Temporal Data Model for Program Debugging

Demian Lessa   Jan Chomicki   Bharat Jayaraman

Department of Computer Science and Engineering
State University of New York, Buffalo

August 29, 2011

Current Debuggers

Debug approach.

- Breakpoints, stepping, variable inspection/modification, line printing.
- Access only to current state.
  - E.g., object heap and call stacks.
- Cause of errors often lies in past states.
  - E.g., completed method calls and already modified variables.
- Must restart the debug session!

$\Rightarrow$ Record execution histories.

Trace-Based Debuggers

Execution history.

- Typically, collection of low-level events.
- Provides (indirect) access to execution states.
- Traces are usually quite large even for small programs.
- Traversal of file-based traces is infeasible.

$\Rightarrow$ Declarative approach.

Trace-Based Debuggers

Declarative approach.

- Queries often involve temporal relationships:
  - When was an invariant first violated?
  - When did variable V change value?
  - Was there a concurrent update of an object in the program?
- Recursive queries involving time are particularly important:
  - Was object O_2 ever reachable from object O_1?
  - When were values 10 and 20 found among the nodes of a binary tree?
  - How did the length of a linked list L change during execution?
  - What are the dynamic reaching definitions of variable V?
- Problem with standard SQL.
  - Queries must incorporate temporal semantics explicitly.
  - In general, this is complex and error prone.
  - Recursive queries involving time may be invalid w.r.t. SQL.

$\Rightarrow$ Contribution: temporal data model and query language for debugging.

Next...

1. Temporal Data Model
2. Recursive Query Evaluation

Temporal Data Model

Point-based temporal model.

- Execution represented as a sequence of events.
- An event happens at a discrete time point and encodes a state change.
- Conceptually, each instant of execution is associated with a state.
- Query language with clean, declarative syntax and clear semantics.

$\Rightarrow$ Storing execution states for every time point is impractical.

Temporal Data Model

Interval-based temporal model.

- Efficent representation.
- Query language subject to a number of problems:
  - Syntax for constructing intervals and accessing interval endpoints.
  - Coalescing to guarantee correctness of results.
  - Representation-dependent queries.
  - Defining the semantics of the language is complex.

$\Rightarrow$ Separate the conceptual data from its representation.

Temporal Data Model

Abstract Temporal Database (ATDB).

- Actual interface with user.
- Point-based temporal model.
- Temporal information is representation-independent.

Concrete Temporal Database (CTDB).

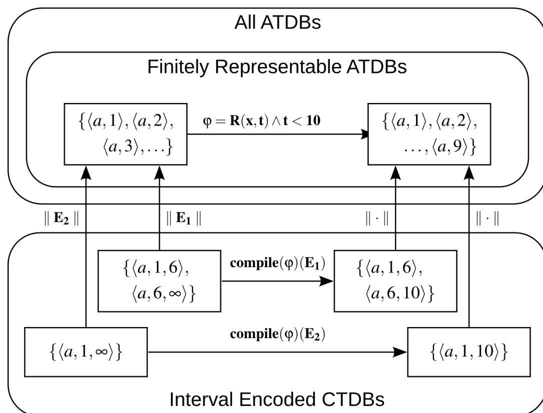
- Interval-based temporal model.
- Space-efficient encoding of the ATDB.
- A semantic mapping $\| \cdot \|$ associates CTDBs with abstract counterparts.

$\Rightarrow$ Cannot evaluate abstract queries directly.

Query Evaluation

An abstract query must be compiled prior to evaluation.

- The resulting concrete query is evaluated against the CTDB.
- Query answers must be finitely representable.
- The compilation must preserve semantics w.r.t. ATDBs.



Query Evaluation: Recursive Queries

Previous approaches did not address recursion involving time.

- Most relevant previous work: SQL/TP.
  - Only non-recursive point-based temporal queries.
- Straightforward extension of the SQL/TP compilation is problematic.
  - Yields non-linear, non-stratified recursive queries.

$\Rightarrow$ Contribution: evaluation of recursive queries involving time.

Overview of JIVE

JIVE is our research system: a visual, query-based debugger.

- Supports basic functionality.
  - Collection of low-level events from a subject program.
  - Inference of execution states from the event trace.
  - But no temporal data model.
- Provides useful applications.
  - Visualizations of event sequences and execution states.
  - Forward and reverse stepping/jumping (state replaying).
  - Simple form-based queries over the event trace.
  - But no temporal queries.

$\Rightarrow$ JIVE is incorporating the temporal data model and query language.

- More principled, query-based approach to debugging.
- Enables complex dynamic analyses such as dynamic slicing.
- Existing applications reimplemented as temporal queries.

Overview of JIVE: Debugger UI

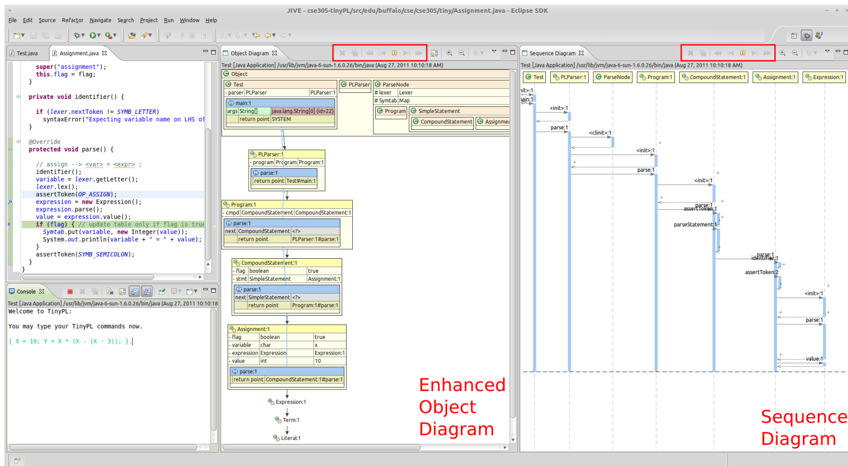


Figure: JIVE represents an event sequence as a sequence diagram (SD) and each execution state as an object diagram (OD). Forward and reverse stepping/jumping synchronize the OD view accordingly.

Overview of JIVE: Sequence Diagram

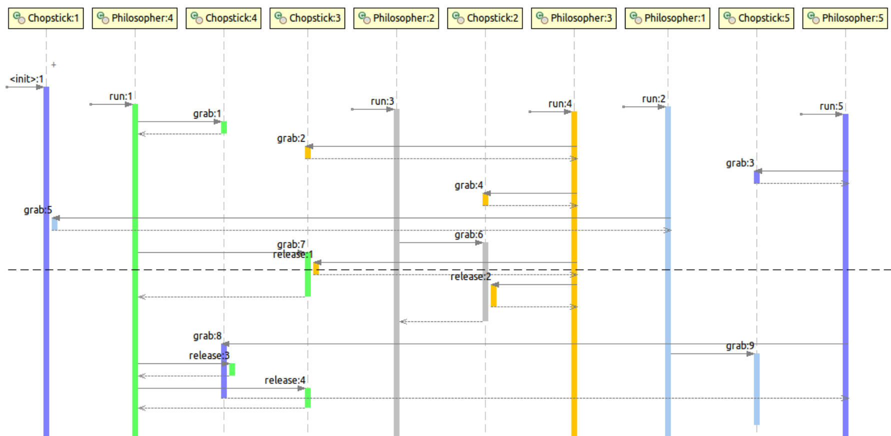


Figure: SDs display method calls per object/class (life lines) and thread. The dashed line indicates the temporal context used to synchronize the OD view.

Overview of JIVE: Object Diagram

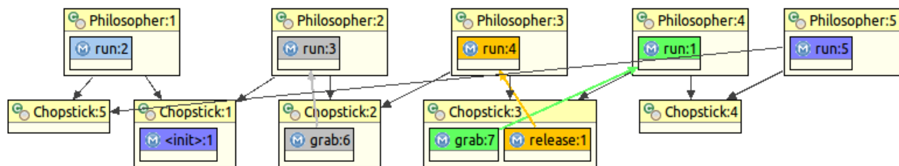


Figure: ODs represent activations within their respecitve object/class contexts, which is useful for clarifying concurrent behavior.

Overview of JIVE: Object Diagram

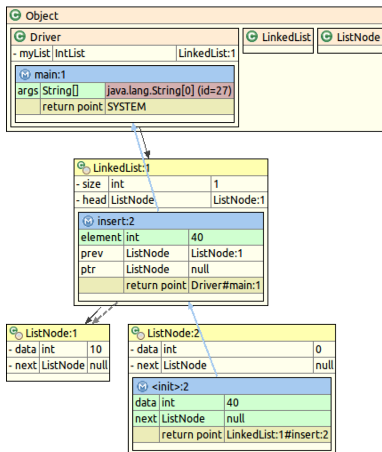


Figure: ODs represent one state of execution: classes, objects, method activations (i.e., stack frames), object relationships, and call stacks. The amount of information displayed is configurable.

Overview of JIVE: Queries

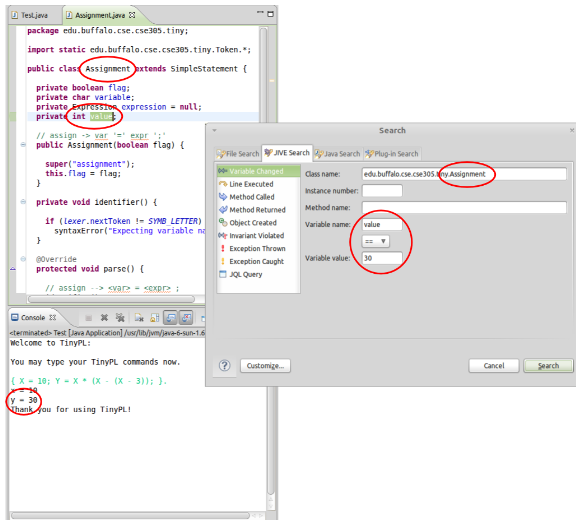


Figure: JIVE supports form-based queries, which execute against the event trace.

Overview of JIVE: Queries

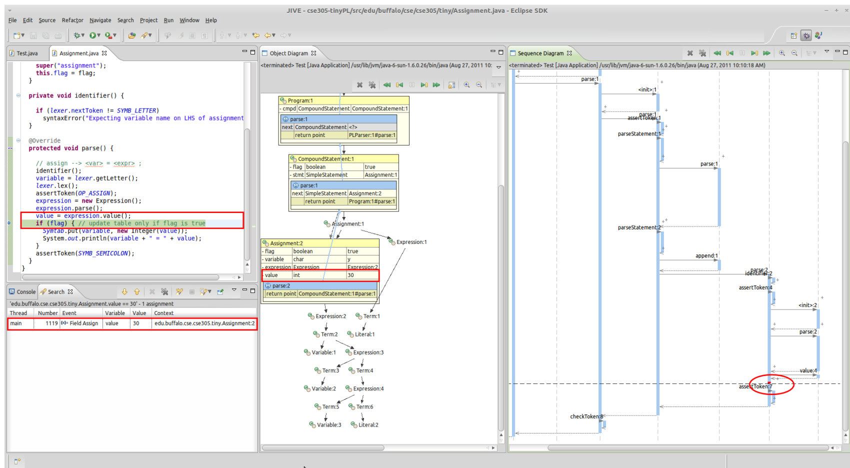


Figure: Query answers are reported on the SD, which provides rich context for interpretation. Diagram regions irrelevant to query answers are collapsed. Hence, queries help focus the SD on regions of interest.

Incorporating the Temporal Query Language

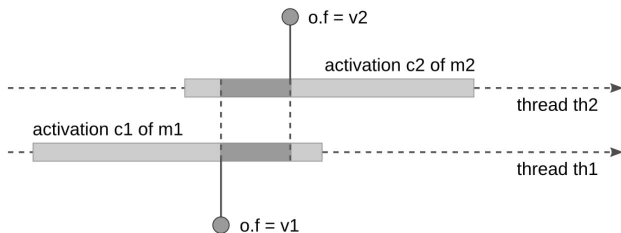
Our temporal query language, PRACTQL, will support diverse applications:

- Query-based debugging.
- Program comprehension.
- Dynamic program visualizations.
- Dynamic analyses.
  - Recursive queries are particularly important, e.g., dynamic slicing.
- Analysis of multiple program runs.

$\Rightarrow$ Syntax is SQL (Datalog notation used for presentation).

Example: Concurrency

Find all pairs of activations of methods m1 and m2 which modify field f of object o while their executions overlap in time.



Concurrent(c1, c2, o, f) :− Activation(c1, th1, _, m1, t2),
EventBind(c1, o, f, v1, t1), Activation(c2, th2, _, m2, t1),
EventBind(c2, o, f, v2, t2), th1 $\neq$ th2

Example: State Analysis

When were the values 10 and 20 found *simultaneously* among the nodes of a binary tree whose root node has $oid = 5$?

```
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)
```

Example: Dynamic Reaching Definitions

Defs(V1,V2,T) records all instants T when variable V1 is assigned a value obtained from the evaluation of an expression involving variable V2. Find all dynamic reaching definitions of 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*.

Next...

1. Temporal Data Model

2. Recursive Query Evaluation

Query Language Overview

PRACTQL extends SQL/TP with recursion.

- SQL/TP is a point-based temporal extension of SQL.
- Compiles non-recursive point-based queries into SQL/92.
  - Points are translated into intervals.
  - Insufficient to handle all queries, e.g., set operations.
  - Some operations must be evaluated over time-compatible queries.
  - I.E., if tuples agree on data components, intervals coincide or are disjoint.
  - Normalization is the operation used to achieve time-compatibility.
  - It changes representation but preserves meaning w.r.t. ATDBs.

Normalization

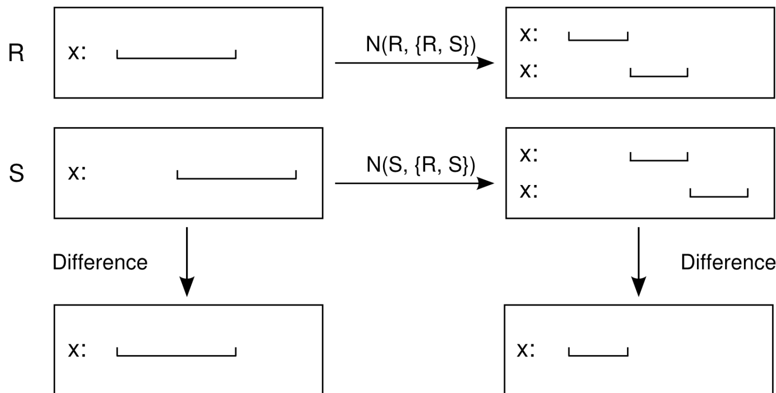


Figure: Only the set difference perfomed on time-compatible relations (right column) preserves semantics with respect to ATDBs.

Recursive Query Evaluation

Evaluation of recursive PRACTQL queries (under set semantics).

1. PRACTQL query compilation.
   - Identical to SQL/TP for non-recursive queries.
   - Omits normalization for recursive queries.
   - Normalization introduces non-linearity and non-stratified negation.
2. Normalizing-Naïve, a modified naïve bottom-up evaluation.
   - Definition of T_{NP}, a normalizing T_P operator.
     - T_{NP} performs unions over time-compatible relations.
   - Modification of the termination condition.
     - Equivalence of instances in consecutive stages tested w.r.t. ATDBs.
     - In practice, we evaluate set differences of time-compatible relations.

```
1  Normalizing−Naive(P)
2   I := Ø;
3   repeat
4     J := I;
5     I := T_NP(I);
6   until || I || = || J ||;
7   return I;
```

Example: Normalizing-Naïve Evaluation

Refs(O, R, T) keeps track of all instants T during which object O references object R. The query below computes a temporal transitive closure of Refs:

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

The PRACTQL compilation produces the query below:

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

Evaluation of $\overline{\mathrm{TTC}}$ for the instance of $\overline{\mathrm{Refs}}$ containing tuple $(1, 1, 1, +\infty)$:

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

$T_{NP} \uparrow 2$ represents the same abstract temporal relation as $T_{NP} \uparrow 1$. Hence, the termination condition is satisfied.

Example: Standard Termination Condition

Evaluation of $\overline{\text{TTC}}$ for the instance of $\overline{\text{Refs}}$ containing tuple $(1, 1, 1, 2^k + 1)$, using the standard termination condition:

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

Equivalence is not detected until stage $2^k + 1$ (blowup!) but for any consecutive stages i and j, $\| T_{NP} \uparrow i \| = \| T_{NP} \uparrow j \|$.

Example: Standard Naïve Bottom-Up Evaluation

Evaluation of $\overline{\text{TTC}}$ for the instance of $\overline{\text{Refs}}$ containing tuple $(1, 1, 1, 2^k + 1)$, using the standard naïve bottom-up evaluation:

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

At every stage $i > 1$, $T_P \uparrow i$ does not preserve set semantics w.r.t. ATDBs, e.g., $(1, 1, 1, 2^k + 1)$ and $(1, 1, 2, 2^k + 1)$ in $T_P \uparrow 2$ represent duplicate information: $\| (1, 1, 1, 2^k + 1) \| \supset \| (1, 1, 2, 2^k + 1) \|$.

Example: SQL/TP Compilation

The SQL/TP compilation produces the query:

$$\overline{\text{TTC}}(x,y,\ell,r) :- \overline{\text{TTC}_1}(x,y,t_\ell,t_r), \text{Imin}(x,y,\ell,r), t_\ell \leq \ell, r \leq t_r$$

$$\overline{\text{TTC}}(x,y,\ell,r) :- \overline{\text{TTC}_2}(x,y,t_\ell,t_r), \text{Imin}(x,y,\ell,r), t_\ell \leq \ell, r \leq t_r$$

$$\overline{\text{TTC}_1}(x,y,\ell,r) :- \overline{\text{Refs}}(x,y,\ell,r)$$

$$\overline{\text{TTC}_2}(x,y,\ell,r) :- \overline{\text{TTC}}(x,z,\ell_1,r_1), \overline{\text{Refs}}(z,y,\ell_2,r), r > \ell_1 + 1, \ell = max(\ell_2, \ell_1 + 1)$$

$$\text{Imin}(x,y,\ell,r) :- \text{EP}(x,y,\ell), \text{EP}(x,y,r), \ell < r, \neg\text{NImin}(x,y,\ell,r)$$

$$\text{NImin}(x,y,\ell,r) :- \text{EP}(x,y,\ell), \text{EP}(x,y,t), \text{EP}(x,y,r), \ell < t, t < r$$

$$\text{EP}(x,y,\ell) :- \overline{\text{TTC}_1}(x,y,\ell,r)$$

$$\text{EP}(x,y,r) :- \overline{\text{TTC}_1}(x,y,\ell,r)$$

$$\text{EP}(x,y,\ell) :- \overline{\text{TTC}_2}(x,y,\ell,r)$$

$$\text{EP}(x,y,r) :- \overline{\text{TTC}_2}(x,y,\ell,r)$$

Normalization makes the $\overline{\text{TTC}}$ rules time-compatible by joining with Imin and projecting the minimal subinterval. The resulting query is non-linear and the negation introduced in the body of Imin is non-stratified.

Soundness, Completeness, and Termination

- Soundness and completeness proof based on:
  - soundness and correctness of both T_P and SQL/TP, and
  - preservation of point-based set semantics at every stage.
- Termination proof based on:
  - termination of the bottom-up evaluation of constraint Datalog.
  - PRACTQL queries are essentially Datalog queries with integer gap-order constraints.

Related Work

Explicitly stored traces.

- Whyline $\Rightarrow$ exposes relevant natural language questions.
- Omniscient $\Rightarrow$ navigates the trace and allows inspection of past states.
- TOD $\Rightarrow$ omniscient debugger with procedural queries.

Implicitly stored traces.

- PTQL $\Rightarrow$ queries are evaluated on-line using instrumented code.
- Coca $\Rightarrow$ Prolog-like queries to define breakpoints and query states.

Other.

- Lencevicius et al $\Rightarrow$ suspend and query the actual object heap.
- bddbddb $\Rightarrow$ Datalog queries for static analysis.

Conclusions and Future Work

- Summary.
  - Temporal data model and query language for debugging.
  - Evaluation of temporal recursive queries.
  - Many practical applications.
  - Principled approach, similar in spirit to bddbddb.
- Future Work.
  - Implementation.
  - Evaluation.
  - Optimizations.
  - Applications.