Temporal Data Model for Program Debugging

Demian Lessa
CSE Department
SUNY at Buffalo
dlessa@buffalo.edu

Jan Chomicki
CSE Department
SUNY at Buffalo
chomicki@buffalo.edu

Bharat Jayaraman
CSE Department
SUNY at Buffalo
bharat@buffalo.edu

ABSTRACT

We present a novel approach to program debugging and dynamic program analysis based on a temporal data model and query language. The data model exposes a point-based view of executions, while the underlying data is represented using intervals, for efficiency reasons. The point-based query language supports selection, projection, joins, grouping/aggregation, and recursion on both data and temporal attributes. We present examples of debug and dynamic analysis queries to illustrate our approach. The main technical contribution of the paper lies in showing how to evaluate point-based recursive queries against the interval-based data. In order to evaluate a non-recursive point-based query, it is first compiled into an interval-based query. The compilation relies on a normalization operation to preserve point-based semantics when evaluating compiled queries against interval-based data. It turns out that a straightforward extension of the compilation to handle recursion yields non-linear recursive queries with non-stratified negation. We circumvent this problem by employing a simpler compilation and introducing a normalizing immediate-consequence operator for bottom-up evaluation. We also present correctness and termination theorems for our temporal recursive query evaluation strategy. This work forms part of a larger research project in developing JIVE, a state-of-the-art dynamic analysis and visualization system for Java.

1. INTRODUCTION

The prevalent run-time debugging model today relies on techniques in use since the early 60s [11, 10]: setting breakpoints, inspection of variables in the call stack, forward stepping, and printing statements. Most current debuggers provide access only to the current state of execution, i.e., call stacks and heap objects. Thus, when a breakpoint is reached, the programmer has to proceed in a procedural, step-by-step fashion in search for the error. Often the cause lies in a method call already completed or a value already modified by the execution, and the process must restart with a new set of breakpoints.

A program execution is naturally represented as a chronological sequence of events, each occurring at a discrete point in time and encapsulating one or more changes to the execution state, e.g., variable assignment, method call, object creation, etc. We propose that debuggers record event sequences, from which execution states can be constructed, stored, and accessed for any time point. Clearly, storing execution states explicitly for every time point is impractical. Internally, the debugger should utilize a space-efficient representation, typically in the form of an interval-based encoding of the execution states, e.g., a variable holds a fixed value over each interval spanning consecutive assignments. In this framework, it would be possible to answer questions such as:

1. When (at which instants) did variable v change value?
2. Was there a concurrent update of an object in the program?
3. Was object o_2 ever reachable from object o_1?
4. How did the length of a linked list change during execution?

These questions express *temporal properties* that are hard, if not impossible, to answer using traditional debuggers. Mechanically traversing an execution trace file is unrealistic: a basic query such as the first one would require traversing the entire trace file which, even for simple programs, can be quite large. If a relational database is used to record executions, it would be possible to answer some questions using SQL. However, because SQL is not a temporal query language, queries would have to explicitly incorporate temporal semantics, a complex and error prone task [15]. In order to answer questions such as the ones above, we argue that a debugger should support *declarative temporal queries* over the point-based view of executions and provide an *efficient mechanism for their evaluation* against the interval-based representation.

Evaluation of point-based queries against an interval-based representation is typically achieved by means of compilation: a point-based temporal query is compiled into a corresponding interval-based one that preserves point-based semantics during evaluation. SQL/TP [18] is a point-based temporal query language supporting such an evaluation strategy for non-recursive queries. In SQL/TP, a normalization operation applied as part of compilation guarantees the preservation of point-based semantics, i.e., all time-intervals for some data item are either disjoint or identical.

The main technical contribution of this paper lies in showing how to evaluate point-based recursive queries against an interval-based representation. It turns out that a straightforward extension of the SQL/TP compilation to handle recursion yields non-linear queries with non-stratified negation. To circumvent this problem, we adopt a simpler compilation strategy and introduce a normalizing immediate consequence operator for bottom-up evaluation. We also present correctness and termination theorems for our temporal recursive query evaluation strategy.

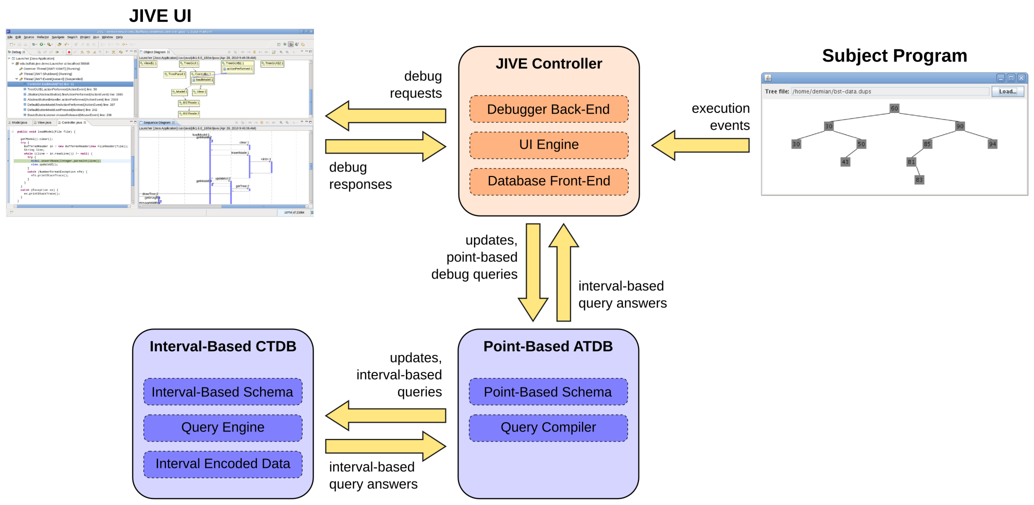

Figure 1: JIVE System Architecture.

In this paper, we introduce a principled approach to debugging and dynamic analysis based on the declarative investigation of executions and supported by a sound temporal framework. We present the temporal data model and query language, and illustrate them with examples. We also show how to evaluate point-based recursive queries against an interval-based representation. This work forms part of a larger research project aimed at developing a state-of-the-art dynamic analysis and visualization system for Java, called JIVE [5] (`http://www.cse.buffalo.edu/jive`). In JIVE, declarative queries and visualizations work in a synergistic manner– the results of queries are reported on (run-time) object and sequence diagrams, while queries help focus on relevant diagram regions. Figure 1 shows JIVE's overall system architecture.

The rest of this paper is organized as follows: Section 2 surveys closely related work. Section 3 describes our temporal model and illustrates temporal queries for debugging. Section 4 describes the recursive extension to the temporal query language. Section 5 concludes and provides directions for further work.

2. RELATED WORK

Typically, trace-based tools work by collecting low-level events during program execution, storing them in a file or database, and providing some form of analysis over the trace. Whyline [7] is a debugger supporting 'why did' and 'why did not' queries exposed to users in the form of a list of natural language questions. The tool uses dynamic slicing [1] to precompute queries and their answers. Omniscient debugging [10] is a trace-based technique that supports navigation through an event history and inspection of previous states. Omniscient debuggers often integrate additional features such as program visualizations and basic queries. For example, TOD [13] is an omniscient debugger featuring query-based debugging based on low-level primitives (*cursor* and *count*) used to construct high-level queries algorithmically. TOD's query language is neither declarative nor temporal, and their database is custom built from scratch for scalability.

Some query-based debuggers do not explicitly store execution traces. PTQL [6] is a relational query language supporting conjunctive queries over traces consisting of method invocations and object allocations. Queries are defined prior to execution and are evaluated online by specialized code instrumented into the subject program. Coca [4] is an automated debugger for C that allows setting event-based breakpoints in the form of Prolog-like queries. When an event match is detected, the program suspends and the developer can query current program states. In [9], queries are formulated in the programming language itself and evaluated against the current state of execution, and may have side-effects.

None of the approaches above support temporal queries, therefore, temporal semantics must be incorporated manually by the user during query formulation, a task that is both challenging and error prone. Further, none of the declarative query languages support recursion, a limitation that affects reasoning over the run-time realization of complex structures (e.g., lists, trees) and limits their applicability for dynamic analyses.

Our unique contribution over prior work lies in combining a formal temporal model for representing and querying executions, separating the conceptual (point-based) view of executions from its (interval-based) representation, and using a highly expressive declarative temporal query language. We claim that this principled approach simplifies debugging as well as the task of specifying and testing new analyses– it is simply a matter of writing a new query.

Declarative query languages have been successfully used in static program analysis [8, 12]. For example, bddbddb [8] allows static analyses to be specified as Datalog programs. Relations are stored using binary decision diagrams (BDDs), which enables bddbddb to efficiently handle exponential relations. Our declarative, query-based approach to dynamic program analysis is closely related to theirs both in design and philosophy.

3. TEMPORAL MODEL

A temporal database is a structure supporting some built-in aspect of time. It normally consists of a *temporal data model* specifying the data and temporal domains supported by the database, and a *temporal query language* for retrieving and modifying data within the database. Restrictions to the supported queries may be necessary to guarantee that query answers are finitely representable.

Temporal databases usually represent time values either as *points* or *intervals*. A point-based temporal data model allows for a temporal query language with clean, declarative syntax and clear semantics [17], at the expense of a more verbose data representation. The main advantage of an interval-based data model lies in its space-efficient representation of temporal facts. However, interval-based query languages are subject to a number of prob-

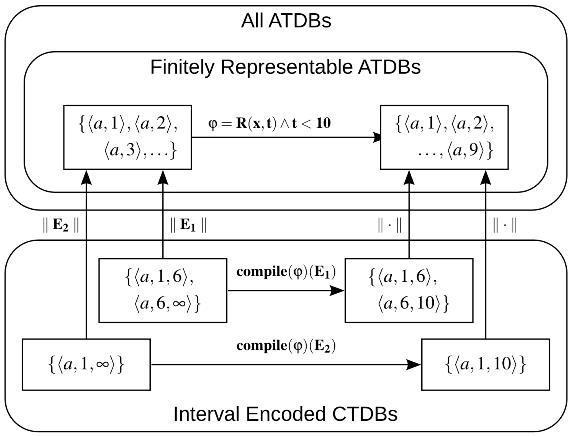
All ATDBs
Finitely Representable ATDBs
$\{\langle a,1\rangle,\langle a,2\rangle, \langle a,3\rangle,\ldots\}$
$\varphi = \mathbf{R}(\mathbf{x},\mathbf{t}) \wedge \mathbf{t} < 10$
$\{\langle a,1\rangle,\langle a,2\rangle, \ldots,\langle a,9\rangle\}$
$\| \mathbf{E_2} \|$
$\| \mathbf{E_1} \|$
$\| \cdot \|$
$\| \cdot \|$
$\{\langle a,1,6\rangle, \langle a,6,\infty\rangle\}$
compile(φ)(E1)
$\{\langle a,1,6\rangle, \langle a,6,10\rangle\}$
$\{\langle a,1,\infty\rangle\}$
compile(φ)(E2)
$\{\langle a,1,10\rangle\}$
Interval Encoded CTDBs

Figure 2: Evaluation of point-based abstract temporal queries.

lems: syntax dependence on the choice of interval encoding [17], necessity of data coalescing to guarantee correctness of results [2], representation-dependent queries [18], etc. Additionally, formally defining the semantics of interval-based languages is a complex task [16], largely due to their semantics being normally defined with respect to a point-based temporal model.

There are significant benefits in separating the conceptual temporal data from its representation [2]. Following this approach, an *abstract temporal database* (ATDB) describes temporal information in a representation-independent manner, normally using a point-based temporal data model, while a *concrete temporal database* (CTDB) provides a storage-efficient encoding of the ATDB, typically by means of an interval-based temporal data model. A semantic mapping $\| \cdot \|$ provides a formal interpretation of concrete temporal databases (also relations and tuples) in terms of the respective abstract ones.

Evaluation of an abstract query Q is carried out by compiling it into the concrete query language and evaluating the resulting query against the CTDB, as illustrated in Figure 2. In the figure, $\mathbf{E_1}$ and $\mathbf{E_2}$ are distinct encodings of the abstract temporal relation $\mathbf{R}$. To evaluate the point-based abstract temporal query φ against $\mathbf{E_1}$ and $\mathbf{E_2}$, φ is compiled and evaluated against each encoding, yielding concrete relations $compile(\varphi)(E_1)$ and $compile(\varphi)(E_2)$. These are clearly distinct relations but their images under $\| \cdot \|$ are identical. Hence, they *represent* the same abstract temporal relation.

Compilation must satisfy two requirements. The first, *semantics preservation*, states that answers to the compiled query must represent the exact same answers that would be obtained by executing Q against the ATDB, that is, for every CTDB D, $Q(\| D \|) = \| compile(Q)(D) \|$. The second, *closure of representation*, establishes that the set of time instants associated with every tuple in the answer to the compiled query must be finitely representable using the encoding of the CTDB. We note that existing compilation-based approaches [17, 18, 3] do not support recursive queries.

JIVE's Temporal Model. Our system uses a point-based ATDB to represent executions at a conceptual level and an interval-based CTDB for space-efficient storage. In this section we present our temporal schema and the features of our temporal query language, PRACTQL. We delay the technical developments regarding the evaluation of temporal recursive queries to next section. Due to limited space, queries are presented throughout the paper using Datalog notation. The actual syntax of the query language, however, is SQL.

JIVE's temporal schema exposes program entities (i.e., at the source language level), their temporal states, and lower-level entities such as trace events. Therefore, users can formulate queries in terms of the concepts that are most adequate for each task or that they are most comfortable with.

Example 1 (Concurrency). Temporal queries can help identify and resolve concurrency errors as well as improve the design of concurrent programs. For instance, consider the problem of finding all pairs of method calls (m1 and m2) that modify the same field (f) of the same object instance (o) while their executions overlap in time. The modifications could indicate a race condition but they could also be protected by an appropriate concurrency protocol (e.g., locking).

Relations Activation(C,TH,CC,M,T) and EventBind(C,O,F,V,T) represent method activations and field assignments, respectively. Activation records every instant T in which call C to method M on thread TH with calling context CC (i.e., type/instance on which the call is made) is active. EventBind records the instant T in which field F of object O is assigned value V in the context of activation C.

$$\begin{aligned} &\text{Concurrent(m1, m2, o, f)} :- \text{Activation(c1, th1, \_, m1, t2)},\\ &\quad \text{EventBind(c1, o, f, v1, t1), Activation(c2, th2, \_, m2, t1)},\\ &\quad \text{EventBind(c2, o, f, v2, t2)}, \text{th1} \neq \text{th2} \end{aligned}$$

Figure 3 provides a visual interpretation for the query: the activations are concurrent for at least the interval delimited by the assignments on the object/field (shaded region). □

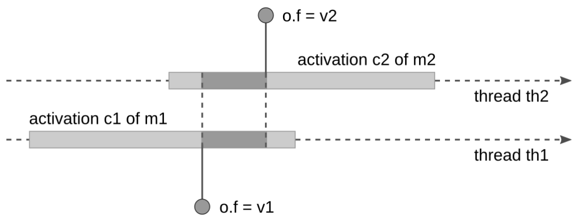

Figure 3: Concurrent modifications.

A large class of debug questions are expressible as temporal SPJ queries such as the one above. For instance, the importance of understanding the who-calls-whom relation (e.g., does method m call method n transitively?) in large, object-oriented programs is described as a non-trivial task in [6]. Such queries are expressible as temporal SPJ queries with negation in our query language. Another relevant class of questions are those involving *aggregation*. For instance, in order to determine how the length of a linked list changes during execution, a query must group by time instants and count the number of nodes in the list.

Recursive queries are particularly important in answering several program understanding, debugging, and dynamic analysis questions. Programs often define recursive data structures, such as lists, trees, and graphs. Several questions about their run-time realization are naturally expressed as recursive queries. Further, the run-time state of a program is typically represented as a graph, so the analysis of program sate requires answering queries such as *temporal reachability*: was object o_2 ever reachable from object o_1? It is important to note that, although none of the data structures mentioned above are first-class citizens in the relational model, they can be encoded as facts in the temporal database. Example 2 illustrates the use of recursion in analyzing the state of run-time objects.

Example 2 (State Analysis). To answer questions about a binary tree, we must traverse its left and right subtrees recursively while observing the temporal relationship among the tree nodes.

For instance, consider the following problem: when were the values 10 and 20 found *simultaneously* among the nodes of a binary tree whose root node has $oid = 5$?

Let TreeNode(O,V,L,R,T) be a relation representing TreeNode instances in the database. It records every instant T during which the node with object identifier O (*oid*) has value V and left and right subtrees identified by L and R, respectively. The answer to the problem above can be expressed as follows:

```
Q(t) :− Path(5, d1, t), TreeNode(d1, 10, _, _, t),
        Path(5, d2, t), TreeNode(d2, 20, _, _, t)
Path(a, d, t) :− TreeNode(a, _, d, _, t)
Path(a, d, t) :− TreeNode(a, _, _, d, t)
Path(a, d, t) :− Path(a, n, t), TreeNode(n, _, d, _, t)
Path(a, d, t) :− Path(a, n, t), TreeNode(n, _, _, d, t)
```

Path is a recursive query that asserts the existence of a path from node a to node d at time t. The temporal equijoin guarantees that each ancestor-descendant relation in the path holds over the projected time. Q returns all time instants during which the root node has descendants with the specified values. □

Dynamic slicing [1] is an important debugging technique in the area of dynamic program analysis. Intuitively, a dynamic slice can be computed from a transitive closure of dynamic data and/or control dependencies in a program. This is a very natural application of temporal recursion in the context of debugging, as illustrated in Example 3.

Example 3 (Dynamic Reaching Definitions). Consider the relation Defs(V1,V2,T) that keeps track of variable assignments at runtime: variable V1 is assigned a value obtained from the evaluation of an expression involving variable V2 at time T. Query RDefs computes the set of dynamic reaching definitions in an execution.

```
RDefs(x, y, t) :− Defs(x, y, t)
RDefs(x, y, t) :− RDefs(z, y, t1), ¬ Defs(z, _, t2), Defs(x, z, t),
                  t1 < t2, t2 < t
```

The first rule states that x is defined by y at time t. The second rule states that x is defined by y at time t if z is defined by y and not redefined before being used to define x at time t. □

Our temporal query language supports all of the aforementioned features and more. The technical developments necessary to incorporate recursion into the language are discussed in the next section.

4. RECURSIVE QUERY EVALUATION

PRACTQL is a point-based temporal query language that extends SQL/TP [18] with support for recursive queries. Its semantics is defined over abstract temporal databases and it supports queries involving selection, projection, joins, set operations, bag operations with finite duplication, grouping and aggregation on both data and temporal attributes. Closure of representation is guaranteed by a syntactic restriction, namely, top-level attribute independence.

PRACTQL extends the SQL data model with the temporal sort $(\mathbb{Z}, \leq)$ to represent individual time points. Non-empty half-open intervals from $\mathbb{Z} \cup \{-\infty, +\infty\}$ are used in the concrete encoding: each abstract temporal attribute t is encoded as a concrete attribute $[t_\ell, t_r)$ of the interval sort. Interval endpoint comparisons use order relations based on the order inherited from the temporal sort. Like SQL/TP, PRACTQL may represent infinite relations, therefore, duplicate preserving projection of temporal attributes encoded by intervals is disallowed in order to avoid space blowup.

Query Compilation. SQL/TP queries are compiled into SQL/92, which allows evaluation using a standard relational database management system. For efficient evaluation, compiled queries reference only values in the active domain and possibly small neighborhoods of time points. Set and bag operations are performed on *time-compatible queries* [18] in order to guarantee preservation of semantics with respect to ATDBs. Intuitively, concrete queries Q_i and Q_j are time-compatible if they have compatible schemas and, for every CTDB D, whenever tuples in $Q_i(D)$ and $Q_j(D)$ agree on their data components, the intervals related to each tuple coincide or are disjoint, i.e., behave like points with respect to set and bag operations. Time-compatibility is achieved by the application of a *normalization* [18] operation on input queries.

DEFINITION 1 (NORMALIZATION). *Given a set $\mathbb{Q}$ of concrete queries with a common schema, a set $\mathbf{x}$ of their data attributes, and a temporal attribute t in their common schema, the normalization of $Q_i \in \mathbb{Q}$ with respect to $\mathbf{x}$ and t is the query $N^t_{\mathbf{x}}(Q_i, \mathbb{Q})$ satisfying, for every CTDB D, the following properties: (i) for every $Q_j \in \mathbb{Q}$, whenever tuples in $N^t_{\mathbf{x}}(Q_i, \mathbb{Q})(D)$ and $N^t_{\mathbf{x}}(Q_j, \mathbb{Q})(D)$ agree on their $\mathbf{x}$ values, the intervals encoded by t in each tuple either coincide or are disjoint and (ii) $\| Q_i(D) \| = \| N^t_{\mathbf{x}}(Q_i, \mathbb{Q})(D) \|$.*

Property (i) states that intervals returned by normalized queries behave like points with respect to set and bag operations and property (ii) states that normalized queries preserve meaning with respect to ATDBs.

Example 4 (SQL/TP Compilation). The abstract temporal relation Refs(O, R, T) keeps track of all instants T during which object O references object R. The recursive query below computes a temporal transitive closure of Refs:

$$\mathsf{TTC}(x,y,t) :- \ \mathsf{Refs}(x,y,t)$$
$$\mathsf{TTC}(x,y,t) :- \ \mathsf{TTC}(x,z,t_1), \mathsf{Refs}(z,y,t), t > t_1$$

A SQL/TP compilation implementing normalization as a first-order query would produce the concrete query below.

$$\overline{\mathsf{TTC}}(x,y,\ell,r) :- \ \overline{\mathsf{TTC}_1}(x,y,t_\ell,t_r), \mathsf{Imin}(x,y,\ell,r), t_\ell \leq \ell, r \leq t_r$$
$$\overline{\mathsf{TTC}}(x,y,\ell,r) :- \ \overline{\mathsf{TTC}_2}(x,y,t_\ell,t_r), \mathsf{Imin}(x,y,\ell,r), t_\ell \leq \ell, r \leq t_r$$
$$\overline{\mathsf{TTC}_1}(x,y,\ell,r) :- \ \overline{\mathsf{Refs}}(x,y,\ell,r)$$
$$\overline{\mathsf{TTC}_2}(x,y,\ell,r) :- \ \overline{\mathsf{TTC}}(x,z,\ell_1,r_1), \overline{\mathsf{Refs}}(z,y,\ell_2,r), r > \ell_1 + 1,$$
$$\ell = max(\ell_2, \ell_1 + 1)$$
$$\mathsf{Imin}(x,y,\ell,r) :- \ \mathsf{EP}(x,y,\ell), \mathsf{EP}(x,y,r), \ell < r, \neg\mathsf{NImin}(x,y,\ell,r)$$
$$\mathsf{NImin}(x,y,\ell,r) :- \ \mathsf{EP}(x,y,\ell), \mathsf{EP}(x,y,t), \mathsf{EP}(x,y,r), \ell < t, t < r$$
$$\mathsf{EP}(x,y,\ell) :- \ \overline{\mathsf{TTC}_1}(x,y,\ell,r)$$
$$\mathsf{EP}(x,y,r) :- \ \overline{\mathsf{TTC}_1}(x,y,\ell,r)$$
$$\mathsf{EP}(x,y,\ell) :- \ \overline{\mathsf{TTC}_2}(x,y,\ell,r)$$
$$\mathsf{EP}(x,y,r) :- \ \overline{\mathsf{TTC}_2}(x,y,\ell,r)$$

An overbar indicates a concrete temporal relation or predicate. Imin, NImin, and EP are helper predicates for the normalization. The first and second $\overline{\mathsf{TTC}}$ rules correspond to $N^t_{x,y}(\overline{\mathsf{TTC}_1}, \{\overline{\mathsf{TTC}_1}, \overline{\mathsf{TTC}_2}\})$ and $N^t_{x,y}(\overline{\mathsf{TTC}_2}, \{\overline{\mathsf{TTC}_1}, \overline{\mathsf{TTC}_2}\})$, respectively.

The concrete query above is obtained as follows. First, TTC rules are compiled into $\overline{\mathsf{TTC}_1}$ and $\overline{\mathsf{TTC}_2}$. Normalization produces time-compatible rules for $\overline{\mathsf{TTC}}$ by joining $\overline{\mathsf{TTC}_1}$ (also $\overline{\mathsf{TTC}_2}$) with Imin. Intuitively, the join with Imin partitions the interval of each tuple returned by $\overline{\mathsf{TTC}_1}$ (also $\overline{\mathsf{TTC}_2}$) into a set of minimal subintervals. Imin constructs minimal intervals based on non-minimal intervals returned by NImin. All intervals are constructed from interval endpoints in EP, which contains all left and right interval endpoints obtained from $\overline{\mathsf{TTC}_1}$ and $\overline{\mathsf{TTC}_2}$. □

Normalization can be extended to handle concrete relations as follows. Given a set $\mathbb{R}$ of concrete relations with a common signature, a subset $\mathbf{x}$ of their data attributes, and a temporal attribute t, the normalization of $R_i \in \mathbb{R}$ with respect to $\mathbf{x}$ and t is the relation $N^t_{\mathbf{x}}(R_i, \mathbb{R})$ satisfying: (i) for every $R_j \in \mathbb{R}$, whenever tuples in $N^t_{\mathbf{x}}(R_i, \mathbb{R})$ and $N^t_{\mathbf{x}}(R_j, \mathbb{R})$ agree on their $\mathbf{x}$ values, the intervals encoded by t in each tuple either coincide or are disjoint, and (ii) $\parallel R_i \parallel = \parallel N^t_{\mathbf{x}}(R_i, \mathbb{R}) \parallel$.

To define a set of time-compatible queries (relations) with respect to $\mathbf{x}$ and $\mathbb{Q}$ ($\mathbb{R}$), normalization is applied for every temporal attribute in the common schema. The $\mathsf{N}_{\mathbf{x}}$ operator designates such application. By convention, the subscript can be omitted when normalizing with respect to all data attributes. Hence, a set operation on concrete queries $Q_1\ op\ Q_2$, where $op \in \{\cup, \cap, -\}$, is compiled into a corresponding set operation on time-compatible queries, $\mathsf{N}(Q_1, \{Q_1, Q_2\})\ op\ \mathsf{N}(Q_2, \{Q_1, Q_2\})$, which we denote as $Q_1\ op^{\mathsf{N}}\ Q_1$ for convenience. Normalizing set operations for concrete relations are defined analogously. The N operator is also used to define bag operations, grouping, and duplicate elimination.

As evidenced by Example 4, the SQL/TP compilation interacts with recursive queries in a way that introduces fundamental difficulties. Recursion on $\overline{\mathsf{TTC_2}}$ is non-linear, since $\overline{\mathsf{TTC_2}}$ references $\overline{\mathsf{TTC}}$ and $\overline{\mathsf{TTC}}$ references $\overline{\mathsf{TTC_2}}$ multiple times– once directly and multiple times indirectly, through Imin. Further, the negation introduced in the body of Imin is non-stratified: the dependence graph of the compiled query contains the cycle ($\overline{\mathsf{TTC}}$, $\overline{\mathsf{TTC_2}}$, EP, NImin, Imin, $\overline{\mathsf{TTC}}$) where (NImin, Imin) is a negative edge. Hence, a standard SQL:1999 (or later) query engine cannot evaluate such query.

PRACTQL Compilation. The problems discussed above have a common cause, namely, the introduction of the join with Imin by the normalization step of the compilation. Unfortunately, there is no way to avoid the use of negation if normalization is implemented as a first-order query– negation is necessary to guarantee minimality of the intervals returned by time-compatible queries.

In order to address this issue, the PRACTQL compiler omits the normalization step for recursive rules, where it introduces difficulties. As a result, given an input recursive query valid with respect to the SQL:1999 standard, the PRACTQL compiler produces a recursive query that preserves such validity and, therefore, can be evaluated by a standards compliant SQL:1999 query engine.

Example 5 (PRACTQL Compilation). Assume that we compile the query of Example 4 using the PRACTQL compilation. After simplification to remove subqueries, the resulting query is:

$$\overline{\mathsf{TTC}}(x,y,\ell,r) :- \ \overline{\mathsf{Refs}}(x,y,\ell,r)$$
$$\overline{\mathsf{TTC}}(x,y,\ell,r) :- \ \overline{\mathsf{TTC}}(x,z,\ell_1,r_1), \overline{\mathsf{Refs}}(z,y,\ell_2,r), r > \ell_1 + 1, \ell = max(\ell_2, \ell_1 + 1)$$

Assume that $\overline{\mathsf{Refs}}$ contains a single tuple $(1,1,1,2^k+1)$, $k \geq 0$. The tuple encodes the fact that object $O = 1$ has a self reference during $[1, 2^k+1)$. Bottom-up evaluation of $\overline{\mathsf{TTC}}$ using the standard immediate consequence operator, T_P, produces the following results:

$T_P \uparrow 1$: $\{(1,1,\mathbf{1},\mathbf{2^k+1})\}$
$T_P \uparrow 2$: $\{(1,1,\mathbf{1},\mathbf{2^k+1}), (1,1,\mathbf{2},\mathbf{2^k+1})\}$
...
$T_P \uparrow 2^k$: $\{(1,1,\mathbf{1},\mathbf{2^k+1}), (1,1,\mathbf{2},\mathbf{2^k+1}), \ldots, (1,1,\mathbf{2^k},\mathbf{2^k+1})\}$

The evaluation terminates at stage 2^k+1, with 2^k tuples. □

As Example 5 illustrates, standard bottom-up evaluation [20] of compiled queries is problematic. First, set operations performed at each iteration do not preserve semantics with respect to ATDBs, e.g., tuples $(1,1,1,2^k+1)$ and $(1,1,2,2^k+1)$ in $T_P \uparrow 2$ represent duplicate information, i.e., $\parallel (1,1,1,2^k+1) \parallel \supset \parallel (1,1,2,2^k+1) \parallel$. Second, evaluation suffers from an exponential blowup in space complexity since it returns 2^k tuples for an input relation instance containing a single tuple. Moreover, the blowup depends on the particular query and database instance against which evaluation is performed– if $\overline{\mathsf{Refs}}$ contained tuple $(1,1,1,+\infty)$, evaluation would simply not terminate.

The first problem is caused by evaluating the set union over queries that are not time-compatible. The second, by using a termination condition that is incapable of identifying concrete relations that are equivalent with respect to their abstract interpretations (i.e., their images under $\parallel \cdot \parallel$) but not their concrete representations. The third, by this same reason but augmented by the fact that concrete relations can encode infinite abstract temporal relations. Unfortunately, it is not possible to address these issues at the query compilation level. Instead, we must modify the bottom-up evaluation.

Normalizing Bottom-up Evaluation. As observed, bottom-up evaluation using the standard immediate consequence operator does not preserve temporal semantics with respect to ATDBs. To overcome this limitation, we define a *normalizing immediate consequence operator*, T_{NP}, that enforces such semantics by incorporating normalization as part of the evaluation of recursive rules.

DEFINITION 2 (T_{NP}). *Let P be a recursive query obtained from the* PRACTQL *compilation. The normalizing immediate consequence operator, denoted by T_{NP}, is defined as follows:*

$T_{NP}(I) = \bigcup_{r_i \in P} T_{NP}(r_i, I)$, *where*

$$T_{NP}(r_i, I) = \begin{cases} T_P(r_i, I) \bigcap^{\mathsf{N}} \mathcal{U}_R(I) & r_i \text{ defines recursive predicate } R \\ T_P(r_i, I) & \text{otherwise} \end{cases},$$

$T_P(r_i, I) = \{A : A \leftarrow B$ *is a ground instance of* r_i *and* $B \subseteq I\}$,

$\mathcal{U}_R(I) = \bigcup_{r_i \text{ defines } R} T_P(r_i, I)$,

and $\bigcap^{\mathsf{N}}$ *is the normalizing set intersection.*

T_{NP} is identical to T_P for every rule that is not part of the definition of a recursive predicate. On the other hand, consider the set of rules $\{r_1, \ldots, r_n\}$ defining a recursive predicate R. T_{NP} evaluates each of the rules using T_P and then performs a normalizing set intersection of $T_P(r_i, I)$ and $\mathcal{U}_R(I)$. While the intersection modifies the concrete representation of $T_P(r_i, I)$, its image under $\parallel \cdot \parallel$ is preserved, that is, $\parallel T_P(r_i, I) \parallel = \parallel T_P(r_i, I) \bigcap^{\mathsf{N}} \mathcal{U}_R(I) \parallel$. The soundness and completeness of the T_{NP} operator is established in Theorem 1.

THEOREM 1. *[Soundness and Completeness] Let P be a recursive* PRACTQL *query and NP the concrete query obtained from the* PRACTQL *compilation. Then,* $\parallel T_{NP} \uparrow \omega \parallel = T_P \uparrow \omega$.

Intuitively, Theorem 1 establishes an equivalence between the evaluation of T_{NP} over CTDBs and T_P over ATDBs. The proof follows from the correctness of T_P, the soundness and completeness of SQL/TP [18], and the preservation of temporal semantics with respect to ATDBs guaranteed by T_{NP} at every stage. We observe, however, that simply replacing T_P for T_{NP} in the standard bottom-up evaluation is not sufficient to guarantee termination.

Example 6 (T_{NP}). We evaluate $\overline{\mathsf{TTC}}$ of Example 5 using T_{NP} and the standard bottom-up evaluation termination condition:

$T_P \uparrow 1$: $\{(1,1,\mathbf{1},\mathbf{2^k+1})\}$
$T_P \uparrow 2$: $\{(1,1,\mathbf{1},\mathbf{2}), (1,1,\mathbf{2},\mathbf{2^k+1})\}$
$T_P \uparrow 3$: $\{(1,1,\mathbf{1},\mathbf{2}), (1,1,\mathbf{2},\mathbf{3}), (1,1,\mathbf{3},\mathbf{2^k+1})\}$
...
$T_P \uparrow 2^k$: $\{(1,1,\mathbf{1},\mathbf{2}), (1,1,\mathbf{2},\mathbf{3}), \ldots, (1,1,\mathbf{2^k},\mathbf{2^k+1})\}$

Temporal semantics with respect to $\parallel \cdot \parallel$ is clearly preserved. □

```
Normalizing−Naive(P)
  I := Ø;
  repeat
    J := I;
    I := $T_{NP}$(I);
  until $\| I \| = \| J \|$;
  return I;
```

Listing 1: Normalizing Naïve bottom-up evaluation.

As Example 6 illustrates, when the bottom-up evaluation reaches a stage with $\| I \| = \| J \|$, it is possible that $I \neq J$. Therefore, termination of the bottom-up evaluation should be based on the equivalence of the images of I and J under $\| \cdot \|$. Obviously, it is impossible to perform the equivalence check on abstract temporal relations as they may be infinite. Even if they were finite, as is the case in program debugging, the test could cause a space blowup in the evaluation (cf. Example 6). Instead, we perform a check on concrete relations as follows: for every recursive predicate R in P, we test whether $R_I -^{\mathbf{N}} R_J = R_J -^{\mathbf{N}} R_I = \emptyset$, where R_I and R_J denote the instances of R in I and J, respectively, and $-^{\mathbf{N}}$ guarantees that set semantics is preserved with respect to ATDBs. Our normalizing bottom-up evaluation is presented in Listing 1. Example 7 shows the algorithm in action and termination is presented in Theorem 2.

THEOREM 2. *[Termination] Normalizing-Naïve terminates for every recursive query (under set semantics) produced by the* PRACTQL *compilation.*

In the standard relational setting, termination of the bottom-up evaluation relies on the finiteness of the input relations and the fact that new symbols are never introduced during the evaluation. In our setting, compilation may introduce new symbols via quantifier elimination (e.g., rule for $\overline{\mathsf{TTC}_2}$ in Example 4), which means that new temporal symbols could be introduced at every iteration of the bottom-up evaluation. Further, although concrete relations are finite, they may encode infinite abstract temporal relations, which would provide an infinite source of new temporal symbols. Therefore, termination of the normalizing bottom-up evaluation must rely on a different argument.

The proof is based on the reduction of Normalizing-Naïve to the bottom-up evaluation of constraint Datalog programs, which terminates for all constraint-compact classes of constraints [19]. The reduction is possible because the PRACTQL compilation produces queries that are essentially Datalog queries with integer gap-order constraints [14], a constraint-compact class of constraints.

Finally, we observe that the techniques used here to modify the standard bottom-up evaluation can also adapted to the semi-naïve bottom-up evaluation [20].

Example 7 (Normalizing Bottom-Up Evaluation). We evaluate $\overline{\mathsf{TTC}}$ of Example 5 for the concrete instance of Refs containing tuple $(1,1,1,+\infty)$:

$T_P \uparrow 1$: $\{(1,1,\mathbf{1},+\infty)\}$
$T_P \uparrow 2$: $\{(1,1,\mathbf{1},\mathbf{2}),(1,1,\mathbf{2},+\infty)\}$

$T_P \uparrow 2$ represents the same abstract temporal relation as $T_P \uparrow 1$. Hence, the termination condition is satisfied and the computation returns with two tuples. □

5. CONCLUSIONS

We described a point-based temporal data model for debugging that supports declarative temporal queries over program executions and transparently stores execution traces in a space-efficient manner. We showed that common debug questions can be adequately formulated in a declarative temporal query language and that recursion is of fundamental importance in answering important dynamic analysis problems. In order to support recursion, we extended a point-based query language with recursion and showed the necessary modifications to the standard bottom-up evaluation.

Future work will evaluate the performance of PRACTQL for a wide range of debug and dynamic analysis queries. We will investigate the scalability of temporal query evaluation in general and of the normalizing bottom-up evaluation in particular. For recursive queries, we will analyze the benefits of incorporating static and dynamic optimizations in the compilation procedure. We will also consider using domain-specific optimizations which take advantage of the temporal semantics of program executions.

The PRACTQL query language and compiler are currently being incorporated into JIVE to replace a simpler, interval-based temporal query language prototype previously supported by the tool. JIVE has been under development since 2007, has been tested on a large number of programs, and has been extensively used in both undergraduate and graduate programming language courses in our department.

Acknowledgment. This work was supported in part by a grant from the Fujitsu Foundation.

6. REFERENCES

[1] H. Agrawal and J. R. Horgan. Dynamic program slicing. In *Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*, PLDI '90, pages 246–256, New York, NY, USA, 1990. ACM.

[2] J. Chomicki and D. Toman. Temporal databases. In M. D. Fisher, D. M. Gabbay, and L. Vila, editors, *Handbook of Temporal Reasoning in Artificial Intelligence*, pages 429–467. Elsevier B. V., March 2005.

[3] J. Chomicki, D. Toman, and M. H. Böhlen. Querying ATSQL Databases with Temporal Logic. *ACM Trans. Database Syst.*, 26(2):145–178, 2001.

[4] M. Ducassé. Coca: An automated debugger for c. In *ICSE '99: Proceedings of the 21st international conference on Software engineering*, pages 504–513, New York, NY, USA, 1999. ACM.

[5] P. V. Gestwicki and B. Jayaraman. Methodology and architecture of jive. In *Proceedings of the 2005 ACM Symposium on Software Visualization (SoftVis'05)*, pages 95–104, May 2005.

[6] S. F. Goldsmith, R. O'Callahan, and A. Aiken. Relational queries over program traces. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 385–402, New York, NY, USA, 2005. ACM.

[7] A. J. Ko and B. A. Myers. Extracting and answering why and why not questions about java program output. *ACM Trans. Softw. Eng. Methodol.*, 20:4:1–4:36, September 2010.

[8] M. S. Lam, J. Whaley, V. B. Livshits, M. C. Martin, D. Avots, M. Carbin, and C. Unkel. Context-sensitive program analysis as database queries. In *Proceedings of the twenty-fourth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, PODS '05, pages 1–12, New York, NY, USA, 2005. ACM.

[9] R. Lencevicius, U. Hölzle, and A. K. Singh. Query-based debugging of object-oriented programs. In *OOPSLA '97:*

Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, pages 304–317, New York, NY, USA, 1997. ACM.

[10] B. Lewis. Debugging backwards in time. *CoRR*, cs.SE/0310016, 2003.

[11] H. Lieberman. The debugging scandal and what to do about it. *Commun. ACM*, 40:26–29, April 1997.

[12] M. Martin, B. Livshits, and M. S. Lam. Finding application errors and security flaws using pql: A program query language. *SIGPLAN Not.*, 40:365–383, October 2005.

[13] G. Pothier, E. Tanter, and J. Piquer. Scalable omniscient debugging. *SIGPLAN Not.*, 42(10):535–552, 2007.

[14] P. Z. Revesz. A Closed-Form Evaluation for Datalog Queries with Integer (Gap)-Order Constraints. *Theor. Comput. Sci.*, 116(1):117–149, 1993.

[15] R. T. Snodgrass. *Developing Time-Oriented Database Applications in SQL*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2000.

[16] A. U. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev, and R. Snodgrass, editors. *Temporal Databases: Theory, Design, and Implementation*. Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1993.

[17] D. Toman. Point vs. Interval-Based Query Languages for Temporal Databases (extended abstract). In *PODS '96: Proceedings of the fifteenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 58–67, New York, NY, USA, 1996. ACM.

[18] D. Toman. Point-Based Temporal Extension of SQL. In *DOOD '97: Proceedings of the 5th International Conference on Deductive and Object-Oriented Databases*, pages 103–121, London, UK, 1997. Springer-Verlag.

[19] D. Toman. Memoing evaluation for constraint extensions of datalog. *Constraints*, 2:337–359, January 1998.

[20] J. D. Ullman. *Principles of database and knowledge-base systems, Vol. I*. Computer Science Press, Inc., New York, NY, USA, 1988.