliminate the ordering constraints as much as possible.

However, the safe use of weak consistency models requires careful reasoning about the outcomes of all possible concurrent execution of any pair of operations. This is because some harmful executions could lead to property violations, including state divergence and invariant violation, which should be avoided via adding runtime coordination. This process can be extremely time-consuming and error-prone [36]. Therefore, it is desirable to design a tool with full automation to relieve developers' burdens, which can leverage the power of SMT solvers to formally verify consistency-related semantics encoded in a large body of application operations written by high-level languages.

The use of dynamic languages (like Python [4] and PHP [3]) and high-level database abstractions (like Django [1] and Laravel [2]) are popular among modern web applications so as to achieve rapid evolution. Therefore, we must extract the database interactions from each code path to understand

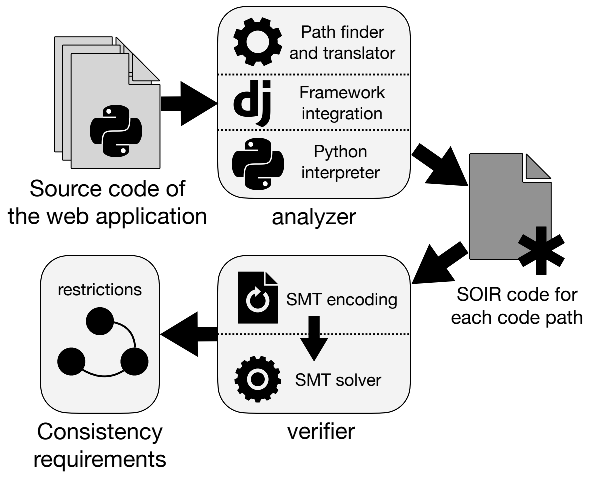

Figure 1. The architecture of the Noctua framework

the effect on the database state precisely because these interactions are executed on all replicas of the underlying geo-replication storage. Given the nature of dynamic languages, the characteristic of real-world web applications, and the expectation of the usefulness of the results, the fully automated system meets unique challenges for modern web applications.

(C1) Language Challenge: Existing static analysis approaches do not work for dynamic languages. For example, dynamic languages can change the variable type or import a module at any time during execution, i.e., such information is not available for a static analyzer. In addition, certain important widely-used features, like mixins, depend critically on dynamic features provided by the language. Such features cannot be statically analyzed.

(C2) Semantic Challenge: There are semantic gaps between the source language and the language for SMT solvers, which are used for consistency verification. In contrast to the source dynamic language, where one can make free use of external modules, mutable states, control flows, etc., the input of an SMT solver is a static description of a code path in first-order logic. Given their vast differences, it is not obvious how to bridge the two.

(C3) Coverage Challenge: A verifier needs to cover as many semantics as possible to output fewer but correct restrictions to improve the overall performance of the underlying geo-replicated storage. Unfortunately, prior works usually assume an orderless, purely key-value model, leading to more false positive results (unnecessary restrictions) [41].

Existing approaches cannot either achieve our goals of full automation or tackle the above challenges. For example, domain-specific languages and specification verifiers require substantial user input or rewriting existing code. Though the Rigi [41] analyzer works on unmodified source code, it assumes explicit and static SQL queries, which is not a realistic assumption in practice since database abstraction libraries construct SQL queries dynamically and lazily.

In response to the above challenges, we propose the Noctua framework, whose architecture is shown in Figure 1. Noctua reads the unmodified source code and outputs the fine-grained consistency restrictions (ordering constraints)for the underlying geo-replicated storage. ❶ The key in Noctua framework is its intermediate language design, called SMT-verifiable Object Language (SOIR). SOIR captures the database interactions from the automatic program analysis to address the semantic challenge (**C2**). SOIR has a rich set of primitives that correspond well to the abstractions shared by most object-oriented database abstractions. Meanwhile, our design also keeps SOIR simple to facilitate the ease of generation and verification. ❷ For automatic program analysis, our Analyzer is a plugin in the language *runtime*/*interpreter*/*virtual machine* to extract the database interaction from the source code, drawing inspiration from dynamic symbolic execution [12]. The plugin approach simplifies program analysis and covers more language semantics than manually developed independent tools, as it can harness the power of language runtime, addressing the language challenge (**C1**). ❸ The last component of Noctua is the Verifier, which generates the verification conditions, as the inputs for the integrated SMT solver, from the IR code in SOIR for each operation extracted according to a set of pre-defined checking rules. These rules are determined by the required consistency model and desired system properties, including state convergence and invariant preservation. Finally, the Verifier invokes the integrated SMT solver (such as Z3 [14]) to identify the set of conflicting pairs (the restriction set). Each invocation uses one rule and a pair of encoded operations. To address the coverage challenge (**C3**), we propose a new encoding schema for database tables, which includes order information. Such encoding also decouples the order information from the row data of the table, enabling more efficient array-based encoding for common cases where no order information is needed.

We implement the Noctua framework for Python and Django to verify its effectiveness. Four real-world and two synthetic applications are used to evaluate the correctness and applicability of the framework. The largest application, OwnPhotos, contains 9,174 lines of Python code and 545 code paths. The total CPU time for analysis and verification takes less than 6 hours without human intervention. This paper has made the following contributions:

1. We automate the program analysis and consistency verification using a modular framework, Noctua, for modern real-world web applications. Noctua can verify their fine-grained consistency semantics and generate proper ordering restrictions over their operations so as to avoid paying high coordination latency within the underlying geo-replicated storage.

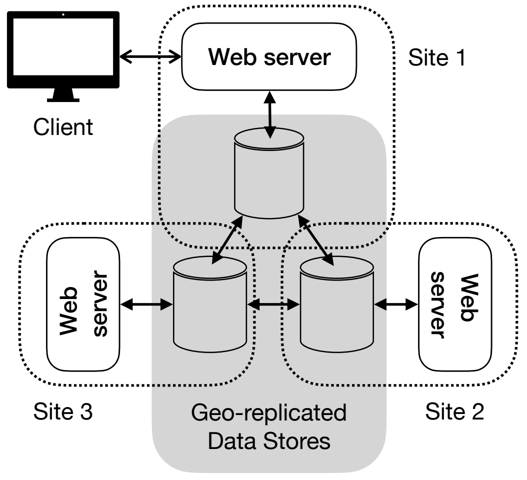

Figure 2. Web applications with geo-replicated storage spanning three sites

2. Noctua integrates several novel designs. The SOIR is an intermediate language to address the gap between the database operations and the back-end SMT solver for consistency verification. Other designs include an embedded approach to analyze source code with full language features and a novel decoupling encoding schema that help the SMT solver cover more semantics.
3. Evaluations show that our Noctua framework can effectively deal with real-world web applications without human intervention.

2 Background and Motivation

2.1 System Model

Figure 2 shows that a target web application consists of web servers and database daemons, where the former implements the application logic, which manipulates and visualizes the application states stored in the latter. A geo-replicated deployment of such an application spans several geographically dispersed sites, each of which runs a web server and a database replica. Database replicas form a reliable storage tier for fault tolerance via operation replication and necessary coordination. In contrast, web servers run in parallel without knowing each other. Specifically, a user request is routed to the web server in the nearest site and speculatively executed against that site's local state to determine whether the request can be accepted, according to specific consistency models. Upon acceptance, the request's side effect, i.e., state mutations, will be propagated to all remote sites and replicated across every database replica.

2.2 Fine-grained Distributed Consistency

Relaxed consistency models [7, 8, 20, 23, 24] reduce cross-replica coordination by removing unnecessary coordination from execution. PoR consistency [24] captures the notion of fine-grained consistency by restricting pairs of operations from unsafe concurrency, that is, a pair of operations P and Q are coordinated so they run in order if their concurrent execution may lead to state divergence or invariant violation. Formally, the ordering constraints of a system can be expressed as a set of pair-wise restrictions R on $U \times U$, where U is the set of operations defined within the corresponding system. In a PoR-consistent system, the local history of each site will be restricted by a partial order $O = (U, \prec)$ with the constraint: $\forall P, Q \in U, (P, Q) \in R \implies P \prec Q \vee Q \prec P$. An operation P is said to *conflict* with another Q, if the pair (P, Q) is restricted. PoR consistency is flexible enough for most web applications and allow them to strike the balance between strong and eventual consistency, depending on the choice of the restriction set. For example, if all possible pairs are restricted, the system will be strongly consistent. We assume PoR consistency in our work.

2.2.1 Checking rules Assuming PoR consistency, to understand the consistency from the source code automatically is to identify the minimal restriction set, that is, to find the minimal set of pairs of operations that break desired system properties, including state convergence and invariant preservation. We expect that (1) all replicas converge to the same state after running the same set of operations, and (2) the system state always satisfies the application's invariants.

It is straightforward to find all pairs of operations that lead to state divergence by the commutativity check over a pair (P, Q). It checks whether for all valid system state S, any instantiations of P and Q can be reordered without making the final state diverge. Denote the arguments of an operation by a vector $\vec{x}$, and the application of an operation on the state S by $S + P(\vec{x})$, and then we can formally state the *commutativity* check as follows,

$$\forall S, \vec{x}, \vec{y}, S + P(\vec{x}) + Q(\vec{y}) = S + Q(\vec{y}) + P(\vec{x}) \tag{1}$$

We say the pair (P, Q) passes the commutativity check if rule 1 holds, and fails if not.

The commutativity check alone does not guarantee safety, as the invariants must be preserved as well. Prior works [10, 16, 18] usually assume explicitly provided invariants for the sake of verification, but Rigi [41] shows that for serializable applications, the invariants are already implied by path conditions, and thus need not be specified by application developers. This is a realistic assumption, because many web frameworks, including Django, readily wrap HTTP responder functions in transactions to achieve serializability. We use the same *semantic* check, formulated as follows,

$$\text{NotInvalidate}(P, Q) \wedge \text{NotInvalidate}(Q, P) \tag{2}$$

where:

$$\text{NotInvalidate}(P, Q) = \forall S, \vec{x}, \vec{y}, g_P(\vec{x}, S) \implies g_P(\vec{x}, S + Q(\vec{y}))$$

Here, $g_P(S, \vec{x})$ is the precondition for this operation to be committed, that is, whether P called with arguments $\vec{x}$ can run to completion in the state S. Intuitively, NotInvalidate(P, Q) means that any effect of Q cannot stop P from running by invalidating P's precondition. Similarly, we say that (P, Q) passes the semantic check if rule 2 holds, and fails if not. For example, if Q deletes an object that P reads, clearly Q invalidates the precondition of P that requires the object to exist, and hence this pair fails the semantic check.

The minimal restriction set that guarantees state convergence and invariant preservation in a serializable and geo-replicated system is the set of pairs that fail either check [41].

2.2.2 State encoding.

To make SMT solvers work on the above check rules, we have to encode system states that are involved in the pre- or post-conditions of an operation, and also manipulated during operation execution and replication.

The state of a relational database relevant to consistency verification is constituted of records of tables. There are two ways to encode a database table, namely, list-based and array-based encodings.

The list-based encoding (e.g. Qex [40]) encodes a table as a list of tuples. The advantage of this encoding is that it allows all database primitives to be defined without loss of their semantics, but SMT solvers do not usually prove enough facts when being used with this kind of encoding. This is because SMT solvers have weak support for inductive reasoning [26, 33].

The array-based encoding (e.g. Rigi [41]) views a table as a mapping from primary keys to data tuples. SMT solvers usually come with a decision procedure for the array theory [11, 14], and thus this kind of encoding methods enable more facts to be derived automatically than the list-based ones. However, they discard the ordering information completely, and thus the verification of queries with the `ORDER BY` keyword is not supported within such verification systems. This ignorance can lead to unnecessary restrictions due to being conservative.

In summary, there is a dilemma between the verifiability and the coverage of supported semantics, and prior works made different trade-offs. We show that the tension can be relieved by incorporating an additional, separately maintained ordering information into the array-based encoding (see details in Section 4). This design is valid since order-related queries are relatively rare in updating operations, while the array-based encoding works well for the vast majority cases without cost for ordering information.

2.3 Modern Web Application Development

A common practice of modern web development is (1) to use a dynamic, object-oriented programming (OOP) language, like Python or PHP, and (2) to store persistent data in a geo-replicated relational database, as shown in Figure 2.

The geo-replicated database is often a database system that provides a relational data model. However, this creates an object-relational impedance mismatch [19] between the source language and the relational data model. To mitigate the mismatch, web applications commonly use high-level database abstractions, making the persistent data easy to query and update. Object-Relational Mapping (ORM for short) is one of the most popular abstractions in practice [30, 39].

```python
class User(Model):
    name = TextField(primary_key=True)
class Article(Model):
    url = TextField(unique=True)
    author = ForeignKey(User, on_delete=SET_NULL)
    title = TextField()
    content = TextField()
    created = DateTimeField(default=datetime.now)
class Comment(Model):
    user = ForeignKey(User)
    article = ForeignKey(Article)
    text = TextField()
def batch_update(request, username):
    user = User.objects.get(name=username)
    articles = Article.objects.filter(author=user)

    if request.POST['action'] == 'delete':
        articles.delete()
    elif request.POST['action'] == 'transfer':
        to_user = User.objects.get(
            name=request.POST['to_user'])
        articles.update(author=to_user)
    else:
        raise RuntimeError()
```

Figure 3. A blog application designed within Django

ORM creates an automatic, bidirectional mapping between the set of objects in the host language (like Python) and a relational database like SQL. It hides and encapsulates the SQL queries into objects and query sets, instead of SQL queries, so that the developers can directly use the objects to access the stored data.

Let's consider a multi-user blog application, where each article can only be authored by a single user. The model definitions using the Django framework are shown in Figure 3. In this system, we define two *models*, `User` and `Article`, and a *view function*, `batch_update`. A model is a special kind of classes whose instances can be persistently stored in a database. An instance of a model is called an *object*. A view function is usually an HTTP endpoint which handles HTTP requests. In this case, the view function `batch_update` either deletes all posts of a user, or transfers the authorship of them to another user, depending on the value of `POST` parameter `action`.

An object (the source object) can hold references to one or more other objects (the target objects) thanks to *relations*. A relation is defined by a specific *related key*, through which the user retrieves the related objects, similar to a pointer. Each related key creates a corresponding *reversal related key* in the target model. In Figure 3, `Article`'s `author` is a related key, and the corresponding reversal related key `article_set` is automatically created in `Article`. The articles written by a user `john` can be retrieved using `john.article_set`.

The related keys enable nested filters across multiple objects. Say we would like to find out the comments for articles authored by John, which can be expressed within Django with the following command:

```
Comment.objects.filter(
                    article__author__name="John")
```

where the first step is to look up the target user whose name is John, followed by filtering out articles authored by John (via the related key `author`), and the final step is to further filter out comments which are made to the set of selected articles (via `article`). In contrast, with SQL, one has to join the three tables, namely, users, comments, and articles.

Besides the programming convenience, the utility classes provided by ORM libraries enable expressing rich application semantics. For example, a `PositiveIntegerField` can only take values of positive integers, while the value of a `ChoiceField` should be one of a set of fixed choices. We need to understand them all since they may be relevant to consistency semantics.

2.4 ORM-enabled Consistency Analysis

The key to understanding the fine-grained consistency semantics of ORM-based applications lies in extracting the database interactions for consistency understanding. This is because only the database is geo-replicated, not the internal states of the application. For each code path, we extract the effects of the path on the database, and then they are SMT-encoded for automated verification.

The database interfaces directly supported by the verifier may not correspond to those used in realistic application code, and this mismatch can considerably complicate the process to extract database interactions. Consider the extreme case where the verifier only accepts key-value access, and apparently to analyze code that uses relational database, the code path analyzer must manually translate SQL queries into low-level `Get` and `Put` calls. Prior works make different choices of the database interfaces. Rigi [41] provides key-value-like abstractions, and ANT [15] does not support first-class query sets.

Fortunately, the widely-used ORM readily expresses a large fraction of database interaction semantics. The main purpose of ORM is to establish a bidirectional mapping between database entities and objects. Furthermore, it provides a set of query primitives, such as `filter`, which are highly flexible and composable to express complex queries. The popularity, flexibility and rich semantics of ORM open door to seize, and inspires the design of our SMT-verifiable object language SOIR as follows.

3 The SOIR Language

The goal of designing an IR for database interactions is to close the semantic gap between the source language and the SMT solver's language. This is made possible by the fact that we need only to reason about the application's effects on the geo-replicated database.

We design SOIR to model the database interaction of a code path in the original application code, motivated primarily by (1) the similarities between different ORM frameworks, and (2) the need of decoupling of analysis and verification. At the high level, SOIR is a simply-typed, imperative language designed with easy generation and efficient verification of verifiable conditions in mind.

3.1 Syntax and Semantics

The syntax of SOIR is presented in Table 1, with three primary syntactic categories, namely, types, expressions, and commands. Note that the analysis results of an application is a set of code paths encoded in SOIR, and each code path consists of three components: (1) arguments, (2) path conditions, and (3) commands. The semantics of a code path is defined as executing commands serially, making queries and changes to the replicated database during execution, until it runs to completion or aborts.

3.1.1 Types and constants. SOIR types mirror the SQL data types, with the addition of ORM abstractions, including objects Obj<μ>, query sets Set<μ> , and references Ref<μ>, where an object is a record of fields, whose types are defined in the corresponding model. A query set is an ordered set of homogeneous objects, while a reference represents a tuple corresponding to the primary key of an object. All values are immutable.

3.1.2 Expressions. SOIR expressions model local computations and database queries that do not change the replicated database state, and are constructed from the database query primitives and basic operations defined on the data types.

The basic operations include conventional ones such as arithmetic, string concatenation, as well as operations on objects and query sets. The data contained in the field f of the object o can be retrieved with `o.f`, while a new object can be constructed from a given object o with its field f transitioned to be v (a new value) via `setf(f,v,o)`. Furthermore, objects, references, and query sets can be converted between each other. For example, `singleton` turns an object into a singleton set, `deref` converts a reference to its corresponding full object, and `any` selects an arbitrary object from a set.

SOIR database queries are constructed from database query primitives. The fundamental query primitive is `all`<μ>, which is a constant query set that evaluates to the current state of the model μ, and `filter` selects a subset of a query set which matches the provided criteria. For example, the query set `articles` in Figure 3 can be stated as follows

Constants and types	Bool, Int, Float, String, Datetime	Conventional data types
	List<T>	A list of T values
	Obj<μ>	An instance of model μ
	Set<μ>	A query set for the model μ
	Ref<μ>	the ID type for μ objects
	Comparator	>, <, >=, <=, . . .
	DRelation	Relation + direction
	Order	`ascending`, `descending`
	Aggregation	`max, min, sum, cnt, avg`
Expressions and Query Primitives	**`all`**<μ: Model>(): Set<μ>	The current state of model μ.
	`singleton`<μ: Model>(obj: Obj<μ>): Set<μ>	
	`deref`<μ: Model>(ref: Ref<μ>): Obj<μ>	
	`any`<μ: Model>(qs: Set<μ>): Obj<μ>	Convert between objects, query sets, and references.
	`follow`<μ: Model, rs: List<DRelation> >(qs: Set<μ>): Set<μ>	Successively follow each relations in rs.
	`filter`<μ: Model, rs: List<DRelation>, fld: Field, op: Comparator>(val: Expr, qs: Set<μ>): Set<μ>	Find a subset $xs \subseteq qs$ such that each object $x \in xs$ satisfies that $y.fld\ op\ val$ for each y in **`follow`**<μ, rs>(**`singleton`**(x)).
	`orderby`<μ: Model, fld: Field, ord: Order>(qs: Set<μ>)	Order objects in qs.
	`aggregate`<μ: Model, ag: Aggregation, fld: Field>(qs: Set<μ>): Any	Aggregate by the field fld on qs. The true return type is determined by fld and ag.
Commands	**`guard`**($cond$: Bool)	Abort if $cond$ is false.
	`update`(qs: Set<μ>)	Merge changed objects in qs into the current state.
	`delete`(qs: Set<μ>)	Delete objects in qs from the current state.
	`link`<ρ: Relation>($from$: Obj<μ_1>, to: Obj<μ_2>)	
	`delink`<ρ: Relation>($from$: Obj<μ_1>, to: Obj<μ_2>)	Link or delink $from$ and to in relation ρ.
	`rlink`<ρ: Relation>($from$: Set<μ_1>, to: Obj<μ_2>)	Link all objects in the set $from$ with object to.
	`clearlinks`<ρ: Relation>(obj: Obj<μ>)	Remove all existing linkings in relation ρ.

Table 1. The syntax of SOIR. <...> denotes static information which does not contain client-provided runtime arguments. Some parts of the syntax are omitted for brevity.

```
filter(author=filter(name=username,all(User)),
  all(Article))
```

There are some other primitives such as `orderby`, `first`, `last` and `aggregate`. `orderby` reorders objects belonging to a query set according to the value of a specified field, `first` (or `last`) selects the object in a query set that has the smallest (or largest) order number, and `aggregate` computes on all objects in a query set for the total number of objects, or the maximum/minimum/average of a field.

3.1.3 Commands.

A command models a transition of the system state during the execution of a code path. Each command take expressions as arguments, where database queries can be made, and can possibly make changes to the underlying replicated database.

Each command has a defined semantics regarding its effect on the system state. `guard` can abort the execution if the boolean expression carried with it evaluates to false. Database queries can be made in the expression. For example, in Figure 3, the high-level expression `User.objects.get(name=username)` is translated into a `deref` expression, while the additional condition that this object exists will be translated into a `guard` command,

```
guard(exists<User>(username));
```

`delete` and `update` both take a single query set as their argument, removing objects from, or updating objects in, the current system state, regardless of their existence in the system state.

SOIR does not offer a dedicated insert command because it can be covered by `update`. To insert a new object o, the object o should be considered an additional argument of this code path, with an extra condition that it does not yet exist, and then the insertion is implemented as merging o into the current system state.

We defer the discussion of `link`, `delink`, `rlink`, `clearlinks` to the next subsection.

3.2 Relations

SOIR support relations through two primitives (`follow` and `filter`), and four commands (`link`, `delink`, `rlink`, `clearlinks`). The primitives read existing relation states, and the commands possibly change them.

A relation is represented as a set of *associations* in SOIR, where an association is a pair of a μ_1 object and a μ_2 object. Given a relation ρ from model μ_1 to μ_2, and a μ_1 query set xs, the set of objects associated with (at least one) objects in xs by ρ can be found using the expression `follow(forward,` ρ`, xs)`, or `xs.`ρ^+ for short. Given a μ_2 set ys, we can follow the relation backwards to find the the set of objects associated with objects in ys, using the expression `follow(backward,` ρ`, ys)`, or `ys.`ρ^- for short.

Similar to the flexible `filter` provided by ORM, SOIR `filter` can also be nested through the use of multiple relations. For example, the high-level expression `filter(author`$^+$`.name="John", all(Article))` finds a subset of articles which are authored by users whose name is John. This generalizes to cases where there are multiple relations, e.g. `filter(relation1`$^{[+-]}$`.relation2`$^{[+-]}$`....field=value, qs)`.

Finally, unlike query sets, SOIR does not expose relations directly, and they can only be manipulated through the four commands. Both `link` and `rlink` create new associations between objects of two related models, `delink` removes an association, and `clearlinks` removes all associates with regard to an object.

Note that SOIR does not allow finding associated objects directly of a given object, e.g. $\texttt{o}.\rho^{+}$ is not valid. Instead, SOIR demands the use of query sets, and objects must be wrapped into a query set with `singleton(o)`. This design deviates from conventional ORM abstractions, where `obj.related_key` is usually allowed, and can be an object or a query set depending on the type of the related key. SOIR lifts all relation operations to the level of query sets, and unifies many corner cases. The uniformity in handling relations simplifies the verification process.

3.3 Discussions and Limitations

The main design consideration of SOIR is to make the database semantics easy to verify. However, some features of general purpose languages, such as unrestricted looping, make the language undecidable and unverifible, and thus should be excluded from a verification language [25].

Non-supported features. There are two primary classes of features missing in SOIR:

1. Sources of unverifiability are not supported, including unrestricted loops, recursions, etc.
2. Features that conflict with the way our analysis works, including closures. Furthermore, an operation can only have finite number of arguments and commands.
3. Other features of the source language are naturally desugared during the program analysis, such as branching, user-defined functions, and user-defined data types.

Though absent from the IR, the third class is supported by Noctua through the program analysis, detailed in Section 4.1.

Implications of missing features. There are some implications from the deliberate absence of the non-supported features:

1. Due to lack of closures, some higher-level primitives such as `map` and `reduce` are not supported.
2. Because of their unverifiability or conflicts with the analysis strategy, some paradigms are impossible to represent in SOIR. Examples include updating or inserting an unbounded number of objects. This further implies that Noctua works best when batch operations are performed through query set interfaces directly, instead of iteration on query sets.

It's worth noting that some valid SOIR IR may not be supported by the verification backend. For instance, a recent version of Z3 (4.12.1 as of writing) still does not support set cardinalities, making it impossible to verify high-level expressions that contain `len(queryset)`. Other examples include aggregation that involving averages.

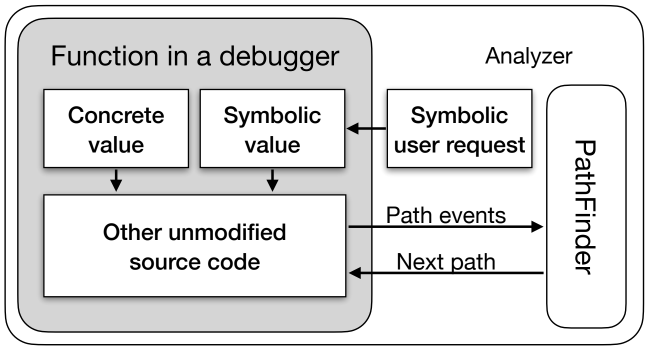

Figure 4. The architecture of ANALYZER

When faced with these limitations, we expect that the analyzer resorts to a conservative analysis to preserve soundness, that is, represent the results of unsupported high-level expressions as opaque, unknown values in the generated verification conditions, or simply restrict the code path from any possible concurrency.

4 Design Overview

Based on the IR language, we can facilitate a streamlined translation process from the source language down to the verification conditions. Both the analyzer and the verifier are simplified. In this section, we present the key ideas of the designs to address the challenges in program analysis and consistency verification.

4.1 Embedded Program Analyzer

The first step of the consistency analysis process is to analyze the source code to translate database accesses in each code path that updates system state into SOIR code. We call such code paths *effectful*. Each view function can contain zero or more than one effectful code path, and each of them will be checked in later steps independently. For example, the view function `batch_update` in Figure 3 corresponds to three code paths, only two of which (the `POST` parameter `action` is “delete” or “transfer”) are effectful, to which we refer as `BU_delete` and `BU_transfer` respectively.

However, dynamic languages are hard to analyze statically, and the difficulty is not neglible in analyzing real applications, because ORM frameworks make free use of language features, such as closures, mixins, and metaclasses. For example, Django offers a feature, *viewsets*, that construct view functions (as closures) at run-time. Even if they are analyzable, replicating language semantics is a major engineering undertaking.

To address the analysis challenge, we propose an embedded, debugger-based, framework-integrated analyzer design, taking inspiration from dynamic symbolic execution [12].

```
debugger = newDebugger();
curState = newOrderedDict();
func AnalyzeApp() {
  paths, fns = getApiEndpoints();
  debugger.initialize()
  debugger.setHookOnBranch((cond) => {
    if (cond is a concrete bool) {
      return cond;
    } else if (cond not in curState) {
      return curState[cond] = True;
    } else {
      return curState[cond];
    }
  });
  for (path, fn) in zip(paths, fns)
    AnalyzeFunc(path, fn);
}
func AnalyzeFunc(path, fn) {
  params = getQueryParams(path);
  args = makeSymbolicArgs(params);
  curState = newEmptyDict();
  do {
    debugger.run(fn, args);
    while(curState.size() > 0) {
      (condExpr, whichBranch) = curState.pop();
      if (whichBranch == True) {
        curState[cond] = False; //insert back
        break;
    }}
  } while (curState.size() > 0);
}
```

Figure 5. The pseudocode for the analysis process, whose entry is AnalyzeApp. AnalyzeFunc discovers all paths in a function, and curState stores the current branching state of the function, ie. if curState[cond] is true (or false), then cond evaluates to true (or false) in the current path. There are more paths to traverse if the length of curState is greater than 0. The debugger defines a hook onBranch, which is triggered whenever a branch is about to happen, and the return value of the hook function is considered the result of the condition expression. Non-symbolic condition expressions are evaluated eagerly.

```
class Sym {
  IR.Expr expr; Type type; IR.Expr? bool_expr;
  // ...
}
class SymInt extends Int, Sym;
class SymBool extends Bool, Sym;
class SymStr extends String, Sym;
Sym operator+(Sym a, Sym b) {
  if (a.type == Int && b.type == Int) {
    return SymInt(IR.Plus(a.expr, b.expr), Int);
  } else { /* ... */ }
}
Sym operator+(Sym a, int b) {
  return a + SymInt(IR.IntLiteral(b));
}
```

Figure 6. The pseudocode for the Sym class. Operations on Sym objects are overridden to generate IR expressions. Concrete values are converted to be symbolic when used in symbolic expressions.

As shown in Figure 4, the analyzer is implemented as a library. It is loaded into a running interpreter process, running alongside the application being analyzed, and actively collaborates with the runtime for the application configuration and other information. To collect the trace of the execution of a view function, the analyzer simulates the execution by calling the function inside a controlled debugger with a symbolic user request as the argument, where both symbolic and concrete computations (e.g. those involving constants defined in the source code) are carried out by the interpreter. During the execution, the debugger notifies the path finder of any branching event, while the path finder maintains the current path state, and controls which path to take next. The path finder guarantees that repeated calls of the function will eventually traverse all the code paths in a function, as long as there are only finite possible code paths.

This design is motivated by the inadequacy of static analysis, and inspired by concolic testing, though our usage is largely different, since our ultimate goal is not to analyze reachability of unsafe code paths, and thus we do not drop any code path and simply re-invoke the function with exactly the same symbolic arguments.

The pseudocode of the analysis process is presented in Figure 5. We will again use the blog example in Figure 3 for illustration.

Initialization. The program analysis begins with a call to AnalyzeApp, which first queries the runtime for a list of defined HTTP endpoints and their view functions, and then sets up the debugger for controlling control flow by replacing the logic of branching with a special hook, where symbolic condition expressions can be true or false depending on the current path state. Finally, Analyzer begins the collection of traces of code paths by calling AnalyzeFunc on each function.

Traverse code paths. Code paths are discovered at the point of branching, which includes not only if but also while and other constructs such as list comprehensions. Before actually running the function in the debugger, AnalyzeFunc first constructs the symbolic arguments from the HTTP endpoint. For instance, batch_update in Figure 3 requires an additional username parameter, which is usually supplied from the URL /batch_update/<username>.

During the execution of the function, all computations are carried out by the same interpreter, including symbolic ones. The execution continues, until L19 where the first branching happens. At this point, the path finder returns true because this symbolic expression is new. The interpreter then continues to L20, where the deletion happens. The analysis of the code path `BU_delete` concludes when the view function returns, and `AnalyzeFunc` adjusts the currently known path states, so that upon the second invocation, the branch at L19 will return false, leading the control flow to L21 and then through L24, corresponding to the code path `BU_transfer`. Likewise, the third invocation will go through L26.

Translate queries and record effects and arguments. To construct SOIR code from normal execution, we take advantage of the operator overloading feature commonly found in dynamic languages, through the `Sym` class in Figure 6. When `Sym` values participate in an computation, the result becomes a new `Sym` object which contains the corresponding IR code in its `expr` field. For example, in `batch_update`, the variable `user` is a symbolic object corresponding to the IR expression `deref(username)`.

To collect effects of a code path, effectful methods of query sets and objects are overridden such that they do not make actual database calls, but instead notify path finder about the events. The path finder then records these effects in the analysis results of the path.

Likewise, additional arguments of paths are also discovered with the help of the symbolic values, rather than declared, during the analysis. Whenever a new POST parameter is accessed, it is automatically recorded as an additional argument of this code path. For example, when `batch_update` accesses the "action" parameter of the POST data, a new argument `arg_POST_action` will be recorded, even though it is not part of the function argument list.

4.2 Order-aware Array-based SMT Encoding

The purpose of consistency analysis is to relax as many unnecessary restrictions as possible. However, if there are code patterns that cannot be analyzed, the verifier will resort to a conservative strategy, which could result in unnecessary false positives. Therefore, it is desirable that we can make the verifier aware of as much code semantics as possible.

To allow SMT solvers work efficiently and effectively, it is much desired to use an array-based encoding: a table is considered a mapping from object IDs to object data. This encoding, however, discards order information. In contrast, a list-based encoding can faithfully support all of the database accesses, but are ineffective due to SMT's lack of support for inductive reasoning.

Here, we propose an encoding that incorporates order information, but pay for the cost only when it's needed. The key idea of our encoding is to decouple order from object data out of the encoding of a model state, which is inspired by the observation that order information is not frequently used in web applications, that is, most operations simply retrieves some objects and update them. The encoding of a query set is presented in Table 2.

Table 2. The SMT encoding for the three model-related types. $[\tau]$ denote the corresponding SMT sort for an IR type τ.

IR type	Part	SMT encoding
μ ref		the sort for the actual type, e.g. Int
μ obj		a tuple of all data fields
	`ids`	`Set<[`μ `ref]>`
μ set	`data`	`Array<[`μ `ref], [`μ `obj]>`
	`order`	`Array<[`μ `ref], Int>`

The encoding of a model state is a tuple of three elements: (1) **ids**, a set of object IDs; (2) **data**, an array from object IDs to object data; (3) **order**, an array from object IDs to integers.

We define the ordering between objects $x \prec y$ in a query set qs via the order array of qs:

$$x \prec y \iff \text{order}[x] < \text{order}[y]$$

modulo the difference between objects and object IDs, assuming both x and y exist in the query set qs.

It is noteworthy that the actual integer order$[x]$ does not matter. This flexibility enables a straightforward encoding for `reverse` and `orderby`.

`reverse(qs)` reverses the order of objects in the query set qs. By the encoding above, the resulting order array, named order$'$, can be specified by order$'[x] = -$order$[x]$ for any ID x.

`orderby(f, ord, qs)` orders the objects in the query set qs by field f in ascending or descending order according to `ord`, which can be `asc` or `desc`. Suppose we are to order qs by f in ascending order, the resulting order array can be specified by order$'[x]$ = data$[x].f$, if the type of f is integer-like.

To merge two query sets, the order array of the result must also be specified. It is trivial when there are no new objects, since the order is not changed at all. However, the order of new objects can be ambiguous. In this case, we conservatively discard the original order information by generating an opaque, unknown order array for the result query set.

5 Implementation Details

We implement the Noctua framework in Python and integrate it with the Django framework. Table 3 summarizes the implementation costs. The design of Noctua is general, and it can also be implemented for other languages and ORM libraries. This section discusses some notable implementation choices.

5.1 Analyzer

Analyzer is responsible for discovering all code paths and translating each code path into its SOIR representation. Analyzer does the translation by simulating the execution of the

Table 3. The implementation costs of each module of the SOIR framework

Module	Lines of Python Code
Analyzer (path traversal)	795
Analyzer (Django integration)	577
Analyzer (misc.)	112
Verifier	2434

code path with an unknown, symbolic user request object on the symbolic database state. For each code path, there are three kinds of information to be collected: (1) a list of arguments, (2) a list of path conditions, and (3) a list of effects. The collection of them have been discussed in section 4.

Entry discovery. The first step is to discover all entries of user request handlers. Django and Django-Rest Framework do not require the user to list all of the entries statically, but allows dynamic construction of these request handlers by means of mixins, closures, or other dynamic features. It is impossible to find entries statically by just looking at the source code.

We address this difficulty by integrating the analyzer with the framework itself. The analyzer is implemented as a management command, installed as a separate module in the application to analyze, and the user can invoke the command to start the consistency analysis process. When the command handler is invoked, the application is already initialized, and the analyzer can query the framework for a list of all HTTP endpoints, and find their corresponding view functions.

Path discovery. Python offers a shortcut way for the `onBranch` hook through the `__bool__` magic function. Whenever a branching is about to happen, the condition expression's `__bool__` is called to determine the truth value. The `Sym` class overrides `__bool__` to communicate with to path finder to return the appropriate truth values in order to control which code path to enter.

Object existence. The existence of Django's objects can be checked using `bool(obj)`, which is implemented by `__bool__`. Unfortunately, the bool magic function is already reserved for the path exploration. We add an extra field, `bool_expr`, to the `Sym` class in Figure 6, which will be used in place of the default path condition logic. For example, the expression `user = User.objects.first()` can carries a `bool_expr` with the IR expression `not(empty(all<User>))`, and then `if user` will be handled as if it is written as `if not User-.objects.empty()`.

5.2 Verifier

Verifier is responsible for generating verification conditions recognizable by an SMT solver (such as Z3) from the IR code of a pair of operations and a specific checking rule. Verifier invokes the SMT solver to automatically decide whether the check is passed, and makes the final judgment whether the pair should be restricted from running concurrently.

Generation. The checking rules were already given in section 2.2, and to generate the verification conditions, the checking rule must be instantiated. As an example, we consider the case of the commutativity check:

$$\forall S, \vec{x}, \vec{y}, S + P(\vec{x}) + Q(\vec{y}) = S + Q(\vec{y}) + P(\vec{x})$$

To prove this fact, we do not directly generate this $\forall$-quantified formula, but transform it into an equivalent problem of finding counterexamples. That is, given some axioms, we ask the SMT solver whether there is a counterexample, and the fact is proven if none is found.

Verifier does not translate P and/or $P(\vec{x})$ as an SMT function. Instead, it computes $S + P(\vec{x})$ directly and plugs in the values.

Additionally, $+P(\vec{x})$ implies that $P(\vec{x})$ is an effect generated somewhere else, and we thus further require that $P(\vec{x})$ *can* be generated by asserting its precondition to be true on another fresh system state.

Axioms. The encoding alone (Section 4.2) allows invalid states: the set of primary keys included in object data may not match the ID set carried by the queryset itself. Therefore, each model state must satisfy the following Well-formedness axiom,

$$\forall qs, id, qs.\text{data}[id].id = id$$

An additional axiom is generated for each unique field f and the model state qs,

$$\forall x, y, qs.\text{data}[x].f = qs.\text{data}[y].f \implies x = y$$

Finally, we also require the order number to be unique, generating an axiom similar to the one above.

Unique ID optimization. Verifier can optionally generate special axioms for arguments that are marked as globally unique. For example, when $a_1, a_2, \ldots, a_n$ are all marked as such, Verifier will assert an axiom

$$\text{distinct}(a_1, a_2, \ldots, a_n).$$

This is motivated by the fact that geo-replicated databases can typically generate system-wide unique new IDs for new objects. Consider a code path that simply inserts a new record. If the unique ID feature is disabled, it will conflict with itself, because the two object IDs could be equal, failing the commutativity check and resulting in an unnecessary restriction. This optimization removes this restriction.

5.3 Soundness and Completeness

In the absence of unrestricted loops and recursions, the analysis of Noctua is sound, in that the system will be free from property-violating behaviors, such as state divergence and invariant violation; but it is incomplete in that identified restrictions can be unnecessary. This is because our path exploration does not discard any possible paths (path exploration is complete), and the semantics of each path is over-approximated. Therefore, the soundness of our analysis is the same as the previous theoretical work developed under the framework of static analysis [16, 41].

In practice, an implementation can be sound or unsound, depending on the handling of these features. Our current implementation is unsound, as it unrolls loops only for finite times. A more conservative implementation wishing to preserve soundness may opt to mark all such operations as conflicting with any other operations.

6 Evaluation

We understand the applicability, generality, and merits of the Noctua framework by answering the following questions:

- Are the analysis results correct?
- How does the program analysis scale?
- How does the consistency verification scale?
- How effective is the encoding?
- Does the analysis improve end-to-end system performance?

6.1 Experimental Setup

We evaluate our framework using four existing codebases and two synthetic benchmarks in Python. All of them use the Django framework. The basic statistics such as number of lines of code and number of code paths are shown in Table 4.

The evaluated existing open-source applications are (1) zhihu [13], a simple clone of a Quora-like Q&A site; (2) PostGraduation [38], a simple management system for postgraduates; (3) OwnPhotos [28], an open source clone of Google Photos; (4) django-todo [34], a toy todo list application. And the evaluated standard benchmarks are (1) SmallBank [32], and (2) Courseware [18]. We run these experiments on a 2019 MacBook Pro with a hexa-core Intel Core i7 CPU, and 16 GB 2400 MHz DDR4 RAM. We use Python 3.10.10, Z3 4.12.1, Django 2.2.28, and Django-RestFramework 3.13.1. The timeout of each check is 2 seconds.

6.2 Correctness

To evaluate the correctness of the results of Noctua, we re-implement standard benchmarks, SmallBank and Courseware, using the Django framework, run Noctua analysis on them, and compare the our results to the results reported by prior works that use the same checking rules, Rigi [41] and Hamsaz [18]. The baseline of Courseware is Hamsaz, and the baseline of SmallBank is Rigi.

The comparison between Noctua and the baseline tool is presented in Table 5. According to the table, we find that the results of Noctua is on par with prior tools, as expected. In fact, Noctua finds the same restriction set as the baseline tools.

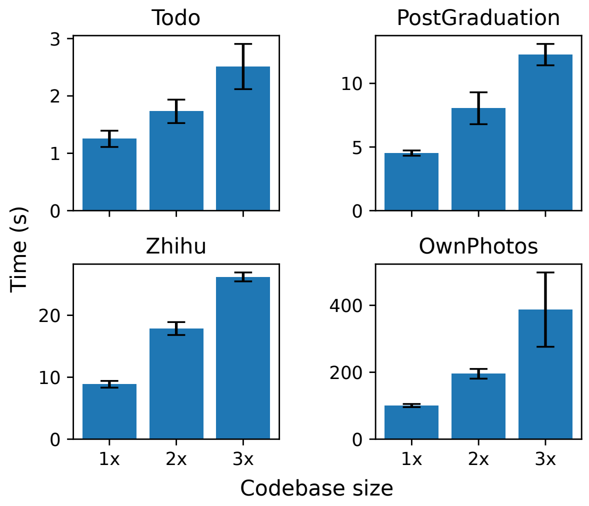

Figure 7. The analysis times of varying sizes of the codebases of the four applications

SmallBank. SmallBank is a simple banking application. It defines five operations, Amalgamate, Balance, DepositChecking, SendPayment, and TransactSavings. Each account has two types of balances, checking and savings. These operations can transfer balances between the two types, and between different accounts. Balance is a read-only operation (thus ignored), while others are effectful operations. For SmallBank, both Noctua and the baseline tool (Rigi) find 4 semantic failures, (TransactSavings,TransactSavings), (SendPayment,SendPayment), (Amalgamate,Amalgamate), and (Amalgamate,SendPayment), all of which arise from the invariant that balances must be non-negative.

Courseware. Courseware as specified by Hamsaz defines four effectful operations, Register, AddCourse, Enroll, and DeleteCourse. This application only considers the referential integrity. There are three models, Student, Course, and Enrolment. Each Enrolment is a pair of a Student and a Course. For Courseware, both Noctua and the baseline tool (Hamsaz) find a commutativity failure (AddCourse,DeleteCourse), because they can carry the same ID, and a semantic failure (Enroll,DeleteCourse), because the course can be deleted before the enrolment, breaking referential integrity.

6.3 Applicability

To evaluate the applicability of Noctua on our target applications (using dynamic languages and high-level database abstractions), we run Noctua on four real-world existing codebases. Figure 7 depicts the bar graph for the analysis times for the evaluated applications and those for the codebase doubled and tripled by repeating the same set of HTTP endpoints. We find that the analysis is sufficiently fast for real-world codebases as large as 9k lines of code, and the analysis times scale linearly with the size of the codebase

Table 4. Basic information about evaluated applications

Application	Static information				Analysis results		
	#LoC	Time (ms)	#Models	#Relations	Time (s)	#Code Paths	#Effectful Paths
Todo	434	503	1	0	1.355	18	10
PostGraduation	939	1792	8	4	8.266	40	19
Zhihu	1350	2672	14	25	8.743	51	17
OwnPhotos	3848	4081	12	46	87.880	545	120
SmallBank	346	386	1	0	0.004	17	4
Courseware	276	547	3	2	0.467	8	4

Table 5. Comparison of the analysis results produced by Noctua and those of prior works. "Com. failures" and "Sem. failures" are the number of failed commutativity checks and the number of failed semantic checks, respectively.

Application	Com. failures		Sem. failures	
	Noctua	Baseline	Noctua	Baseline
SmallBank	0	0	4	4
Courseware	1	1	1	1

Table 6. Overall verification results for the applications studied in this paper

Application	#Checks	#Restr.	#Failures	
			Com.	Sem.
Todo	55	31	28	3
PostGraduation	190	34	24	10
Zhihu	171	80	66	80
OwnPhotos	7260	3066	2572	2940

as expected, and nearly linearly with the number of code paths. The current implementation of Analyzer works best with pure Django codebases. The semantics of third-party libraries are by default handled conservatively by Analyzer, but we also added a few annotations in OwnPhotos that override the default strategy when we feel it is not good enough.

Figure 8 depicts the bar graphs for the verification times for the evaluated applications. The overall verification results of the four real-world codebases are summarized in Table 6. The verification time is quadratic in the number of verified code paths.

Finally, to evaluate the effectiveness of the order-decoupling design, we modify the verifier to not use the order information at all, and compare the verification times with and without order for PostGraduation, which does not use order-related primitives. There are no differences in the verification results, and the verification times (Figure 9).

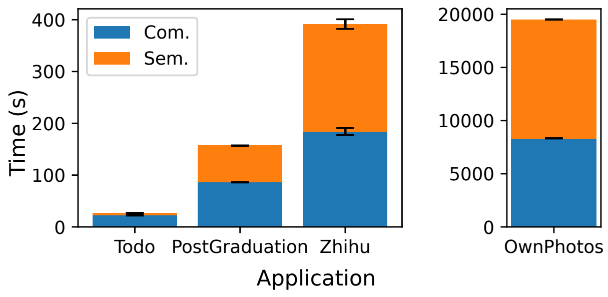

Figure 8. The verification times of the four applications

Table 7. The verification results for PostGraduation with order enabled or disabled

	Has order	No order
#Com. failures	24	24
#Sem. failures	10	10

6.4 Case Study

We manually check the results of smaller applications like zhihu and PostGraduation. We present some cases to confirm that our results are expected. In the following case studies, we will use `CreateQuestion` and `FollowQuestion` as the example operations. `CreateQuestion` creates a new `Question` object, and `FollowQuestion` subscribes a user to new activities in a question by (1) creating a new `FollowQuestion` object with all fields initialized to empty or the supplied arguments, and (2) increase the `follow` count of the question.

`CreateQuestion` *and* `CreateQuestion`. `CreateQuestion` does not conflict with itself. This is due to our assertion that new IDs are globally unique (Section 5.2), and thus this pair is trivially commutative. If we remove this assertion, `CreateQuestion` will conflict with itself due to semantic violation *and* commutativity violation, since the uniqueness of ID will be invalidated by another effect with the same ID, and they can write to the same object with different, e.g., titles.

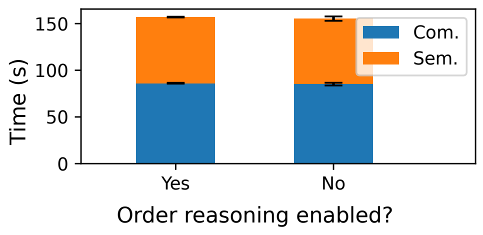

Figure 9. The verification times for PostGraduation when order is enabled or disabled. The lower boxes (blue) represent the total time for commutativity checks, and the upper boxes (orange) for semantic checks.

`CreateQuestion` ***and*** `FollowQuestion`. `FollowQuestion` conflicts with `CreateQuestion` due to the commutativity violation, as `FollowQuestion` will update the `follow` field of the referenced question object, while `CreateQuestion` will set its `follow` count to zero.

`FollowQuestion` ***and*** `FollowQuestion`. `FollowQuestion` conflicts with itself due to the semantic violation. `FollowQuestion` asserts that the pair `(user,question)` is "unique together", and thus the effect of a preceding `FollowQuestion` operation will invalidate the precondition of a later `FollowQuestion` by the same user on the same question.

6.5 End-to-End Performance

To evaluate how the analysis can be used to improve the end-to-end system performance, we deploy two applications, zhihu (ZH) and PostGraduation (PG), each as a three-node system in a local network with an extra injected latency of 1 ms for cross-node communication. For each application, we consider three different workloads depending on the percentage of operations that update system states. For example, the "15%" workload means only 15% are "writes", and the remaining 85% read-only transactions are executed locally immediately without any coordination. The baseline workload, strong consistency (SC), is implemented such that such that all requests, including read-only ones, are coordinated. Operations are initiated by sending random HTTP requests continuously for an extended period of time.

For coordination, we implement a centralized coordination service. The coordination service maintains a list of currently active operations. An operation is allowed to proceed when there are no conflicts. To simplify the implementation, we did not use the full analysis results, but only consider the HTTP endpoints and the parameters contained in the HTTP requests. We note that this is not a full implementation, as it does not consider the code path taken during the execution, but it should suffice to demonstrate the performance improvement from removing unnecessary coordination overheads.

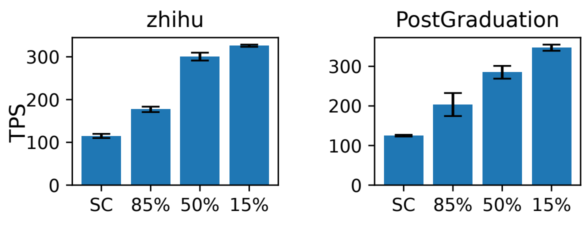

Figure 10. The throughputs of different workloads

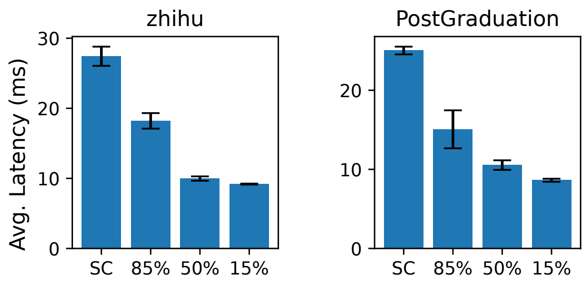

Figure 11. The average user-perceived latency

The end-to-end throughput performance is presented in Figure 10, and the average user-perceived latency in Figure 11. Compared to strong consistency, by relaxing consistency, we achieve up to 2.8x speedup in the case of ZH. As the write ratio decreases, it can be observed that the throughputs of ZH and PG both increase as expected, due to less coordination and network contention.

7 Related Work

Fine-grained consistency and its verification. Fine-grained consistency models allow consistency to be specified at a finer granularity, typically operations. RedBlue consistency [23] assigns a consistency level (strong or causal) to each operation, and allows the two levels to co-exist in the same system. Explicit consistency [9] is defined by a set of application-specific invariants, and a conforming system can reorder operations as long as no invariants are broken. The CISE consistency [16] associates tokens to operations, and guarantees that operations with conflicting tokens do not run concurrently. PoR consistency [24] expresses consistency as a set of restrictions over pairs of operations exposed by the system, and a conforming system will be coordinated such that no restricted pairs run concurrently.

Fine-grained consistency models are harder to use than their strong counterparts, since they require reasoning about possible concurrent behaviors, and there are various tools proposed for each model, but they demand non-trivial input from the developers. Indigo [9] automatically derives the Explicit consistency requirements of an application based on the provided invariants, assuming a geo-replicated key-value store. The CISE tool verifies whether an application is CISE-consistent, given the complete specifications of the application. In contrast, our work computes the weakest preconditions and the strongest postconditions with little developer assistance.

Conditions for coordination avoidance. The CALM theorem [17] formally captures the class of computations that can be coordination-free from the perspective of query results. It states that a program can run safely without coordination if and only if it is monotonic.

Safe coordination avoidance requires preserving application-specific invariants. $\mathcal{I}$-confluence [6] is a sufficient condition for invariant preservation of state-based replicated objects. It states that every execution of a set of operations is invariant-preserving if each operation in the set and the state merge operation are invariant-preserving. Well-coordination [18] is a sufficient condition for invariant preservation of operation-based replicated objects. It defines that well-coordinated executions are locally permissible, conflict-synchronizing, and dependency-preserving. Our work assumes an operation-based model, but we only consider the conflicting relationship between operations.

Domain-specific languages. Bloom [5] is built upon the CALM theorem. Bloom is a domain-specific language embedded in Ruby, and each Bloom program is restricted to a Datalog-like form. A Bloom operation can safely avoid coordination if it only contains monotonic operators.

Quelea [37] proposes a declarative programming model for eventually consistent data stores, which is manifested as a shallow embedding in Haskell. The user is expected to specify fine-grained consistency properties in a provided contract language. The specified contracts are substantially lower-level, in that they are built upon primitives about various orders.

MixT [27] is another domain-specific language, but is embedded in C++, and associates consistency information with data. It defines an information-flow-based analysis that prevents less consistent data from flowing into strongly consistent data. In contrast to domain-specific languages, our work focuses on conventional programming languages, and can be integrated into existing projects.

Program synthesizers. Hamsaz [18] is a program synthesizer which automatically synthesizes a *correct-by-construction* replicated object from user-supplied specifications based on the notion of well-coordination. Different from Hamsaz, our work focuses on the analysis and verification of existing codebases, especially ones in dynamic languages. Moreover, Hamsaz features a static analysis and protocol co-design, where our work simply adopts an existing consistency model, PoR consistency, since it is flexible enough for our use case and has been studied extensively before.

Automated program analyzers. SIEVE [22, 23] automates the choice of consistency levels in the RedBlue consistency [23]. It assumes the use of CRDT [35], requiring the users to annotate database schema with merge strategies, and thus does not validate commutativity, (operations are commutative by construction). It then computes the weakest precondition for the effect of each operation. The runtime determines at runtime with the weakest preconditions to determine whether the current effect should be strongly or causally coordinated.

AutoGR [41] automates the analysis PoR consistency by inferring a conflict table between operations using SMT solvers. The key difference is that Rigi assumes explicit and static SQL queries in source code, which are rare in practice. Our work directly works on the high-level database abstractions instead of SQL, and thus can work on more realistic applications.

ANT [15] proposes a static analysis for applications with mixed consistency on a language called OOlang. It introduces the problem of method call anticipation, that is, the conditions under which two operations can be reordered. It analyzes OOlang code and generates anticipation tables. Like MixT, ANT associates consistency information with data, and expects class fields in the source code to be annotated with `strong` (strongly-consistent) or `weak` (eventually-consistent). In contrast, our work does not support generating anticipation conditions and does not require additional user input.

8 Conclusion

We present the Noctua framework, which is, to the best of our knowledge, the first fully automated consistency analyzer that can work on realistic codebases using dynamic language and high-level database abstractions. At the core of the framework is a unified representation of the semantics of object-oriented database applications, SMT-verifiable Object Intermediate Representation. We built a fast and efficient Python program analyzer embedded in the framework that can extract application semantics from codebases as large as 9k lines of real-world Python code. To improve the coverage of verifiable semantics, we designed a novel encoding approach that decouples infrequently used primitives from the frequent ones, without paying costs when they are not required. A comprehensive evaluation confirms the correctness and the applicability of our framework.

Acknowledgements

We sincerely thank all anonymous reviewers for their insightful feedback and especially thank our shepherd Nuno Preguica for his thorough guidance in our camera-ready preparation. This work is supported in part by the National Natural Science Foundation of China under Grant No. 62141216, 62172382, and 61832011. Cheng Li is the corresponding author.

References

[1] Homepage of Django. https://www.djangoproject.com/, 2023.

[2] Homepage of Laravel. https://laravel.com/, 2023.

[3] The PHP programming language. https://www.php.net/, 2023.

[4] The Python programming language. https://www.python.org/, 2023.

[5] Peter Alvaro, Neil Conway, Joseph M Hellerstein, and William R Marczak. Consistency analysis in Bloom: a CALM and collected approach. In *Proceedings of the 5th Biennial Conference on Innovative Data Systems Research (CIDR '11)*, pages 249–260, January 2011.

[6] Peter Bailis, Alan Fekete, Michael J. Franklin, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Coordination avoidance in database systems. *Proc. VLDB Endow.*, 8(3):185–196, November 2014.

[7] Peter Bailis, Alan Fekete, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. The potential dangers of causal consistency and an explicit solution. In *Proceedings of the Third ACM Symposium on Cloud Computing (SoCC '12)*, pages 1–7, San Jose, California, 2012. ACM Press.

[8] Peter Bailis and Ali Ghodsi. Eventual consistency today: Limitations, extensions, and beyond. *Commun. ACM*, 56(5):55–63, May 2013.

[9] Valter Balegas, Sérgio Duarte, Carla Ferreira, Rodrigo Rodrigues, Nuno Preguiça, Mahsa Najafzadeh, and Marc Shapiro. Putting consistency back into eventual consistency. In *Proceedings of the Tenth European Conference on Computer Systems (EuroSys '15)*, pages 1–16, 2015.

[10] Valter Balegas, Nuno Preguiça, Sérgio Duarte, Carla Ferreira, and Rodrigo Rodrigues. Ipa: Invariant-preserving applications for weakly-consistent replicated databases. *arXiv preprint arXiv:1802.08474*, 2018.

[11] Clark Barrett, Christopher L Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV '11)*, volume 6806 of *Lecture Notes in Computer Science*, pages 171–177. Springer, 2011.

[12] Alessandro Disney Bruni, Tim Disney, and Cormac Flanagan. A peer architecture for lightweight symbolic execution. https://hoheinzollern.files.wordpress.com/2008/04/seer1.pdf, 2011.

[13] Chaoyingz. zhihu github repository. https://github.com/Chaoyingz/zhihu, February 2018.

[14] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[15] Marco Giunti, Hervé Paulino, and António Ravara. Anticipation of method execution in mixed consistency systems. In *Proceedings of the 38th ACM/SIGAPP Symposium on Applied Computing (SAC '23)*, page 1394–1401, New York, NY, USA, 2023. ACM.

[16] Alexey Gotsman, Hongseok Yang, Carla Ferreira, Mahsa Najafzadeh, and Marc Shapiro. 'Cause I'm strong enough: Reasoning about consistency choices in distributed systems. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '16)*, page 371–384, New York, NY, USA, 2016. ACM.

[17] Joseph M Hellerstein and Peter Alvaro. Keeping CALM: when distributed consistency is easy. *Commun. ACM*, 63(9):72–81, 2020.

[18] Farzin Houshmand and Mohsen Lesani. Hamsaz: Replication coordination analysis and synthesis. *Proc. ACM Program. Lang.*, 3(POPL), January 2019.

[19] Christopher Ireland, David Bowers, Michael Newton, and Kevin Waugh. A classification of object-relational impedance mismatch. In *Proceedings of the First International Confernce on Advances in Databases, Knowledge, and Data Applications (DBKDA '09)*, pages 36–43. IEEE, Mar 2009.

[20] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.

[21] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998.

[22] Cheng Li, João Leitão, Allen Clement, Nuno Preguiça, Rodrigo Rodrigues, and Viktor Vafeiadis. Automating the choice of consistency levels in replicated systems. In *Proceedings of the 2014 USENIX Annual Technical Conference (USENIX ATC '14)*, pages 281–292, 2014.

[23] Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno Preguiça, and Rodrigo Rodrigues. Making geo-replicated systems fast as possible, consistent when necessary. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI '12)*, pages 265–278, USA, October 2012. USENIX Association.

[24] Cheng Li, Nuno Preguiça, and Rodrigo Rodrigues. Fine-grained consistency for geo-replicated systems. In *Proceedings of the 2018 USENIX Annual Technical Conference (USENIX ATC '18)*, pages 359–372, Jul 2018.

[25] Mark Marron and Deepak Kapur. Comprehensive reachability refutation and witnesses generation via language and tooling co-design. Technical Report MSR-TR-2021-17, August 2021.

[26] Microsoft. Z3 will not prove inductive facts. https://microsoft.github.io/z3guide/docs/theories/Datatypes/#z3-will-not-prove-inductive-facts, 2023.

[27] Matthew Milano and Andrew C Myers. MixT: A language for mixing consistency in geodistributed transactions. *ACM SIGPLAN Notices*, 53(4):226–241, 2018.

[28] Hooram Nam. Ownphotos github repository. https://github.com/hooram/ownphotos, January 2021.

[29] Brian M Oki and Barbara H Liskov. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing (PODC '88)*, pages 8–17, 1988.

[30] Elizabeth J O'Neil. Object/relational mapping 2008: hibernate and the entity data model (edm). In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data (SIGMOD '08)*, pages 1351–1356, jun 2008.

[31] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Annual Technical Conference (USENIX ATC '14)*, pages 305–319, Jun 2014.

[32] Andy Pavlo. H-store benchmarks smallbank. https://github.com/apavlo/h-store/tree/master/src/benchmarks/edu/brown/benchmark/smallbank/, July 2013.

[33] Andrew Reynolds and Viktor Kuncak. Induction for smt solvers. In *Proceedings of the 16th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI '15)*, pages 80–98. Springer, 2015.

[34] Shrey Shah. django-todo github repository. https://github.com/shreys7/django-todo, November 2022.

[35] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. *A comprehensive study of convergent and commutative replicated data types.* PhD thesis, Inria–Centre Paris-Rocquencourt; INRIA, 2011.

[36] Jeff Shute, Radek Vingralek, Bart Samwel, Ben Handy, Chad Whipkey, Eric Rollins, Mircea Oancea, Kyle Littlefield, David Menestrina, Stephan Ellner, John Cieslewicz, Ian Rae, Traian Stancescu, and Himani Apte. F1: A distributed sql database that scales. *Proc. VLDB Endow.*, 6(11):1068–1079, Aug 2013.

[37] Kc Sivaramakrishnan, Gowtham Kaki, and Suresh Jagannathan. Declarative programming over eventually consistent data stores. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15)*, pages 413–424, Portland OR USA, June 2015. ACM.

[38] Amine Smahi. Postgraduation github repository. https://github.com/Amine-Smahi/PostGraduation, October 2019.

[39] Alexandre Torres, Renata Galante, Marcelo S Pimenta, and Alexandre Jonatan B Martins. Twenty years of object-relational mapping: A survey on patterns, solutions, and their implications on application design. *Information and Software Technology*, 82:1–18, 2017.

[40] Margus Veanes, Nikolai Tillmann, and Jonathan De Halleux. Qex: Symbolic sql query explorer. In *Proceedings of the 16th International Conference on Logic Programming and Automated Reasoning (LPAR '10)*.

Springer, Apr 2010.

[41] Jiawei Wang, Cheng Li, Kai Ma, Jingze Huo, Feng Yan, Xinyu Feng, and Yinlong Xu. Autogr: Automated geo-replication with fast system performance and preserved application semantics. *Proc. VLDB Endow.*, 14(9):1517–1530, May 2021.