DIFFBASE: A Differential Factbase for Effective Software Evolution Management

Xiuheng Wu
Nanyang Technological University
Singapore
xiuheng001@e.ntu.edu.sg

Chenguang Zhu
The University of Texas at Austin
USA
cgzhu@utexas.edu

Yi Li
Nanyang Technological University
Singapore
yi_li@ntu.edu.sg

ABSTRACT

Numerous tools and techniques have been developed to extract and analyze information from software development artifacts. Yet, there is a lack of effective method to process, store, and exchange information among different analyses. In this paper, we propose differential factbase, a uniform exchangeable representation supporting efficient querying and manipulation, based on the existing concept of program facts. We consider program changes as first-class objects, which establish links between intra-version facts of single program snapshots and provide insights on how certain artifacts evolve over time via inter-version facts. We implement a series of differential fact extractors supporting different programming languages and platforms, and demonstrate with usage scenarios the benefits of adopting differential facts in supporting software evolution management.

ACM Reference Format:
Xiuheng Wu, Chenguang Zhu, and Yi Li. 2021. DIFFBASE: A Differential Factbase for Effective Software Evolution Management. In *Proceedings of the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '21), August 23–27, 2021, Athens, Greece.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3468264.3468605

1 INTRODUCTION

Real software is seldom created "all at once" and changes are inevitable [60]. Software developers often take advantage of the knowledge and insights they gain over time to repair, enhance and optimize earlier versions of the system through incremental updates. The software artifacts accumulated during the development process have become crucial resources for understanding and analyzing software systems from an evolutionary point of view. These include not only the source code written, but also the "by-products" produced such as code change histories, log messages, and test run results. Many tools and techniques have been developed to harvest useful information and use it to support different software evolution management tasks. For example, trends and patterns observed in the past changes can be used to estimate quality of new changes [42], characterize architectural evolution [64, 73], optimize development process [11], identify high-level software functionalities [47, 48], and discover project-specific API usage patterns [54] as well as developer expertise [16, 58].

These evolution management tasks require not only a clear understanding about each program version, but also insights about the longitudinal evolution of program elements introduced by incremental changes. Such understanding and insights are now being produced by different analysis techniques, and can potentially be shared and reused. But there still does not exist an effective way to process, store and exchange information among different analyses. Software Configuration Management systems (SCMs), such as Git [26] and SVN [61], are widely used in the development practices, where incremental changes are manually grouped by developers to form *commits* (a.k.a. *change sets*). Yet, the main goal of SCMs is to support development activities, e.g., recording, examining, and reverting changes, rather than analysis tasks, e.g., inferring high-level program properties from changes and establishing relationship among individual changes. The information embedded in change sets goes beyond lines added and removed. When combined with the understanding of language syntax and semantics, much richer information can be obtained and analyzed about the evolution of software.

Inspired by the *program fact extraction* techniques [1, 2, 7, 33] which generate facts about a single version of the software artifacts, we propose *differential facts*, a uniform representation of software changes consolidating relevant information across multiple versions of the same artifacts. Program facts can be any desired information about the software artifacts—*structural* relations such as "method `A` is contained in class `C`" and *semantic* relations such as "method `A` is called by another method `B`", all could be considered facts. Differential facts go beyond just a single program version, and consider program changes, which highlight differences and linkage between multiple versions, as first-class objects. Fig. 1 is an illustration of the meta-model of differential facts, which demonstrate only a subset of the possible fact types. These include the *intra-version facts* capturing the containment, calling, and referencing relations between code entities at the same version; there are also the *inter-version facts* which include the code-level insertion, deletion, and update changes between the old and new versions as well as the dependencies [48] between commits.

The inter-version facts are tightly connected to the intra-version facts through common entity nodes, making it possible to reason about facts across multiple versions. For example, we represent various types of differential facts in a common exchangeable format and store them in a *differential factbase*. Then by querying the factbase, we could answer questions such as "what are the functions whose bodies get changed but the signatures stay constant in an upgrade"

ESEC/FSE '21, August 23–27, 2021, Athens, Greece

ACM ISBN 978-1-4503-8562-6/21/08…$15.00
https://doi.org/10.1145/3468264.3468605

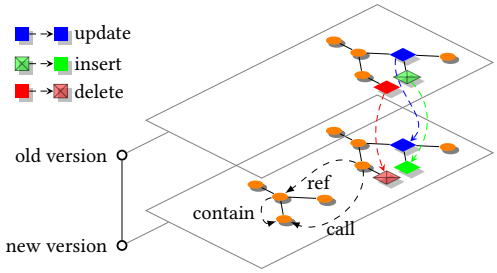


Figure 1: An example meta-model of differential facts.

and "what are the tests potentially be affected by this upgrade". The differential factbase supports standard Tarski relational algebra [62] operations such as union, intersection, composition, and transitive closure. Existing languages and tools, such as JGrok [65], can be used to query and manipulate facts. To support more complex analysis tasks and ease the implementation of analysis scripts, Datalog is also supported so that one can also run inference on differential factbase using Datalog engines such as Soufflé [37].

Using differential facts to manage evolving software artifacts has several benefits. First, the factbase serves as an abstraction of the artifacts and their change histories. It hides all the complex details of different programming languages as well as change representations and provides a unified queryable interface for downstream analyses. Second, the differential facts produced during offline fact extraction can be shared and reused by many online analyses, thus saving overall computation resources. Furthermore, with the versioning information encoded, differential facts support the lifting of analysis tasks from a single version to multiple versions, and again improves analysis efficiency. Finally, the fact extractors are relatively lightweight and built on top of existing tool chains. For instance, facts can be stored in any relational database for persistent storage and efficient processing.

We re-implemented several software maintenance tasks with Diffbase, demonstrating its advantages in efficiency and interoperability. These include a 44% time cost reduction in semantic history slicing, similar precision without safety violations compared with the state-of-the-art static regression test selection tools, and about 80% time and space usage reduction in lifted pointer analysis.

Contributions. In this paper, we make the following contributions.

- We present differential facts, a uniform exchangeable representation of incremental software changes, supporting efficient querying, manipulation, and reuse.
- We implement several differential fact extractors targeting different fact types, programming languages, and platforms, including inter-version fact extractors for atomic changes and commit dependencies, intra-version fact extractors for both static dependencies and code coverage information. All language-dependent extractors are implemented for both C/C++ and Java. The prototype tools are open-source and additional results are available at: https://d-fact.github.io.
- We apply differential facts in three evolution management tasks, namely *semantic history slicing* [48], *change impact analysis* [5], and *regression test selection* [67].

$$\frac{y \in V(r)}{V(r') \leftarrow V(r) \cup \{x\} \quad \textsc{Parent}(x) \leftarrow y \quad id(x) \leftarrow n \quad v(x) \leftarrow v} \textsc{Ins}((x,n,v),y) \qquad \frac{x \in V(r)}{V(r') \leftarrow V(r) \setminus \{x\}} \textsc{Del}(x) \qquad \frac{x \in V(r)}{v(x) \leftarrow v} \textsc{Upd}(x,v)$$

Figure 2: Types of atomic changes [23].

- We evaluate our approach on real open-source software projects and demonstrate the benefits of adopting differential facts over existing techniques.

2 BACKGROUND

This section introduces the necessary background and terminology.

2.1 Program Change Histories

A valid program p can be parsed as an *abstract syntax tree* (AST), denoted by $\textsc{Ast}(p)$. Formally, $r = \textsc{Ast}(p)$ is a rooted tree with a set of nodes $V(r)$. The root of r is denoted by $\textsc{Root}(r)$ which represents the compilation unit, i.e., the program p.

Each entity node x has an identifier and a value, denoted by $id(x)$ and $v(x)$, respectively. In a valid AST, the identifier for each node is unique (e.g., fully qualified names in Java) and the values are canonical textual representations of the corresponding entities. We denote the parent of a node x by $\textsc{Parent}(x)$. The children are unordered—the ordering of child nodes is insignificant. Therefore, each program has its unique AST representation.

Let Γ be the set of all ASTs. Now we define changes, change sets and change histories as AST transformation operations.

Definition 1 (Atomic Change [48]). *An atomic change operation $\delta : \Gamma \nrightarrow \Gamma$ is a partial function which transforms $r \in \Gamma$ producing a new AST r' such that $r' = \delta(r)$. It can be either an* insert, delete *or* update *(see Fig. 2).*

An insertion $\textsc{Ins}((x,n,v),y)$ inserts a node x with identifier n and value v as a child of node y. A deletion $\textsc{Del}(x)$ removes node x from the AST. An update $\textsc{Upd}(x,v)$ replaces the value of node x with v. A change operation is *applicable* on an AST if its preconditions are met. For example, the insertion $\textsc{Ins}((x,n,v),y)$ is applicable on r if and only if $y \in V(r)$. Insertion of an existing node is treated the same as an update.

Definition 2 (Change Set). *Let r and r' be two ASTs. A change set $\Delta : \Gamma \nrightarrow \Gamma$ is a sequence of atomic changes $\langle \delta_1, \ldots, \delta_n \rangle$ such that $\Delta(r) = (\delta_n \circ \cdots \circ \delta_1)(r) = r'$, where $\circ$ is standard function composition.*

A change set $\Delta = \Delta_{-1} \circ \delta_1$ is applicable to r if δ_1 is applicable to r and Δ_{-1} is applicable to $\delta_1(r)$. Change sets between two ASTs can be computed by tree differencing algorithms [12].

Definition 3 (Change History). *A history of changes is a sequence of change sets, i.e., $H = \langle \Delta_1, \ldots, \Delta_k \rangle$.*

2.2 Program Facts as Typed Graphs

Querying and analyzing program facts (relations) requires a specialized data structure and a set of operators to manipulate the facts. Now we define *typed graph* (also known as edge-colored graph), which is a graph with a fixed number of *edge types*.

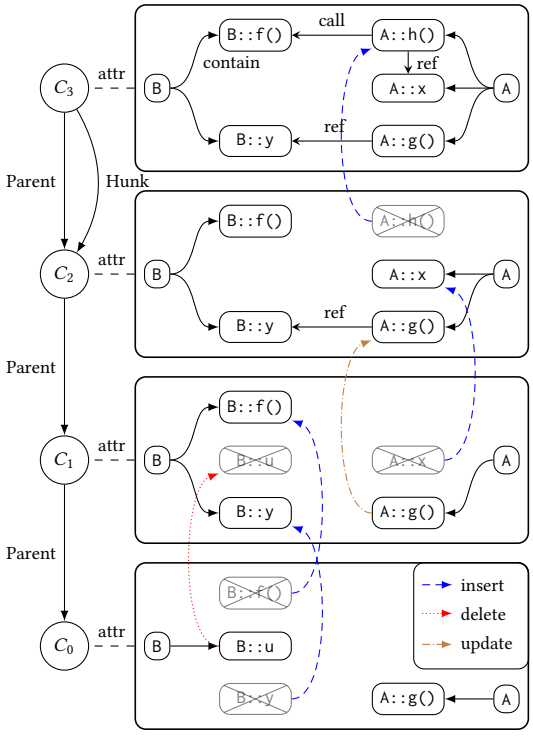


Figure 3: A typed graph illustrating source code and changes between four consecutive versions.

DEFINITION 4 (TYPED GRAPH [33]). *Let T be a non-empty set of types. We say that a directed graph $G = (V, E, T)$ is a typed graph with vertex-set V and edge-set $E \subseteq V \times V \times T$.*

We denote edges $(v_1, v_2, \tau) \in E$ by $(v_1, v_2) : \tau$. A *homogeneous* typed graph G of type τ is a typed graph $G = (V, E, \{\tau\})$ whose edges are all the same type τ. Graphs in the classic graph theory are equivalent to homogeneous typed graphs and thus just special cases of typed graphs.

DEFINITION 5 (TYPED SUB-GRAPH). *Let $G_1 = (V_1, E_1, T_1)$ be a typed graph. $G_2 = (V_2, E_2, T_2)$ is a typed sub-graph of G_1, denoted by $G_2 \subseteq G_1$, if and only if $V_2 \subseteq V_1$, $E_2 \subseteq E_1$, and $T_2 \subseteq T_1$.*

DEFINITION 6 (TYPE-INDUCED SUB-GRAPH). *Let $G = (V, E, T)$ be a typed graph and $T' \subseteq T$. We define the type-induced sub-graph of G, denoted by $G[T']$, as the typed sub-graph (V', E', T') of G such that $E' = \{(v_1, v_2, \tau) | \tau \in T'\}$ and $V' = \{v | (v, \_, \_) \in E' \text{ or } (\_, v, \_) \in E'\}$.*

Essentially, each type of facts is represented using a separate edge type (details discussed in Sec. 3.1). For instance, a simple program with four versions are shown as a typed graph in Fig. 3. (More details, including the textual differences and associated facts, are provided in the supplemental materials). Each of the four rectangles indicates a version sub-graph, where solid arrows connecting code entities within rectangles are edges representing static dependencies and dashed arrows connecting code entities across different rectangles are edges representing atomic changes. For example, in the sub-graph associated with C_3, `A::h()` *calls* `B::f()` and *references* field `A::x`. Member methods and fields are all *contained* within corresponding classes. The circles positioned at the left of the rectangles are commit entities connected by commit dependency edges. In this example, the commit history is linear, with C_0 being the oldest ancestor and C_3 *hunk depends* on C_2 (cf. Sec. 3.1).

```
a := new A();
b := a;
```

```
PointTo(a, o1)
Assign(a, b)
```

Figure 4: Example source code and its corresponding facts.

Different sub-graphs can interact by sharing common nodes. For example, `A::x` is on edges of "insert", "contain", and "ref" types. Nodes and edges can have attributes too. For instance, the commit nodes C_i in Fig. 3 are connected to each rectangle as attributes, indicating the versions those program facts hold. There could also be other purely informational attributes, such as source code locations.

Algebraic Operators. The list of operators for manipulating typed graph follow the classic Tarski's relational calculus [62], which include *identity* (E^0), *inverse* (E^{-1}), basic set operations: *union* ($E_1 \cup E_2$), *intersection* ($E_1 \cap E_2$), *substraction* ($E_1 \setminus E_2$), composition($E_1 ; E_2$), transitive closure (E^+), and reflexive transitive closure (E^*). Inspired by projection operator used in JGrok [65], we use $\circ$ to denote this variant of composition operation, which takes a set and a relation: $V \circ E = \{v_2 | v_1 \in V, (v_1, v_2) \in E\}$, $E \circ V = \{v_1 | v_2 \in V, (v_1, v_2) \in E\}$. For producing a set (vertices) from a relation (edges), selecting columns by index is represented as $E[i]$ and i starts from 1.

2.3 Datalog

To provide inference capability on top of program facts, i.e., the ability to generate new facts based existing ones, we may choose to represent typed graphs using Datalog, which is a declarative logic programming language with syntax similar to that of Prolog. Datalog extends relational calculus with recursion and can be manipulated according to inference rules instead of low-level relational algebra operations, thus improving both usability and expressiveness.

Two constructs in Datalog are *facts* and *rules*. Rules are defined as Horn clauses of predicates, usually written in the following form.

$$r_0(X_1, X_2, ..., X_k) \text{ :- } r_1(Y_1, Y_2, ..., Y_s), \ldots, r_n(Z_1, Z_2, ..., Z_t).$$

where r_i is a predicate and the arguments, X_i, Y_i, Z_i, are variables or constants. Considering the existing ambiguity in Datalog definitions, the following concepts used in this paper are defined.

- $r_1(a_1, \ldots, a_s)$ is a fact if all of its arguments $(a_1, \ldots, a_s)$ are constants. As shown in Fig. 4, `a`, `b`, `o1` on the right are constants representing the concrete source elements on the left (`o1` is for the object allocated by `new A()`).
- Predicates defined a priori by facts can only appear on the right hand side of rules, which form schema of the *extensional database* or EDB and those predicates are called EDB predicates. EDB is defined as the set of facts of EDB predicates.
- Predicates appeared on the left are defined by rules, which form schema of *intensional database* or IDB and those predicates are called IDB predicates. r_0 is an IDB predicate. IDB is defined as the combination of a set of rules and facts of IDB predicates.

A Datalog engine consumes EDB and IDB, then produce instances of IDB predicates according to the rules in IDB. Program facts

```
0  SCHEME TUPLE :
1  Upd Prog Prog
2  Del Prog Prog
3  Ins Prog Prog
4
5  Contain Prog Prog
6  Call  Prog Prog
7  Ref Prog Prog
8
9  Coverage Test Prog
10 Hunk Commit Commit
11 Parent Commit Commit
12
13 SCHEME ATTRIBUTE :
14 (Upd) {commit}
15 (Del) {commit}
16 (Ins) {commit}
```

(a) General fact schema.

```
0  SCHEME TUPLE :
1  $INHERIT jProg Prog
2  $INHERIT jClass jProg
3  $INHERIT jFunc jProg
4  $INHERIT jVar jProg
5  $INHERIT jCtor jFunc
6  $INHERIT jMethod jFunc
7  $INHERIT jStaticMethod jMethod
8  $INHERIT jEnum jClass
9
10 Contain jProg jProg
11 Call jFunc jFunc
12 Ref jProg jProg
13 Inherit jClass jClass
```

(b) Java extended schema.

Figure 5: Fact schema written in TA language.

can be stored as Datalog facts, with each type-induced sub-graph representing facts of one predicate.

Example. In a simplified Anderson-style pointer analysis [4], an assignment operation $y := x$ marks all objects pointed-to by x also pointed-to by y, which is expressed by the following Datalog rule:

```
PointTo(y,z) :- PointTo(x,z), Assign(x,y).
```

Here, `PointTo` and `Assign` are IDB and EDB predicates, respectively. Following the rule above and the two facts in Fig. 4, a Datalog engine can produce all facts of IDB predicates, i.e., `PointTo(a,o1)` and `PointTo(b,o1)`.

3 DIFFERENTIAL FACTS

In this section, we first present the *general schema framework* for differential facts which applies to most of the languages and platforms. We then describe the architecture of Diffbase with various differential fact extractors and support for fact reusing in specific tasks. Finally, we describe how analysis lifting is made possible by the application of differential facts.

3.1 General Schema Framework

We construct the differential factbase as a typed graph with typed nodes. This is to differentiate various types of entities in program facts. Let the set of all node types be T_v. Then the fact schema is described by, $M \subseteq T_v \times T_v \times T$.

Similar to the relation declarations in Datalog and the fact scheme in the TA language [32], the *fact schema* describes the meta-model of a differential factbase. What follow are the *fact instances*, which represent the edges of the graph, following the rules described by the schema. The rules can be explicitly written as, $\forall(v_1, v_2, \tau) \in E \cdot (t(v_1), t(v_2), \tau) \in M$, where $t(v)$ is the type of the node v.

Fig. 5a shows the general schema for differential facts, written in TA language. The general schema only defines the core entities and generic entity relations, which are not specific to any programming language or version control system. The entity types, *Prog*, *Commit*, and *Test* (Lines 1–3) are defined to represent AST nodes, commit objects, and test cases, respectively. Based on these entity types, we define four classes of relations, namely, the *static dependency*, *coverage*, *atomic changes*, and *commit dependency*.

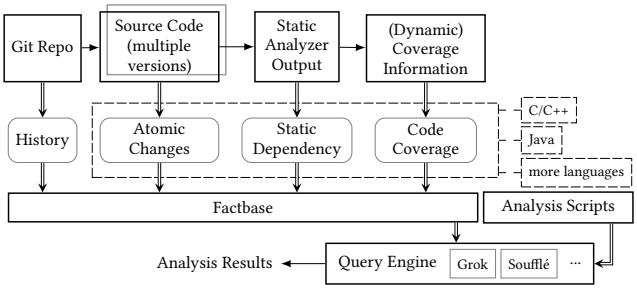


Figure 6: An overview of Diffbase architecture.

Let G be the typed graph containing all differential facts. Each class of relations can be viewed as a type-induced sub-graph,

$$G[T_s] = G[\{\text{Call}, \text{Ref}, \text{Contain}\}] \quad \text{(static dependency)}$$
$$G[T_c] = G[\{\text{Cov}\}] \quad \text{(coverage)}$$
$$G[T_d] = G[\{\text{Ins}, \text{Upd}, \text{Del}\}] \quad \text{(atomic changes)}$$
$$G[T_h] = G[\{\text{Hunk}, \text{Parent}\}] \quad \text{(commit dependency)}$$

$G[T_s]$ captures the dependency relations between *Prog*, namely, Call, Ref, and Contain, with their standard semantics. $G[T_c]$ captures the coverage relations between *Test* and *Prog*, denoted by Cov, and the relation holds when a test run covers the given code entity. $G[T_d]$ captures three types of atomic changes, namely, Ins, Upd, and Del.[1] Finally, $G[T_h]$ captures two types of dependencies between commits, namely the *hunk dependency* [48] and the *history dependency*, denoted by Hunk and Parent respectively. For instance, there are history dependencies from a parent commit to all its children. Among those sub-graphs, commit dependencies and atomic changes lead to *inter-version* facts, while facts describing static dependencies and coverage information are *intra-version*. For intra-version facts, we designate the commit to which an atomic change belongs, as an attribute of the corresponding fact.

The general schema can be extended for specific types of artifacts by adding new schema rules. Fig. 5b shows a simplified specific extension for Java. They key extension is a hierarchy of language-specific types (Lines 1–8). With those language-specific types, the typing information in relations can be refined (Lines 10–12) and new relations can be added (Line 13). For example, parameters of Call are now *jFunc* instead of the generic type *Prog*; and Inherit is added to represent the inheritance relations between *jClass* types.

Since facts we use are low-level, any structured data with a well-defined schema can be encoded and extended similarly. Examples include architecture diagram, E-R diagram, and other UML models. In this paper, we focus on facts extracted from source code and commit histories.

3.2 Extraction of Differential Facts

The overall architecture of Diffbase's fact extraction and query engine is shown in Fig. 6. We implement Diffbase as a general framework supporting different tasks with the help of multiple fact extractors and analysis scripts. In practice, Diffbase can be integrated with IDE and version control tools or services (e.g., GitHub), accumulating facts incrementally with the evolution of software.

[1] For readability and simplicity, edge types will be used to represent edge sets of corresponding types when there is no ambiguity, i.e., Ins means E[Ins].

The rectangles at the top represent data sources, including Git repositories, source code versions, and coverage information obtained from test runs. These are either raw inputs or easily derived by running existing tools such as compilers. The rounded rectangles below represent basic facts, extracted from corresponding data sources, where the fact extraction processes are represented by incoming double arrows. The *factbase* is where the basic facts are organized and stored, and *analysis scripts* are executed to derive the *analysis results* in the form of new facts.

To perform an empirical evaluation on the various applications of differential facts, we implemented several fact extractors supporting different programming languages and platforms. (1) We developed inter-version fact extractors for C/C++ and Java. We replaced the C/C++ back-end of the AST differencing tool, GumTree [21], with our own AST emitter based on Clang [13] to ensure maximal compatibility. The Java version is implemented based on ChangeDistiller [24]. (2) We developed extractors for test coverage facts based on existing code coverage tools—Gcov [25] for C/C++ and JaCoCo Code Coverage Library [35] for Java. (3) We used ClangEx [3] and Apache BCEL [6] to extract intra-version static dependency facts, for C/C++ and Java, respectively. (4) We developed a language-neutral extractor for commit dependencies based on CSLICER [48]. (5) We built a tool for merging version annotated facts to support lifted analyses. The implementation of these fact extractors is relatively simple thanks to the existing open source tools. The support of new fact types and programming languages in the future is also straightforward.

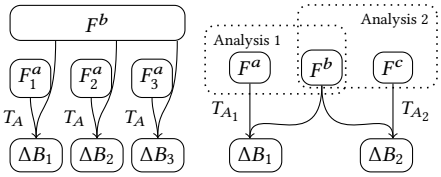
Facts are mainly stored in the format of Tuple-Attribute files. For example, intra-version dependency facts of each version are stored in a separate text file and each line represents a "depends on" relation between two program entities. JGrok [65] is used to query facts in TA language. Facts in the lifting experiments are represented in Datalog, where facts for each relation is stored in a separate file. The Datalog engine we used is Soufflé [37].

3.3 Reusing of Differential Facts

Differential factbase not only supports storage, exchange, and manipulation of facts, but also enables reusing of facts in different ways. Many complex software analysis tasks require information from various sources, which may be used more than once in subsequent analyses. This creates opportunities for performance improvement if repeatedly used information is persisted and reused. However, intermediate analysis results are often not explicitly exposed or represented in non-standard formats, which makes reusing challenging when performing new tasks.

We propose to separate the *creation* and the *usage* of facts: DIFFBASE serves as an intermediate layer providing uniformed data format to the downstream analysis scripts. This enables efficient and flexible reuse of both the data and the analysis logic. We first define necessary terminology, before illustrating the possible venues of reusing.

DEFINITION 7. *(Analysis Task). Let F be a set of facts and A be an analysis script with a set of operations. We define $B = T_A(F)$ as an analysis task taking F as the input and producing B as the results, where $F \subseteq B$.*



(a) Cross-run **(b) Cross-analysis**

Figure 7: Fact reusing between sub-tasks.

An analysis task grows the set of known facts and it can comprise several sub-tasks. Some sub-tasks depend on others, while independent sub-tasks can be executed in parallel. For simplicity, a task is represented as a linearized composition of sub-tasks.

DEFINITION 8. *(Analysis Sub-Task). Let $\langle T_{A_1}, \ldots, T_{A_n} \rangle$ be a sequence of tasks. We say $\{B_{i+1} = T_{A_i}(F_i) \cup B_i \mid i \in [1, n]\}$ are sub-tasks of $T_A(F)$ if and only if $T_A(F) = T_{A_n} \circ \cdots \circ T_{A_1}(F)$.*

With the completion of more sub-tasks, the set of facts available for consumption, i.e., B_i, is also growing. The amount of growth is defined as $\Delta B_{i+1} = B_{i+1} - B_i$. The set of facts consumed by a sub-task is only a subset of the facts available at the moment, i.e., $F_i \subseteq B_i$.

There are various scenarios where reuse can happen with the support of DIFFBASE. We first discuss the cases where facts can be shared among multiple analysis tasks. These include the *cross-run* and the *cross-analysis* fact reusing. Note that this is not a unique feature for differential facts and applies for other program fact-based approaches as well.

Cross-Run Fact Reusing. One common scenario for fact reusing is when the same analysis task is repeatedly performed with only part of the input data varying every time. For example, in Fig. 7a, an analysis task T_A is executed three times, but on different inputs sharing common facts, where $F_i = F_i^a \cup F^b$. Here only F_i^a needs to be created across different runs and F^b can be reused. A real-world example is *Semantic history slicing* [46, 48, 72] and an in-depth analysis of this scenario can be found in Secs. 4.1 and 4.2.

Cross-Analysis Fact Reusing. Similar to the cross-run reusing, two different analysis tasks can share information through persistent facts. As shown in Fig. 7b, two tasks, T_{A_1} and T_{A_2}, require $\{F^a, F^b\}$ and $\{F^b, F^c\}$ as inputs, respectively. Because they both require F^b as a part of their inputs, F^b needs only be created once, but can be used many times subsequently.

If we were to perform both history slicing (T_{A_1}) and regression test selection [20] (T_{A_2}) on the same software project within the same history range, then we would be able to reuse differential facts across the two analyses. In particular, intra-version facts including the test dependencies and inter-version facts (i.e., change information) can be reused. Experimental results and more detailed discussions on cross-analysis fact reusing can be found in Sec. 4.3.

Analysis Script Reusing. Apart from reusing facts, DIFFBASE also enables the reusing of analyses. Analysis scripts implementing inference logic can be reused on different input data. The *analysis script reusing* can be viewed as a special case for cross-run reusing. When the same analysis task T_A is executed multiple times on disjoint input facts, not being able to reuse facts, we may still reuse

Input: annotated EDB facts E and IDB facts I, Datalog program P
Output: updated IDB facts I

```
1  repeat
2      fixpoint ← true;
3      foreach rule R = (r_0 :- r_1, ..., r_n) ∈ P do
4          foreach ⟨(f_1, v_1), ..., (f_n, v_n)⟩ ∈ E ∪ I, r_i(f_i) is true do
5              f_0 ← R(f_1, ..., f_n);
6              v_g ← ⋂_i v_i;
7              if v_g ← ∅ then continue;
8              if ∃(f_0, v_0) ∈ I then
9                  if v_g ≠ v_0 then
10                     fixpoint ← false;
11                     Update I: (f_0, v_0) ← (f_0, v_g ∪ v_0);
12             else
13                 fixpoint ← false;
14                 I ← I ∪ (f_0, v_g);
15 until fixpoint;
```

Algorithm 1: Lifted Datalog inference algorithm ($\hat{\mathcal{I}}$) on differential facts.

the analysis script A. This is made possible with the hierarchical design of our schema framework, which is general enough to be compatible with different types of input facts. For example, the core analysis script needs only written once using the basic schema, and specific extensions can be added accordingly. We demonstrate this with a case study on history slicing (Sec. 4.1.1), where the fact usage becomes language- and platform-agnostic.

3.4 Analysis Lifting on Differential Facts

The differential fact schema presented so far represents extracted facts as a typed graph and supports querying using relational algebra as defined in Sec. 2.2. This is sufficient to support many rudimentary analysis tasks concerning about only the structural information represented by the graph. Examples include history slicing, change impact analysis, and regression test selection, which will be discussed in detail in Sec. 4.1.

More importantly, we show that, when combined with inference rules, differential facts can support *analysis lifting* [59], which greatly improves the efficiency. Analysis lifting is the process of adapting a single-version analysis task to a *lifted analysis* which works on multiple versions simultaneously. For example, a *points-to* analysis, which determines the set of objects potentially pointed to by a program expression, can be lifted to multiple versions of the same program. Instead of applying a brute-force approach, where each version is analyzed separately, the lifted points-to analysis would take the differential facts of n versions and produces the expected results for all versions. Analysis lifting creates new opportunities for reusing of the intermediate analysis results, which will be discussed more in Line 15.

The lifting of analyses largely depends on the version annotated facts. Inspired by Shahin et al.'s lifting algorithm [59] on software product lines, version annotated facts have an additional version tag after usual fact terms. The tag is a sequence of version strings, following an "@" symbol and separated by commas, indicating the set of versions on which the facts hold. For example, "`fn-a fn-b @v1,v2`" in an input file `Call.facts` means that the fact, `fn-a` calling `fn-b`, holds on both `v1` and `v2`.

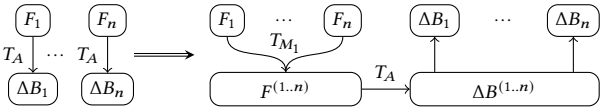


Figure 8: Analysis reusing enabled by lifting.

Algorithm 1 shows the modified Datalog inference algorithm to properly handle version annotated EDB facts and generate IDB facts with correct annotations. It takes basic facts (E) generated from fact extractors and Datalog programs (P) which specify how to generate new facts from existing ones as input. The basic facts are tagged with versions and merged into a compact representation. The algorithm first sets the *fixpoint* variable to true and iterates over the rules from P (Lines 3 to 14). The condition "$r_i(f_i)$ *is True*" on Line 4 indicates that f_i is a fact on the predicate r_i. For each set of existing facts in EDB which satisfies the predicates on the right hand side of the rule, i.e., $\langle (f_1, v_1), \ldots, (f_n, v_n)\rangle$, the annotated version labels are intersected with the existing ones (Line 6) and a new fact f_0 is generated by applying the rule. (On Line 5, $f_0 \leftarrow R(f_1, \ldots, f_n)$ means applying the rule R on facts $(f_1, \ldots, f_n)$ to generate a new fact f_0.) If the computed intersection is empty (Line 7), then we know that this new fact does not hold at any version.

Otherwise, we search the IDB facts for f_0. If there already exists a fact f_0 with a different version label set, we set the *fixpoint* variable to false and update IDB by replacing the version annotation (Line 11). If f_0 is not already in IDB, this new found fact is added with its version annotation (Line 14) and in this case, *fixpoint* is also set to false. The algorithm terminates when a fixpoint is reached, and the resulting IDB facts (I) contains the analysis results annotated with the correct version labels. The correctness of Algorithm 1 is given in Theorem 1, and we provide the proof in the supplemental materials.

Theorem 1. *Let $\mathcal{I}$ and $\hat{\mathcal{I}}$ be the unlifted and lifted inference algorithms, respectively. Given an EDB E annotated with version strings from a set V and a Datalog program P, we have $\forall v \in V : \hat{\mathcal{I}}(E, P)|_v = \mathcal{I}(E|_v, P)$, where $\cdot|_v$ selects facts which are valid on version v.*

Lifting-Enabled Reusing. When a complex analysis task consists of multiple similar sub-tasks, intermediate results produced by earlier sub-tasks may be reused by the latter ones. This is not possible if the sub-tasks are implemented as standalone black-boxes. Algorithm 1 automatically enables the sharing of intermediate results: instead of performing an analysis on n versions of a project separately, we can run it once on the consolidated differential facts. Fig. 8 illustrates this process. The left side shows the case where the task T_A is executed on n versions individually, i.e., $\langle T_A(F_1), \ldots, T_A(F_n)\rangle$. When T_A is lifted, the new analysis process includes three sub-tasks, $\langle T_{M_1}, T_A(F^{(1..n)}), T_{M_2}\rangle$, where T_{M_1} merges $\{F_1, \ldots, F_n\}$ into $F^{(1..n)}$, T_A runs on the merged facts, and T_{M_2} consolidates the results with the correct version labels. Essentially, the lifted T_A enables the reusing of the analysis sub-tasks common to all versions.

4 EVALUATION

In this section, we aim to answer the following research questions through the empirical evaluation. **RQ1**: how does differential facts support evolution management tasks? **RQ2**: how significant is the

Table 1: Summary of experiments and RQs answered.

	History Slicing	CIA	RTS	Lifting
Sections	Secs. 4.1.1 and 4.2	Sec. 4.1.2	Sec. 4.3	Sec. 4.4
RQs	RQ1,2	RQ1,3		RQ4
Facts (Re)used	$\circlearrowleft G[T_s]$ $\circlearrowleft G[T_d]$ $\circlearrowleft G[T_h]$ $G[T_c]$	$\leftrightharpoons G[T_s]$ $\leftrightharpoons G[T_d]$	$\leftrightharpoons G[T_s]$ $\leftrightharpoons G[T_d]$	$\updownarrow$ intra-version facts

efficiency improvement brought by fact reusing? **RQ3**: how well does differential facts support cross-analysis reusing? **RQ4**: how significant is the efficiency improvement brought by lifting enabled analysis reusing?

The mappings between the experiments and RQs answered are shown in Table 1. In Sec. 4.1, two case studies, *history slicing* and *change impact analysis (CIA)* are conducted to answer RQ1. Then in Sec. 4.2 we answer RQ2 by demonstrating that cross-run facts reusing saves time in history slicing. To answer RQ3, we re-implement *regression test selection (RTS)* based on differential facts and validate cross-analysis reusing in Sec. 4.3. Finally, Sec. 4.4 answers RQ4 with a lifting-enabled reusing for pointer analysis.

In Table 1, we also summarize the facts reused in each experiment using the graph notations from Sec. 3.1, i.e., $G[T_s]$ corresponds to static dependency facts, $G[T_d]$ represents atomic changes, $G[T_h]$ represents commit dependencies and $G[T_c]$ captures coverage information. Symbols before each type of facts indicate how facts are reused: cross-run reusing is prefixed by $\circlearrowleft$, cross-analysis reusing is prefixed by $\leftrightharpoons$, and $\updownarrow$ means lifting-enabled reusing.

The following experiments were conducted on a 6-core Intel(R) Xeon(R) CPU E5-1650 v3 @ 3.50GHz machine with 16 GB RAM, running Debian sid, with OpenJDK 1.8.0_151 and Python 3.7.4. The fact extractor implementations, analysis scripts, and raw data are available as supplemental materials at: https://d-fact.github.io.

4.1 Case Study: DIFFBASE in Evolution Tasks

DIFFBASE serves as an infrastructure for storing, exchanging, and manipulating multi-version program relations embedded in software change histories, which enables a wide range of evolution management tasks. We conducted case studies on two of such tasks, namely, *semantic history slicing* and *change impact analysis*.

4.1.1 History Slicing with Differential Facts. The state-of-the-art history slicing technique, CSLICER [48], relies on static analysis of dependencies between atomic changes to decide which commits to keep in the history slices, in order to pass certain tests. It first analyzes the latest program version to collect test coverage information and then computes an over-approximated set of atomic changes touching the covered elements. Then, through *change dependency analysis* [48], it includes additional changes required for proper compilation of the program. Finally, the identified atomic changes are mapped back to the commits in the original change history. The produced history slices are guaranteed to not causing any merge conflict, and the resulting program is guaranteed to compile and pass the tests.

The original CSLICER was implemented as a monolithic application which only works on Java projects. In this case study, we re-implement it with DIFFBASE in order to achieve better reusing and support for multiple programming languages.

$$R_f = C\,;\,(\text{Ins} \cup \text{Upd} \cup \text{Del}) \tag{1}$$

$$D = C \circ (\text{Call} \cup \text{Ref} \cup \text{Contain})^{+} \tag{2}$$

$$R_d = D\,;\,(\text{Ins} \cup \text{Del}) \tag{3}$$

The aforementioned history slicing process can be described as algebraic operations on typed graphs. Each slicing criteria defined by a group of tests is a set of vertices T in the sub-graph Cov, where each edge connects two vertices, namely, a test entity and a code entity. To find the affected code entities by one specific slicing criteria, we find the corresponding code entities from its test entities, denoted by $C = T \circ \text{Cov}$. The atomic changes touching the covered elements are equivalent to the subset $R_f = (v, \_) \subseteq E[T_d]$ where $v \in C$, thus can be calculated by applying composition on C and all three types of atomic changes relation, as shown in Eq. (1).

Next is to calculate the set of all code entities depended on by C. This can be done by traversing the static dependency graph $G[T_s]$ starting from the vertices in C. We use the transitive closure operator to collect all vertices reachable from v in the dependency graph $G[T_s]$, and produce the resulting code entities by projecting C through the closure (Eq. (2)). Moreover, the atomic changes touching code entities in D is captured by the subset $R_d = (v, \_) \subseteq E[\text{Ins}] \cup E[\text{Del}]$, where $v \in D$. Hence, we derive R_d from the composition shown in Eq. (3). Upd is excluded in the calculation of R_d, since modifications to program entities only appearing in D shall not affect compilations. But insertions and deletions of entities in D are essential to avoid compilation errors.

R_f and R_d contains all the atomic changes which should be kept in the history slice. Therefore, the resulted history slice is the set of commits associated with change set $R_f \cup R_d$. We were able to verify the correctness of our implementation by comparing with the original CSLICER on a number of C/C++ and Java projects, which demonstrates DIFFBASE's support for interoperability. We also perform a performance evaluation in Sec. 4.2.

4.1.2 Change Impact Analysis with Differential Facts. It is also possible to implement change impact analysis [5, 56] using the same set of facts as in Sec. 4.1.1. From the high-level, we would like to identify the set of code entities potentially affected by a commit. This can be derived by first mapping the changed entities from the given commit and storing them in X.

$$X = (\text{Upd} \cup \text{Del} \cup \text{Ins})\,[1] \tag{4}$$

$$D = (\text{Call} \cup \text{Ref} \cup \text{Contain})^{*} \circ X \tag{5}$$

Then we derive the set of affected entities (D in Eq. (5)) by finding all entities that transitively depend on the elements of X. This is realized by projecting the reflexive transitive closure of the union of all dependency relations onto X.

The change impact analysis query can be further applied in a library upgrade scenario, where we try to find out the client classes affected by a library upgrade. When an original version of the library lib upgrades to the upgraded version lib', the fact set X' in Eq. (6) captures all the updates and deletions between lib and lib'. Note that we leave out insertions, because client cannot depend on

program entities which do not exist in the original version of the library.

$$X' = (\text{Upd} \cup \text{Del})\,[1] \tag{6}$$

Taking a client *cli* as input, the depedency graph $G[T_s]$ captures all *call*, *contain*, *reference* relations between entities within *cli*, as well as *call* and *reference* from *cli* to *lib*. Therefore, using Eq. (5) on X', we can find out all program entities in the client which directly or indirectly depend on the modified or deleted library entities. In practice, D can be used to guide the test generator to more efficiently generate tests. After executing those tests with both *lib* and *lib'*, any test failure reveals an incompatibility caused on *cli*, brought by the upgrade of *lib*. This effectively implements a light-weight solution for the *client-specific upgrade checking* problem [49]. We implemented this task with the same fact extractors used in Sec. 4.1.1 and more results are presented on the companion website.

4.2 Experiment: Cross-Run Fact Reusing

In this experiment, to demonstrate the efficiency improvement brought by cross-run fact reusing, we compare our differential facts based history slicing technique with CSlicer, a state-of-the-art semantic history slicing tool.

4.2.1 Subjects. Since CSlicer does not work on C/C++ projects, the performance evaluation was conducted on Java projects. To evaluate how fact reusing can improve the efficiency of history slicing, we use a benchmark consisting of 37 functionalities selected from the DoSC dataset [71]. Each functionality is identified by a unique key which refers to its corresponding issue key on the JIRA issue tracker [36] and is accompanied by a set of tests. DoSC includes the starting and ending commit of the software history which determine the life cycle of the development of a functionality.

We selected functionalities from eight open source projects, namely, Compress [14], Configuration [15], CSV [17], Flume [22], IO [34], Lang [43], Maven [52], and Net [53]. All of these projects are written in Java and have publicly accessible change histories. Initially, five functionalities were randomly selected from each project available in DoSC. We then removed functionalities that are currently not supported by CSlicer (e.g., those that include changes that modify non-Java files, because CSlicer does not handle such files). In the end, our experiments used 10 different history ranges and 37 functionalities from eight projects—see Table 2 (the first column) for their unique keys. In Table 2, each ID corresponds to a history range marked by the SHA-1 of the *Start* and *End* commits, as well as the number of commits (i.e., *Length*). Those subjects from the same projects with different history ranges are distinguished by suffixes (e.g., "-1" and "-2").

4.2.2 Results. The efficiency improvement of the fact-based history slicing, implemented using Diffbase, come from the reusability of facts. With our approach, when there exist slicing tasks which operate on the same fragment of history differed only in slicing criteria, all facts except coverage facts are generated once and stored for reusing, while CSlicer does repeated work. We compare the total time used by our approach and CSlicer on all subjects and also on the basis of each unique history segment, defined by the 3-tuple (Project, HistoryStart, HistoryEnd) in Table 2. As shown in

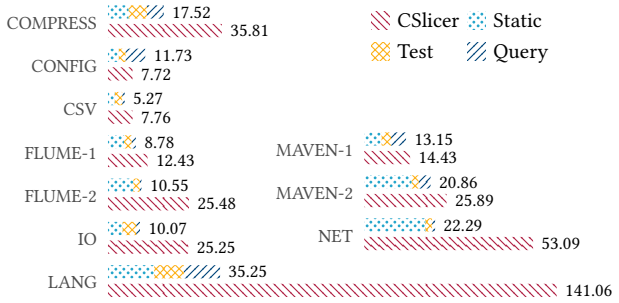


Figure 9: Comparison of the time costs (in seconds) between Diffbase and CSlicer.

Fig. 9, each pair of bars represent the time costs of Diffbase and CSlicer, respectively. The time costs of Diffbase are divided into three parts: static dependency fact extraction ("Static", recorded per history segment), test coverage fact extraction ("Test", recorded per functionality), and query processing ("Query"). Our approach performs consistently better than CSlicer. The one exception where it performs worse is CONFIG, which has the shortest history and fewest LOC in all projects. Therefore, time saved from storing differential facts is counteracted by the time costs of querying, depicted by a larger portion of query processing time shown in the stacked part of the bar plot.

Answer to RQ1 & RQ2: In all experiments, differential facts based history slicing can produce correct slices, matching the results of the state-of-the-art tools. It also operates on multiple languages. On average, we reduce 44% of the time cost by reusing differential facts. It performs worse in only one subject, while it runs 75% faster in the best case.

4.3 Experiment: Cross-Analysis Fact Reusing

In this experiment, to evaluate the effectiveness of cross-analysis fact reusing, we implemented the regression test selection (RTS) technique proposed by Yoo et al. [67] based on differential facts, and compared it with three state-of-the-art RTS tools.

We implemented our fact-based RTS by reusing the facts collected from the change impact analysis experiment in Sec. 4.1.2. Specifically, for a given commit, we first extracted the set of program entities affected by it (D in Eq. (5)), then intersected these entities with the set of test entities to obtain the set of tests affected by the commit (R in Eq. (7)). From a query's perspective, given the set of test entities τ, R is the intersection of τ and D.

$$R = \tau \cap D \tag{7}$$

The resulting set R contains the selected regression tests that needs to be rerun on this new commit. This implementation demonstrates how differential facts are used, in a cross-analysis manner, to support reusing intermediate results across different evolution management tasks.

4.3.1 Experiment Setup. To evaluate our fact-based RTS, we use the DefectsEP component proposed in [70], a state-of-the-art framework for evaluating RTS tools. DefectsEP uses 151 real-world bugs from three Java projects as evaluation subjects, which were selected from the Defects4J dataset [38]. Table 3 shows the original project

Table 2: Subjects used in the history slicing experiment.

ID	Projects	History						# Criteria
		Start	End	Length	# Edited Files	LOC(+)	LOC(-)	
COMPRESS	Apache Commons Compress	99bc508	b29395d	148	144	4,644	2,006	5
CONFIG	Apache Commons Configuration	89428f1	9fb4ad8	50	34	1,201	655	2
CSV	Apache Commons CSV	b230a6f5	7310e5c6	79	28	1,640	713	4
FLUME-1	Apache Flume	cda3bd10	31d45f1b	101	181	14,742	3,097	3
FLUME-2		f7560038	5e400ea8	100	428	17,341	8,187	3
IO	Apache Commons IO	8de491fc	b1b9f1af	136	182	5,647	1,681	5
LANG	Apache Commons Lang	24767d6	76cc69c	262	146	6,741	2,076	8
MAVEN-1	Apache Maven	b175144	308d4d4	51	78	1,816	713	3
MAVEN-2		b7e3ce2	ea8b2b0	97	160	4,431	4,144	2
NET	Apache Commons Net	d483631	abd6711	269	233	6,845	2,393	2

IDs and bug IDs of these subjects in Defects4J. The bug IDs are consecutive and inclusive, e.g., "28-53" means the bugs No.28, No.53, and all others falling in between.

In DefectsEP, for each bug, two consecutive revisions—V_{bug} and V_{fix}—of the program are provided in the dataset, such that there is at least one *triggering* test fails at V_{bug} and passes at V_{fix}. Given a set of RTS tools as input, for each bug, DefectsEP runs each tool on both versions in the order $V_{\text{fix}} \rightarrow V_{\text{bug}}$, and collects the number of selected and failing tests on V_{bug} of each tool. In the end, for each RTS tool under test, DefectsEP contrasts its result with other tools and RetestAll (a baseline strategy that always runs all tests) and uses a set of rules to detect abnormal behaviors. For each identified abnormal behavior, it reports a violation to the user. By default, DefectsEP evaluates safety, precision, and generality of each tool. In our experiment, we focus on safety and precision, as they are fundamental properties of RTS tools [70].

To evaluate safety, we compare the number of *newly failing* tests on V_{fix} selected by that tool versus RetestAll, i.e., applying the rule R1 in [70]. The rationale is that a safe RTS tool should never miss a newly failing test, as the test is for sure affected by the changes in $V_{\text{fix}} \rightarrow V_{\text{bug}}$. If on a subject, a tool misses any newly failing test, then the subject is counted as a safety violation of the tool. To evaluate precision, we compare the number of tests selected by each tool. Given RTS tools R_1 and R_2, if both tools do not violate safety, i.e., do not miss newly failing tests, R_1 is considered to be more precise than R_2 if R_1 selects fewer tests than R_2.

We compare our fact-based RTS with all the three state-of-the-art RTS tools evaluated in [70]: Clover [55], Ekstazi [27], and STARTS [45]. Clover is developed by industry, while Ekstazi and STARTS are developed by researchers. We run DefectsEP on the four tools on all 151 subjects, comparing their numbers of safety violations and numbers of selected tests (for precision).

4.3.2 Results. Table 3 shows the results of the comparison. The numbers of safety violations are shown on the "# Safety Violations" row in Table 3, while the percentage of the selected test methods are measured on average over all the subjects of a project. For example, on project commons-lang [43] (LANG in the table), Clover selects 10.4% tests on average over bugs 28-53; on all the subjects, Clover has 16 safety violations in total.

According to the results, our fact-based RTS is the only tool having no safety violations. Both Ekstazi and STARTS have one violation and Clover has 16 violations.

Table 3: Experimental results of the RTS experiment.

Projects	Bug IDs	Selected Test Methods (Avg. %)			
		Clover	Ekstazi	STARTS	FACTS
LANG	28-53	10.4%	26.4%	53.9%	54.1%
MATH	5-104	9.5%	12.8%	20.0%	16.7%
TIME	1-20,22-26	100.0%	47.8%	100.0%	100.0%
# Safety Violations		16	1	1	0

For precision, the results indicate that our fact-based RTS has similar precision as STARTS, though it is less precise than Ekstazi and Clover. This is expected, because Ekstazi and Clover perform test selection based on dynamic analysis. Using runtime information, they can further refine their dependency analysis by reducing the set of code entities that are not covered by tests. Still, our fact-based RTS has comparable precision as STARTS, a state-of-the-art static RTS tool. This further demonstrates the effectiveness and applicability of our approach.

Most facts used by our RTS technique are pre-computed and shared among different tasks (e.g., case studies in Sec. 4.1), including inter-versions facts representing atomic changes and intra-version facts depicting static dependencies between program entities. This is another example for the benefit of reusing facts across analyses.

Answer to RQ3: On the 151 subjects of DefectsEP, our fact-based RTS technique outperforms all other state-of-the-art RTS tools in safety, and obtains similar precision as STARTS, the most recent static RTS tool. The experiment results confirm the applicability of our fact-based RTS tool and the effectiveness of cross-analysis fact reusing.

4.4 Experiment: Lifting Enabled Reusing

In this experiment, we evaluate time and space savings brought by lifting enabled analysis reusing. For this purpose, we conducted a one-call-site heap-sensitive pointer analysis on Java projects using Doop [9]. We used six out of the eight projects in Table 2 as the subjects. Maven and Flume were excluded because they contain multiple modules, currently not supported by our tooling. For each project, ten versions before the *End* version were used and the same analysis was conducted with two different approaches. Without lifting, Doop was repeatedly ran on each version and the results were collected. With lifting, a modified version of Doop was first

Table 4: Time and space savings brought by analysis lifting.

Project	Before Lifting		After Lifting			
	T (ms)	S (MB)	T' (ms)	ΔT	S' (MB)	ΔS
CSV	49129	40.26	5287	89.65%	4.88	87.88%
IO	53752	360.92	7085	86.82%	25.87	92.83%
CONFIG	91421	887.32	15751	82.77%	81.10	90.86%
COMPRESS	48581	486.84	11554	76.28%	81.51	83.26%
LANG	72892	614.19	11641	84.03%	75.98	87.63%
NET	60736	313.62	8750	85.59%	41.62	86.73%

invoked on each version with the *stop-at-facts* mode (i.e., sub-task T_{M_1} in Fig. 8) so that version annotated facts were generated, which were then consolidated in Diffbase. During the consolidation process, duplicated facts were removed and each fact was annotated with a set of version labels. Finally, Doop was invoked again in the *start-after-facts* mode, which consumes the consolidated differential facts and generates the results following Algorithm 1 (sub-tasks T_A and T_{M_2} in Fig. 8).

Table 4 shows the results of the experiments. The project names, time taken (in milliseconds), and space taken for storing the facts (in MBs), before and after the analysis lifting, are listed in the columns. Columns "T" and "T'" list the time costs, while Columns "S" and "S'" list the space usage. T' includes the time spent on merging, which shows that the time gain brought by reusing outweighs the overhead of lifting computation. The percentage reductions are listed in Columns "ΔT" and "ΔS", respectively. For most projects, lifting introduces about 80% savings on both time and space.

Answer to RQ4: Analysis lifting powered by differential facts brings significant savings (more than 80% reduction in average) in both space and time through the reusing of intermediate analysis results.

4.5 Threats to Validity

Subjects used in our evaluation may not be representative. To mitigate this threat, we used the published dataset from prior research on both history slicing and regression test selection; these subjects were taken from large open-source projects covering diverse domains. We ran all the experiments on a single machine, and our findings related to execution time might differ on another machine. During the implementation of our tools, we have used several machines and observed similar trends on these machines.

5 RELATED WORK

Our work intersects with a few research areas and is mostly related to *program fact extraction* and *software evolution management*.

Fact Extraction. Fact extractors are custom, human-defined analyzers that automatically scan structured software artifacts to pull out pertinent details to be included in a resultant factbase. The idea of reverse engineering programs and representing relevant information as facts is not new. The existing work on fact extraction can be broadly categorized into the *intra-version* and *inter-version* ones.

Intra-version fact extraction focuses on a single version of the program artifacts. Fact extractors for different programming languages and platforms have been built, including *Javax* [2] for Java, *Cppx* [1] and *ClangEx* [3] for C/C++, and *ASX* [18] for assembler, objects, dynamic libraries and executables. We have developed our own fact extractor for Java and modified *ClangEx* for C/C++ to generate intra-version facts of the format compatible with the rest of the differential facts. There are also many downstream analyses performed on the intra-version facts for architecture understanding [7], visualization and redocumentation [39].

Inter-version fact extraction relies on sophisticated *structural differencing* [8, 12, 21] and *code change classification* [23, 24, 30] algorithms. The former is used to compute an optimal sequence of atomic edit operations that can transform one AST into another, and the latter is used to classify atomic changes according to their change types. Examples of AST differencing algorithm include ChangeDistiller [24] and GumTree [21]. They both use individual statements as the smallest AST nodes and categorizes source code changes into elementary tree edit operations, for instance, insert, delete, move and update. ChangeDistiller only works on Java while GumTree works on a number of different languages. GumTree converts a source file into a language-agnostic tree format and is able to export the tree differences into various formats. We built our inter-version fact extractor based on GumTree by implementing a specialized tree exporter to the TA format. We use canonical identifiers for entity nodes to ensure the proper linkages between inter- and intra-version facts. The key difference of our approach is the emphasis on the reusing of facts, which was not explicitly established in the previous work. We also retain versioning information as edge attributes to allow commit-level history-related analyses.

Inter-version facts can go beyond structural differences. Kim et al. [40, 41] proposed to infer rules from systematic structural differences between versions, which be viewed as logical summarizations of changes. Le and Pattison [44] introduced the *Multiversion Interprocedural Control Flow Graphs* (MVICFG) to integrate and compare control flow of multiple versions of programs. Albeit their differences in the representations chosen, the results from these analyses can all be encoded as the lower-level differential facts and consumed by other downstream analyses.

Evolution Management. There is a large body of work on analyzing and understanding software histories. The basic research goals are retrieving useful information from change histories to help understand development practices [11, 50, 51, 57], localize bugs [68, 69] and features [48], and support predictions [31, 74].

The problem of classifying changes and identifying change impacts widely present in many evolution management tasks. Brito et al. [10] proposed a tool, named APIDiff, to identify API breaking and non-breaking changes between two versions of a Java library. APIDiff defines rules to classify changes based on their types, e.g., the breaking types (e.g., removal) and non-breaking types (e.g., addition). Yokomori et al. [66] studied the evolution of an application and its underlying libraries or frameworks, by analyzing the evolution of their use relationship. They use component ranking to identify core components of a library, and analyze the impact of a change based on the use relationship between the changed component and the core components. Gyori et al. [28] evaluated regression test selection (RTS) opportunities in the Maven Central ecosystem by changing a library and running RTS techniques on all its transitive clients. In the case studies presented in Sec. 4.1.2,

we performed similar change impact analysis between the library changes and the client code with the help of differential facts. Our analysis can easily be ported to support other tasks with minimal modifications to the query scripts.

Incremental Datalog-Based Analysis. Incremental code queries based on Datalog have been used for analysis of evolving software systems. Hajiyev et al. [29] implemented *CodeQuest*, a source code querying tool for program understanding, which incrementally updates its database when program changes occur. They compiled Datalog to SQL to improve scalability and achieve incremental updates by only re-compiling changed source code and replacing affected facts. Eichberg et al. [19] created a domain-specific language based on Datalog for continuously checking structural dependencies between program entities. While we adopted similar incremental strategies for extracting facts, our approach enables querying across multiple versions as well as analysis lifting, through customized storage and query engines for differential facts.

Analysis Lifting. Various attempts on optimizing analysis with declarative methods has been made. Shahin et al. [59] lifted a Datalog engine so that it can analyze all the product variants of a software product line at once. The way we represent differential facts enables lifting in a similar style, which is otherwise not possible without the feature models available in product lines. However, the effectiveness of lifting on version history has not been studied before. Visser et al. [63] proposed a results caching mechanism to reuse SMT query results across multiple runs and analyses. Their work is specific to SMT constraints and does not answer the question on how to uniformly represent knowledge about software development artifacts, especially inter-version facts.

6 CONCLUSION

In this paper, we proposed DIFFBASE, a simple yet powerful representation of pertinent information in evolving software artifacts. The inter-version changes are treated as first-class objects and facilitate effective cross-version fact querying. The resultant differential factbase allows efficient storage, querying, and manipulation of facts. We demonstrated the applications of DIFFBASE in supporting evolution management tasks such as history slicing and regression test selection. The experimental results highlight the benefits of our approach in terms of sharing and reusing of intermediate analysis results as well as cross-language/platform interoperability.

ACKNOWLEDGMENTS

This research is supported by the Ministry of Education, Singapore, under its Academic Research Fund Tier 2 (MOE2019-T2-1-040).

REFERENCES

[1] 2001. Cppx - Open Source C++ Fact Extractor. http://www.swag.uwaterloo.ca/cppx.
[2] 2010. Javex - Java Fact Extractor. http://www.swag.uwaterloo.ca/javex.
[3] 2018. ClangEx – A Fast C/C++ Fact Extractor. https://github.com/bmuscede/ClangEx.
[4] Lars Ole Andersen. 1994. *Program Analysis and Specialization for the C Programming Language.* Ph.D. Dissertation. University of Cophenhagen.
[5] Robert S. Arnold. 1996. *Software Change Impact Analysis.* IEEE Computer Society Press, Los Alamitos, CA, USA.
[6] BCEL 2015. Apache Commons Byte Code Engineering Library. https://commons.apache.org/proper/commons-bcel.
[7] Dirk Beyer, Andreas Noack, and Claus Lewerentz. 2003. Simple and Efficient Relational Querying of Software Structures. In *Proceedings of the 10th Working Conference on Reverse Engineering*. 216–225.
[8] Philip Bille. 2005. A Survey on Tree Edit Distance and Related Problems. *Theoretical Computer Science* 337, 1-3 (June 2005), 217–239.
[9] Martin Bravenboer and Yannis Smaragdakis. 2009. Strictly declarative specification of sophisticated points-to analyses. In *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*. 243–262.
[10] Aline Brito, Laerte Xavier, André C. Hora, and Marco Tulio Valente. 2018. APIDiff: Detecting API breaking changes. In *International Conference on Software Analysis, Evolution and Reengineering*. 507–511.
[11] Yuriy Brun, Reid Holmes, Michael D. Ernst, and David Notkin. 2013. Early Detection of Collaboration Conflicts and Risks. *IEEE Transactions on Software Engineering* 39, 10 (Oct. 2013), 1358–1375.
[12] Sudarshan S. Chawathe, Anand Rajaraman, Hector Garcia-Molina, and Jennifer Widom. 1996. Change Detection in Hierarchically Structured Information. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*. 493–504.
[13] Clang 2019. Clang: a C language family frontend for LLVM. https://clang.llvm.org.
[14] Compress 2018. Apache Commons Compress Library. https://commons.apache.org/proper/commons-compress.
[15] Configuration 2019. Commons Configuration Library. https://commons.apache.org/proper/commons-configuration.
[16] Catarina Costa, Jair Figueiredo, Anita Sarma, and Leonardo Murta. 2016. TIPMerge: Recommending Developers for Merging Branches. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, New York, NY, USA, 998–1002. https://doi.org/10.1145/2950290.2983936
[17] CSV 2017. Apache Commons CSV Library. https://commons.apache.org/proper/commons-csv.
[18] I. J. Davis and M. W. Godfrey. 2010. From Whence It Came: Detecting Source Code Clones by Analyzing Assembler. In *2010 17th Working Conference on Reverse Engineering*. 242–246. https://doi.org/10.1109/WCRE.2010.35
[19] Michael Eichberg, Sven Kloppenburg, Karl Klose, and Mira Mezini. 2008. Defining and continuous checking of structural program dependencies. In *Proceedings of the 30th international conference on Software engineering*. 391–400.
[20] Emelie Engström, Per Runeson, and Mats Skoglund. 2010. A Systematic Review on Regression Test Selection Techniques. *Information and Software Technology* 52, 1 (Jan. 2010), 14–30. https://doi.org/10.1016/j.infsof.2009.07.001
[21] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. 2014. Fine-Grained and Accurate Source Code Differencing. In *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014*. 313–324. https://doi.org/10.1145/2642937.2642982
[22] Flume 2019. Flume. https://flume.apache.org.
[23] Beat Fluri and Harald C. Gall. 2006. Classifying Change Types for Qualifying Change Couplings. In *Proceedings of the 14th IEEE International Conference on Program Comprehension*. IEEE, 35–45.
[24] Beat Fluri, Michael Wuersch, Martin Pinzger, and Harald Gall. 2007. Change Distilling: Tree Differencing for Fine-Grained Source Code Change Extraction. *IEEE Transactions on Software Engineering* 33, 11 (Nov. 2007), 725–743.
[25] GCov 2019. GCov. https://gcc.gnu.org/onlinedocs/gcc/Gcov.html.
[26] Git 2016. Git Version Control System. https://git-scm.com.
[27] Milos Gligoric, Lamyaa Eloussi, and Darko Marinov. 2015. Practical regression test selection with dynamic file dependencies. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. 211–222.
[28] Alex Gyori, Owolabi Legunsen, Farah Hariri, and Darko Marinov. 2018. Evaluating Regression Test Selection Opportunities in a Very Large Open-Source Ecosystem. In *International Symposium on Software Reliability Engineering*. 112–122.
[29] Elnar Hajiyev, Mathieu Verbaere, and Oege De Moor. 2006. Codequest: Scalable source code queries with Datalog. In *European Conference on Object-Oriented Programming*. Springer, 2–27.
[30] Masatomo Hashimoto and Akira Mori. 2008. Diff/TS: A Tool for Fine-Grained Structural Change Analysis. In *Proceedings of the 15th Working Conference on Reverse Engineering*. IEEE, Antwerp, 279–288.
[31] Kim Herzig and Andreas Zeller. 2013. The Impact of Tangled Code Changes. In *Proceedings of the 10th Working Conference on Mining Software Repositories*. IEEE Press, Piscataway, NJ, USA, 121–130.
[32] Ric Holt. 1997. *TA: The Tuple Attribute Language.* Technical Report. University of Waterloo.
[33] Richard C Holt. 1998. Structural Manipulations of Software Architecture Using Tarski Relational Algebra. In *Proceedings Fifth Working Conference on Reverse Engineering (Cat. No. 98TB100261)*. IEEE, 210–219.
[34] IO 2017. Apache Commons IO Library. https://commons.apache.org/proper/commons-io.

[35] Jacoco 2016. JaCoCo Java Code Coverage Library. http://www.eclemma.org/jacoco.

[36] JIRA 2017. JIRA Software. https://www.atlassian.com/software/jira.

[37] Herbert Jordan, Bernhard Scholz, and Pavle Subotić. 2016. Soufflé: On synthesis of program analyzers. In *International Conference on Computer Aided Verification*. Springer, 422–430.

[38] René Just, Darioush Jalali, and Michael D Ernst. 2014. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. 437–440.

[39] Holger M. Kienle and Hausi A. Müller. 2010. Rigi – An Environment for Software Reverse Engineering, Exploration, Visualization, and Redocumentation. *Science of Computer Programming* 75, 4 (April 2010), 247–263. https://doi.org/10.1016/j.scico.2009.10.007

[40] Miryung Kim and David Notkin. 2009. Discovering and Representing Systematic Code Changes. In *Proceedings of the 31st International Conference on Software Engineering*. IEEE Computer Society, Washington, DC, USA, 309–319. https://doi.org/10.1109/ICSE.2009.5070531

[41] Miryung Kim, David Notkin, Dan Grossman, and Gary Wilson Jr. 2013. Identifying and Summarizing Systematic Code Changes via Rule Inference. *IEEE Transactions on Software Engineering* 39, 1 (Jan. 2013), 45–62. https://doi.org/10.1109/TSE.2012.16

[42] Sunghun Kim, E. James Whitehead, Jr., and Yi Zhang. 2008. Classifying Software Changes: Clean or Buggy? *IEEE Transactions on Software Engineering* 34, 2 (March 2008), 181–196. https://doi.org/10.1109/TSE.2007.70773

[43] Lang 2018. Apache Commons Lang Library. https://commons.apache.org/proper/commons-lang.

[44] Wei Le and Shannon D. Pattison. 2014. Patch Verification via Multiversion Interprocedural Control Flow Graphs. In *Proceedings of the 36th International Conference on Software Engineering*. ACM, New York, NY, USA, 1047–1058. https://doi.org/10.1145/2568225.2568304

[45] Owolabi Legunsen, Farah Hariri, August Shi, Yafeng Lu, Lingming Zhang, and Darko Marinov. 2016. An extensive study of static regression test selection in modern software evolution. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 583–594.

[46] Yi Li, Chenguang Zhu, Julia Rubin, and Marsha Chechik. 2016. Precise Semantic History Slicing through Dynamic Delta Refinement. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. Singapore, Singapore, 495–506.

[47] Yi Li, Chenguang Zhu, Julia Rubin, and Marsha Chechik. 2017. FHistorian: Locating Features in Version Histories. In *Proceedings of the 21st International Systems and Software Product Line Conference - Volume A*. ACM, New York, NY, USA, 49–58. https://doi.org/10.1145/3106195.3106216

[48] Yi Li, Chenguang Zhu, Julia Rubin, and Marsha Chechik. 2017. Semantic Slicing of Software Version Histories. *IEEE Transactions on Software Engineering* 44, 2 (Feburary 2017), 182–201.

[49] Federico Mora, Yi Li, Julia Rubin, and Marsha Chechik. 2018. Client-Specific Equivalence Checking. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE 2018)*. ACM, New York, NY, USA, 441–451. https://doi.org/10.1145/3238147.3238178

[50] Kivanç Muşlu, Luke Swart, Yuriy Brun, and Michael D. Ernst. 2015. Development History Granularity Transformations. In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering*. Lincoln, NE, USA, 697–702.

[51] Emerson Murphy-Hill, Chris Parnin, and Andrew P. Black. 2012. How We Refactor, and How We Know It. *IEEE Transactions on Software Engineering* 38, 1 (Jan 2012), 5–18.

[52] Mvn 2015. Apache Maven Project. https://maven.apache.org.

[53] Net 2017. Apache Commons Net Library. https://commons.apache.org/proper/commons-net.

[54] Anh Tuan Nguyen, Michael Hilton, Mihai Codoban, Hoan Anh Nguyen, Lily Mast, Eli Rademacher, Tien N. Nguyen, and Danny Dig. 2016. API Code Recommendation Using Statistical Learning from Fine-grained Changes. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, New York, NY, USA, 511–522. https://doi.org/10.1145/2950290.2950333

[55] Marek Parfianowicz. 2017. Open Clover. https://openclover.org.

[56] Xiaoxia Ren, Fenil Shah, Frank Tip, Barbara G. Ryder, and Ophelia Chesley. 2004. Chianti: A Tool for Change Impact Analysis of Java Programs. In *Proceedings of the 19th aAnual ACM SIGPLAN conference on Object-oriented Programming, Systems, Languages, and Applications*. ACM, New York, NY, USA, 432–448.

[57] Francisco Servant and James A. Jones. 2011. History slicing. In *Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering*. 452–455.

[58] Francisco Servant and James A. Jones. 2012. WhoseFault: Automatic Developer-To-Fault Assignment Through Fault Localization. In *Proceedings of the 34th International Conference on Software Engineering*. IEEE Press, Piscataway, NJ, USA, 36–46.

[59] Ramy Shahin, Marsha Chechik, and Rick Salay. 2019. Lifting datalog-based analyses to software product lines. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 39–49.

[60] Ian Sommerville. 2004. *Software Engineering (7th Edition)*. Pearson Addison Wesley.

[61] SVN 2016. Apache Subversion (SVN) Version Control System. http://subversion.apache.org.

[62] Alfred Tarski. 1941. On the Calculus of Relations. *The Journal of Symbolic Logic* 6, 3 (1941), 73–89.

[63] Willem Visser, Jaco Geldenhuys, and Matthew B Dwyer. 2012. Green: reducing, reusing and recycling constraints in program analysis. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. 1–11.

[64] Byron J. Williams and Jeffrey C. Carver. 2010. Characterizing Software Architecture Changes: A Systematic Review. *Information and Software Technology* 52, 1 (Jan. 2010), 31–51. https://doi.org/10.1016/j.infsof.2009.07.002

[65] Jingwei Wu. 2004. JGrok: A query language for reverse engineering. https://www.swag.uwaterloo.ca/jgrok.

[66] Reishi Yokomori, Harvey P. Siy, Masami Noro, and Katsuro Inoue. 2009. Assessing the impact of framework changes using component ranking. In *International Conference on Software Maintenance*. 189–198.

[67] Shin Yoo and Mark Harman. 2012. Regression Testing Minimization, Selection and Prioritization: A Survey. *Software Testing, Verification & Reliability* 22, 2 (March 2012), 67–120.

[68] Andreas Zeller. 1999. Yesterday, My Program Worked. Today, It Does Not. Why?. In *Proceedings of the 7th European Software Engineering Conference Held Jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. Springer-Verlag, London, UK, UK, 253–267.

[69] Andreas Zeller and Ralf Hildebrandt. 2002. Simplifying and Isolating Failure-Inducing Input. *IEEE Transactions on Software Engineering* 28, 2 (2002), 183–200.

[70] Chenguang Zhu, Owolabi Legunsen, August Shi, and Milos Gligoric. 2019. A Framework for Checking Regression Test Selection Tools. In *2019 IEEE/ACM 41st International Conference on Software Engineering*. IEEE, 430–441.

[71] Chenguang Zhu, Yi Li, Julia Rubin, and Marsha Chechik. 2017. A Dataset for Dynamic Discovery of Semantic Changes in Version Controlled Software Histories. In *Proceedings of the 14th International Conference on Mining Software Repositories*. IEEE Press, Piscataway, NJ, USA, 523–526. https://doi.org/10.1109/MSR.2017.49

[72] Chenguang Zhu, Yi Li, Julia Rubin, and Marsha Chechik. 2020. GenSlice: Generalized Semantic History Slicing. In *2020 IEEE International Conference on Software Maintenance and Evolution*. 81–91. https://doi.org/10.1109/ICSME46990.2020.00018

[73] Thomas Zimmermann, Stephan Diehl, and Andreas Zeller. 2003. How History Justifies System Architecture (or Not). In *Proceedings of the 6th International Workshop on Principles of Software Evolution*. IEEE Computer Society, Washington, DC, USA, 73–83.

[74] Thomas Zimmermann, Peter Weißgerber, Stephan Diehl, and Andreas Zeller. 2004. Mining Version Histories to Guide Software Changes. In *Proceedings of the 26th International Conference on Software Engineering*. IEEE Computer Society, Washington, DC, USA, 563–572.

A SUPPLEMENTAL MATERIALS

A.1 Detailed Walk-Through of Inter-Version Fact Extraction

To give a more detailed view on the typed-graph model and inter-version fact extraction from software change histories, we provide a walk-through of the example illustrated in the typed graph (Fig. 3) with textual differences and associated facts. As shown in Fig. 10, the short history includes changes on two simple classes, `A` and `B`. There are four commits shown, from C_0 to C_3, among which C_0 is the initial commit and its text diff is left out. Alongside the commit history represented by the line in the middle are unified diff and the corresponding differential facts.

In change set C_1, a member declaration `B::u` was removed, replaced by another member `B::y` and a member function `B::f()` was added. According to Defs. 1 and 2, the changes can be described as a change set Δ_1 including three AST transformations:

- DEL(`B::u@C0`)
- INS((`B::y@C1, B::static-int-y, 0`),`B`))
- INS((`B::f@C1, B::static-int-f-int, {return x-1}`),`B`)

Accordingly, three facts reflecting the changes are produced, as shown on the opposite sides of textual diffs in the figure.

Code entities in facts are represented as strings containing entity names, types, other modifiers if exist, and version numbers. For simplicity, $C_0, \ldots, C_3$ are used instead of the commit hashes and attributes enclosed in brackets are written right below the relations they belong to, to avoid duplication of relation texts. Each INS operation can be converted to an Insert fact, using the *id* field as its second operand, while the first operand is a special code entity denoted by `NULL`. Actual contents of the inserted entities defined as *value* are left out in facts. Meanwhile, the parent node can be found with the help of intra-version facts as shown in Fig. 3. A Delete fact can be deduced by using the only parameter of DEL operation as its first operand, while keeping the second one as `NULL`.

Then in C_2, a member declaration `A::x` was inserted and the body of `A::g()` was changed. Δ_2 includes two atomic changes displayed as follows.

- INS((`A::x@C2, A::int-x, nil`),`A`))
- UPD(`A::int-g@C1, {return B::y+1;}`)

The Update facts are also produced from the UPD operations without the *value*, and a string representation of the AST node names and the version information are used as the two operands.

Finally, in C_3, a member function `B::h()` was added, which calls `B::f()` and references `A::x`.

- INS((`A::int-h@C3, A::int-h, {return B::f(x);}`),`A`))

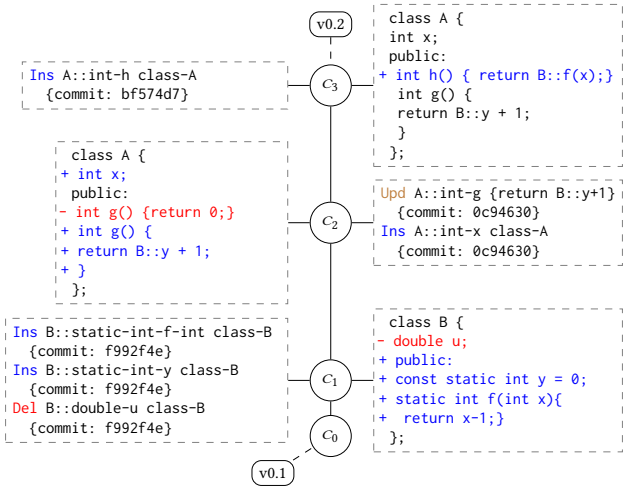


Figure 10: Example change history.

A.2 Proof of Theorem 1

We first prove the following lemma.

LEMMA. $(\forall v \in V)\, \hat{\mathcal{I}}(E,P)|_v \cup E|_v = \mathcal{I}(E|_v,P) \cup E|_v \iff (\forall v \in V)\, \hat{\mathcal{I}}(E,P)|_v = \mathcal{I}(E|_v,P)$

PROOF. Since in our notation, both $\mathcal{I}(E|_v,P)$ and $\hat{\mathcal{I}}(E,P)|_v$ only contains facts of IDB relations and E only contains facts of EDB relations, there is no intersection between them, i.e., we have $\mathcal{I}(E|_v,P) \cap E|_v = \emptyset$ and $\hat{\mathcal{I}}(E,P)|_v \cap E|_v = \emptyset$.

Therefore, according to basic set theory, we have $(\forall v \in V)\, \hat{\mathcal{I}}(E,P)|_v \cup E|_v = \mathcal{I}(E|_v,P) \cup E|_v \iff (\forall v \in V)\, \hat{\mathcal{I}}(E,P)|_v = \mathcal{I}(E|_v,P)$. □

$$\frac{r_0 :- r_1, \ldots, r_n \quad (\forall i : 0 \le i \le n)[x \mapsto s(x)]r_i = f_i \quad (\forall i : 1 \le i \le n)(f_i, V(f_i)) \in E \cup I}{(f_0, \bigcap_{1 \le i \le n} V(f_i)) \in I}\ MP$$

Figure 11: Modus ponens for the lifted Datalog inference in Algorithm 1.

Theorem. *Let $\mathcal{I}$ and $\hat{\mathcal{I}}$ be the unlifted and lifted inference algorithms, respectively. Given an EDB E annotated with version strings from a set V and a Datalog program P, we have $(\forall v \in V)\, \hat{\mathcal{I}}(E, P)|_v = \mathcal{I}(E|_v, P)$, where $\cdot|_v$ selects facts which are valid on version v.*

Proof. To simplify the structural induction, we prove $(\forall v \in V)\, \hat{\mathcal{I}}(E, P)|_v \cup E|_v = \mathcal{I}(E|_v, P) \cup E|_v$ here. We have proved in the lemma above that this is equivalent to the original thorem.

To remind readers about the lifted inference algorithm, Fig. 11 shows the inference rule in the form of modus ponens, where $[x \mapsto s(x)]r_i = f_i$ means applying a substitution from variables to constant symbols on predicate r_i so that f_i is a gounded fact. Any fact $f_0 \in \hat{\mathcal{I}}(E, P)|_v$ can be derived starting from input facts, by applying *MP* repeatly. The process forms a tree on which we can apply structural induction.

To prove the equality of the two sets, we prove that for any facts f_0, $f_0 \in \hat{\mathcal{I}}(E, P)|_v \cup E|_v \iff f_0 \in \mathcal{I}(E|_v, P) \cup E|_v$ by structural induction on the derivation tree of f_0.

Base case. The leaf nodes of the derivation tree must be in the input EDB. If $f_0 \in E|_v$, then the property holds.

Induction Hypothesis. The hypothesis is that the property holds for premises of the rule application $f_0 :- f_1, \ldots, f_n$, i.e.,

$$(\forall i : 1 \le i \le n)\ f_i \in \hat{\mathcal{I}}(E, P)|_v \cup E|_v \iff f_i \in \mathcal{I}(E|_v, P) \cup E|_v$$

Induction Step. In $\hat{\mathcal{I}}$, let $V(f_i)$ be the current version set associated with fact f_i. The left hand side of the hypothesis,

$$(\forall i : 1 \le i \le n)\ f_i \in \hat{\mathcal{I}}(E, P)|_v \cup E|_v \tag{8}$$

indicates

$$(\forall i : 1 \le i \le n)\ v \in V(f_i). \tag{9}$$

Thus, we have,

$$v \in \bigcap_{1 \le i \le n} V(f_i). \tag{10}$$

According to the lifted inference shown in Fig. 11, we can derive a new fact from Eq. (8):

$$(f_0, \bigcap_{1 \le i \le n} V(f_i)) \in \hat{\mathcal{I}}(E, P) \tag{11}$$

Combining Eq. (10) and Eq. (11), $f_0 \in \hat{\mathcal{I}}(E, P)|_v$.

According to the semantics of rule in $\mathcal{I}$ (the un-lifted Datalog), the right hand side of the hypothesis indicates

$$(\forall i : 1 \le i \le n)\ f_i \in \mathcal{I}(E|_v, P) \cup E|_v \implies f_0 \in \mathcal{I}(E|_v, P).$$

□