JIVE: Dynamic Analysis for Java

Overview, Architecture, and Implementation

Demian Lessa

Computer Science and Engineering
State University of New York, Buffalo

Dec. 01, 2010

Outline

Next...

What can you tell about a program?

Questions we often need to answer about programs and their execution:

- How is the system designed?
- How do system components communicate?
- How does control flow during execution?
- How does the state of an object change during execution?
- Did an object ever have a particular state? At what times?
- What caused an object to have a particular state?
- How do threads and objects/methods interact?
- What caused a method to execute?
- Was a particular method ever called? At what times?
- What parameters were passed to a method call?
- What value was returned by a method call?

How do you answer such questions?

Pick the right technique(s) for the job.

- Static analysis looks at the code but does not execute it.
  - all execution paths, undecidability issues, AST/DFG/CFG/..., etc;
  - comprehension– architecture extraction, querying, etc;
  - debugging– static checkers to match source code patterns.

How do you answer such questions?

Pick the right technique(s) for the job.

- Static analysis looks at the code but does not execute it.
  - all execution paths, undecidability issues, AST/DFG/CFG/..., etc;
  - comprehension– architecture extraction, querying, etc;
  - debugging– static checkers to match source code patterns.
- Model checking verifies if a model of the program violates its specs.
  - all execution paths, symbolic execution, state space explosion, etc;
  - debugging– execution trace of a spec violation.

How do you answer such questions?

Pick the right technique(s) for the job.

- Static analysis looks at the code but does not execute it.
  - all execution paths, undecidability issues, AST/DFG/CFG/..., etc;
  - comprehension– architecture extraction, querying, etc;
  - debugging– static checkers to match source code patterns.
- Model checking verifies if a model of the program violates its specs.
  - all execution paths, symbolic execution, state space explosion, etc;
  - debugging– execution trace of a spec violation.
- Dynamic analysis executes the code and looks at execution data.
  - single execution path, probe effect, scalability, execution traces, etc;
  - comprehension– interaction extraction, querying, etc;
  - debugging– reverse execution, querying, etc.

How do you answer such questions?

Pick the right technique(s) for the job.

- Static analysis looks at the code but does not execute it.
  - all execution paths, undecidability issues, AST/DFG/CFG/..., etc;
  - comprehension– architecture extraction, querying, etc;
  - debugging– static checkers to match source code patterns.
- Model checking verifies if a model of the program violates its specs.
  - all execution paths, symbolic execution, state space explosion, etc;
  - debugging– execution trace of a spec violation.
- Dynamic analysis executes the code and looks at execution data.
  - single execution path, probe effect, scalability, execution traces, etc;
  - comprehension– interaction extraction, querying, etc;
  - debugging– reverse execution, querying, etc.
- Hybrid approaches combine aspects of the above techniques.
  - symbolic execution for test generation;
  - static analysis for selective tracing;
  - ...

Where does JIVE stand?

JIVE is a dynamic analysis tool for Java programs featuring:

- Traditional, break-step-inspect interactive debugging.

Where does JIVE stand?

JIVE is a dynamic analysis tool for Java programs featuring:

- Traditional, break-step-inspect interactive debugging.
- Forward and reverse stepping/jumping.
  - Revert to previous states in order to diagnose errors.

Where does JIVE stand?

JIVE is a dynamic analysis tool for Java programs featuring:

- Traditional, break-step-inspect interactive debugging.
- Forward and reverse stepping/jumping.
  - Revert to previous states in order to diagnose errors.
- Dynamic visualizations of state and execution.
  - UML-based object diagrams (ODs) for state snapshots.
  - UML-based sequence diagrams (SDs) for execution.
  - ODs clarify many aspects of OO semantics.
  - SDs clarify concurrent program behavior and object interactions.

Where does JIVE stand?

JIVE is a dynamic analysis tool for Java programs featuring:

- Traditional, break-step-inspect interactive debugging.
- Forward and reverse stepping/jumping.
  - Revert to previous states in order to diagnose errors.
- Dynamic visualizations of state and execution.
  - UML-based object diagrams (ODs) for state snapshots.
  - UML-based sequence diagrams (SDs) for execution.
  - ODs clarify many aspects of OO semantics.
  - SDs clarify concurrent program behavior and object interactions.
- Queries over execution traces.
  - Investigate (temporal) program properties.
  - Debug programs by identifying suspicious conditions.
  - Integrate query answers with dynamic visualizations.

Where does JIVE stand?

JIVE is a dynamic analysis tool for Java programs featuring:

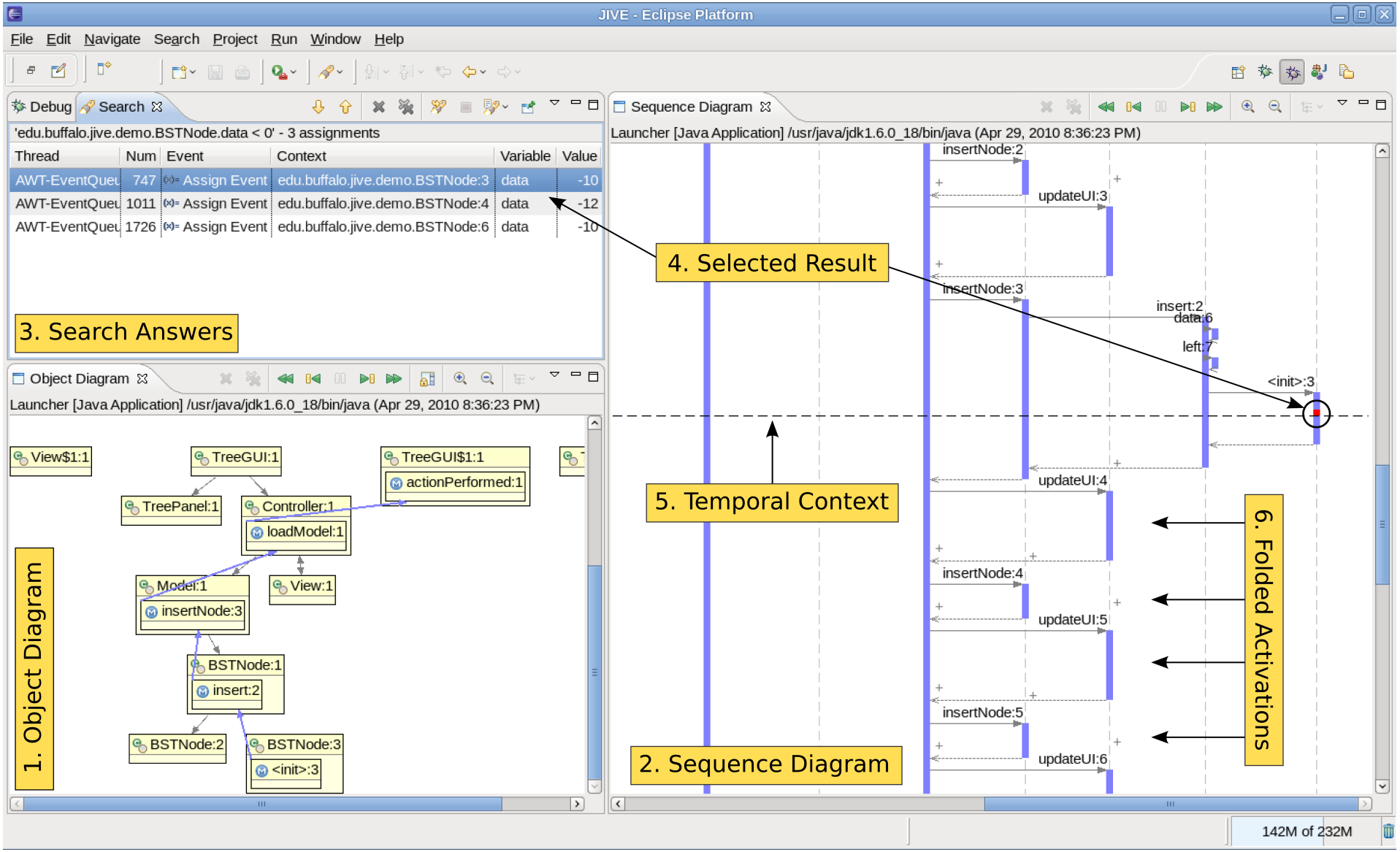
- Traditional, break-step-inspect interactive debugging.
- Forward and reverse stepping/jumping.
  - Revert to previous states in order to diagnose errors.
- Dynamic visualizations of state and execution.
  - UML-based object diagrams (ODs) for state snapshots.
  - UML-based sequence diagrams (SDs) for execution.
  - ODs clarify many aspects of OO semantics.
  - SDs clarify concurrent program behavior and object interactions.
- Queries over execution traces.
  - Investigate (temporal) program properties.
  - Debug programs by identifying suspicious conditions.
  - Integrate query answers with dynamic visualizations.
- Selective trace filtering.
  - Focus on relevant parts of the source.

JIVE in Practice

Things you probably know...

- JIVE is integrated with Eclipse as a collection of plugins.
- Requires programs to execute in debug mode.
- To run JIVE, it must be enabled in your program's debug profile.
- The JIVE perspective provides several views.
  - Contour Model.
  - Object Diagram.
  - Sequence Model.
  - Sequence Diagram.
  - Event Log.
- Requires Java 1.6+ and Eclipse 3.5+; supports *ix, Mac, and Win.
- It has a home: `http://www.cse.buffalo.edu/jive`.
- It is open source: `http://code.google.com/p/jive`.

JIVE's User Interface



Next...

Data Collection

JIVE collects data continually from a running program.

- How do we collect data?
  - Modified JVM? JVM API? Debug API? AOP? Instrumentation?

Data Collection

JIVE collects data continually from a running program.

- How do we collect data?
  - Modified JVM? JVM API? Debug API? AOP? Instrumentation?
- What does the collected data look like?
  - Depends! But expect variable reads/writes, method calls/returns, etc.

Data Collection

JIVE collects data continually from a running program.

- How do we collect data?
  - Modified JVM? JVM API? Debug API? AOP? Instrumentation?
- What does the collected data look like?
  - Depends! But expect variable reads/writes, method calls/returns, etc.
- How do we model data within JIVE (JIVE data model)?
  - Relations? Objects? Graphs? Should these be temporal?
  - Decouple data collection– simply map it to the JIVE data model.

Data Collection

JIVE collects data continually from a running program.

- How do we collect data?
  - Modified JVM? JVM API? Debug API? AOP? Instrumentation?
- What does the collected data look like?
  - Depends! But expect variable reads/writes, method calls/returns, etc.
- How do we model data within JIVE (JIVE data model)?
  - Relations? Objects? Graphs? Should these be temporal?
  - Decouple data collection– simply map it to the JIVE data model.
- How do we store data (JIVE data store) with minimum contention?
  - Memory? Disk? RDBMS? Other DBMS– OODBMS, NoSQL?
  - Decouple data store– use a common abstraction for storage and access.

Data Collection

JIVE collects data continually from a running program.

- How do we collect data?
  - Modified JVM? JVM API? Debug API? AOP? Instrumentation?
- What does the collected data look like?
  - Depends! But expect variable reads/writes, method calls/returns, etc.
- How do we model data within JIVE (JIVE data model)?
  - Relations? Objects? Graphs? Should these be temporal?
  - Decouple data collection– simply map it to the JIVE data model.
- How do we store data (JIVE data store) with minimum contention?
  - Memory? Disk? RDBMS? Other DBMS– OODBMS, NoSQL?
  - Decouple data store– use a common abstraction for storage and access.
- How do we access data cheaply and with minimum contention?
  - Directly? OO APIs (e.g., iterators, visitors)? Declarative query language?

Data Collection

JIVE collects data continually from a running program.

- How do we collect data?
  - Modified JVM? JVM API? Debug API? AOP? Instrumentation?
- What does the collected data look like?
  - Depends! But expect variable reads/writes, method calls/returns, etc.
- How do we model data within JIVE (JIVE data model)?
  - Relations? Objects? Graphs? Should these be temporal?
  - Decouple data collection– simply map it to the JIVE data model.
- How do we store data (JIVE data store) with minimum contention?
  - Memory? Disk? RDBMS? Other DBMS– OODBMS, NoSQL?
  - Decouple data store– use a common abstraction for storage and access.
- How do we access data cheaply and with minimum contention?
  - Directly? OO APIs (e.g., iterators, visitors)? Declarative query language?
- Shouldn't we also collect some static data?

Data Processing

JIVE processes the collected data continually.

- Update the trace model (i.e., raw trace data).

Data Processing

JIVE processes the collected data continually.

- Update the trace model (i.e., raw trace data).
- Update derived models (e.g., object and sequence models).

Data Processing

JIVE processes the collected data continually.

- Update the trace model (i.e., raw trace data).
- Update derived models (e.g., object and sequence models).
- Notify interested parties (typically views) of model updates.

Data Processing

JIVE processes the collected data continually.

- Update the trace model (i.e., raw trace data).
- Update derived models (e.g., object and sequence models).
- Notify interested parties (typically views) of model updates.
- Views respond to model updates by rendering affected diagram parts.

Data Processing

JIVE processes the collected data continually.

- Update the trace model (i.e., raw trace data).
- Update derived models (e.g., object and sequence models).
- Notify interested parties (typically views) of model updates.
- Views respond to model updates by rendering affected diagram parts.
- Ideally, a subsystem should coordinate these tasks. That is,
  - Data arrivals should be isolated from data updates.
  - Data updates should be isolated from view renderings.
  - In general, subsystems should be decoupled from each other.

Visualizations

JIVE renders views continually in response to model updates.

- Views are renderings of their respective models.

Visualizations

JIVE renders views continually in response to model updates.

- Views are renderings of their respective models.
- Some views require simple processing (e.g., Event Log).
  - For each model element, display a log entry line.

Visualizations

JIVE renders views continually in response to model updates.

- Views are renderings of their respective models.
- Some views require simple processing (e.g., Event Log).
  - For each model element, display a log entry line.
- Others require more complex processing (e.g., OD and SD).
  - Select a strategy to traverse some data strucure (i.e., model).
  - Use configurations, interactive state, and traversal context to decide:

Visualizations

JIVE renders views continually in response to model updates.

- Views are renderings of their respective models.
- Some views require simple processing (e.g., Event Log).
  - For each model element, display a log entry line.
- Others require more complex processing (e.g., OD and SD).
  - Select a strategy to traverse some data strucure (i.e., model).
  - Use configurations, interactive state, and traversal context to decide:
    - what to render at each step;
    - whether to traverse substructures;
    - whether to continue the traversal after each step completes.

Visualizations

JIVE renders views continually in response to model updates.

- Views are renderings of their respective models.
- Some views require simple processing (e.g., Event Log).
  - For each model element, display a log entry line.
- Others require more complex processing (e.g., OD and SD).
  - Select a strategy to traverse some data strucure (i.e., model).
  - Use configurations, interactive state, and traversal context to decide:
    - what to render at each step;
    - whether to traverse substructures;
    - whether to continue the traversal after each step completes.
- Views should be rendered independently and concurrently.

Omniscient Debugging

JIVE supports omniscient debugging.

- It knows about all program states.

Omniscient Debugging

JIVE supports omniscient debugging.

- It knows about all program states.
- It supports interactive navigation to arbitrary points in execution.

Omniscient Debugging

JIVE supports omniscient debugging.

- It knows about all program states.
- It supports interactive navigation to arbitrary points in execution.
- This requires support for the notion of temporal context (TC).

Omniscient Debugging

JIVE supports omniscient debugging.

- It knows about all program states.
- It supports interactive navigation to arbitrary points in execution.
- This requires support for the notion of temporal context (TC).
- Normally, TC is in sync with the program counter (PC).

Omniscient Debugging

JIVE supports omniscient debugging.

- It knows about all program states.
- It supports interactive navigation to arbitrary points in execution.
- This requires support for the notion of temporal context (TC).
- Normally, TC is in sync with the program counter (PC).
- Temporal navigation breaks this sync and initiates replay mode.

Omniscient Debugging

JIVE supports omniscient debugging.

- It knows about all program states.
- It supports interactive navigation to arbitrary points in execution.
- This requires support for the notion of temporal context (TC).
- Normally, TC is in sync with the program counter (PC).
- Temporal navigation breaks this sync and initiates replay mode.
- Replay mode continues until TC and PC are in sync again.

Omniscient Debugging

JIVE supports omniscient debugging.

- It knows about all program states.
- It supports interactive navigation to arbitrary points in execution.
- This requires support for the notion of temporal context (TC).
- Normally, TC is in sync with the program counter (PC).
- Temporal navigation breaks this sync and initiates replay mode.
- Replay mode continues until TC and PC are in sync again.
- Notably, views are rendered to reflect TC not PC.

Omniscient Debugging

JIVE supports omniscient debugging.

- It knows about all program states.
- It supports interactive navigation to arbitrary points in execution.
- This requires support for the notion of temporal context (TC).
- Normally, TC is in sync with the program counter (PC).
- Temporal navigation breaks this sync and initiates replay mode.
- Replay mode continues until TC and PC are in sync again.
- Notably, views are rendered to reflect TC not PC.
- Hence, views are naturally synchronized during replay mode.

Queries

JIVE supports template-based searches and declarative SPJ queries.

- JIVE maintains data against which queries are formulated.

Queries

JIVE supports template-based searches and declarative SPJ queries.

- JIVE maintains data against which queries are formulated.
- Template-based searches are predefined, form-based queries.
  - Select a template, provide parameters, and execute.

Queries

JIVE supports template-based searches and declarative SPJ queries.

- JIVE maintains data against which queries are formulated.
- Template-based searches are predefined, form-based queries.
  - Select a template, provide parameters, and execute.
- SPJ queries are still at a proof-of-concept stage.
  - Write a declarative query using a SQL-like language and execute.

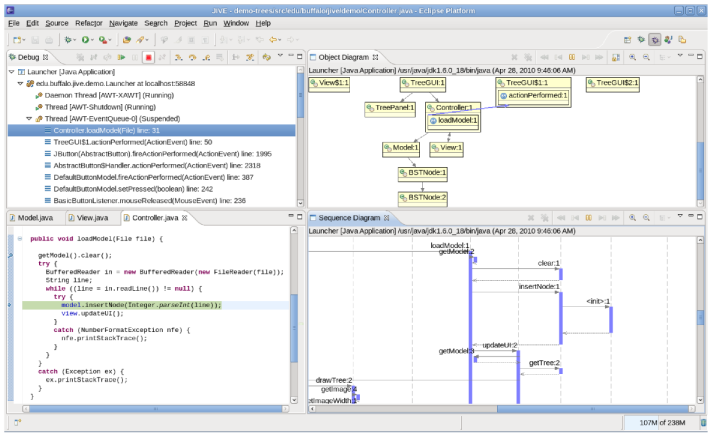
Queries

JIVE supports template-based searches and declarative SPJ queries.

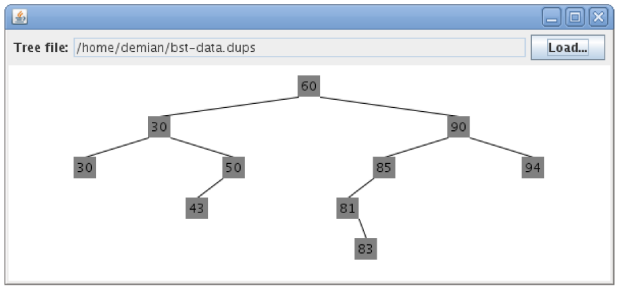
- JIVE maintains data against which queries are formulated.
- Template-based searches are predefined, form-based queries.
  - Select a template, provide parameters, and execute.
- SPJ queries are still at a proof-of-concept stage.
  - Write a declarative query using a SQL-like language and execute.
- Query answer reporting is uniform.
  - Eclipse's seach results window provides tabular and hierarchical views.
  - The SD highlights query answers and focuses on their activations.
  - Focusing means maximally hiding all unrelated parts of the SD.
  - Double-clicking query answers navigates to the corresponding TC.

JIVE's MVC Architecture (Simplified)

JIVE UI



Debuggee UI



debugger requests

debugger responses

debuggee inputs

debuggee UI updates

JIVE Data Model

- Trace Model
- Object Model
- Sequence Model

model writes

model reads

JIVE Controller

- Event Handler
- Model Manager
- UI Engine

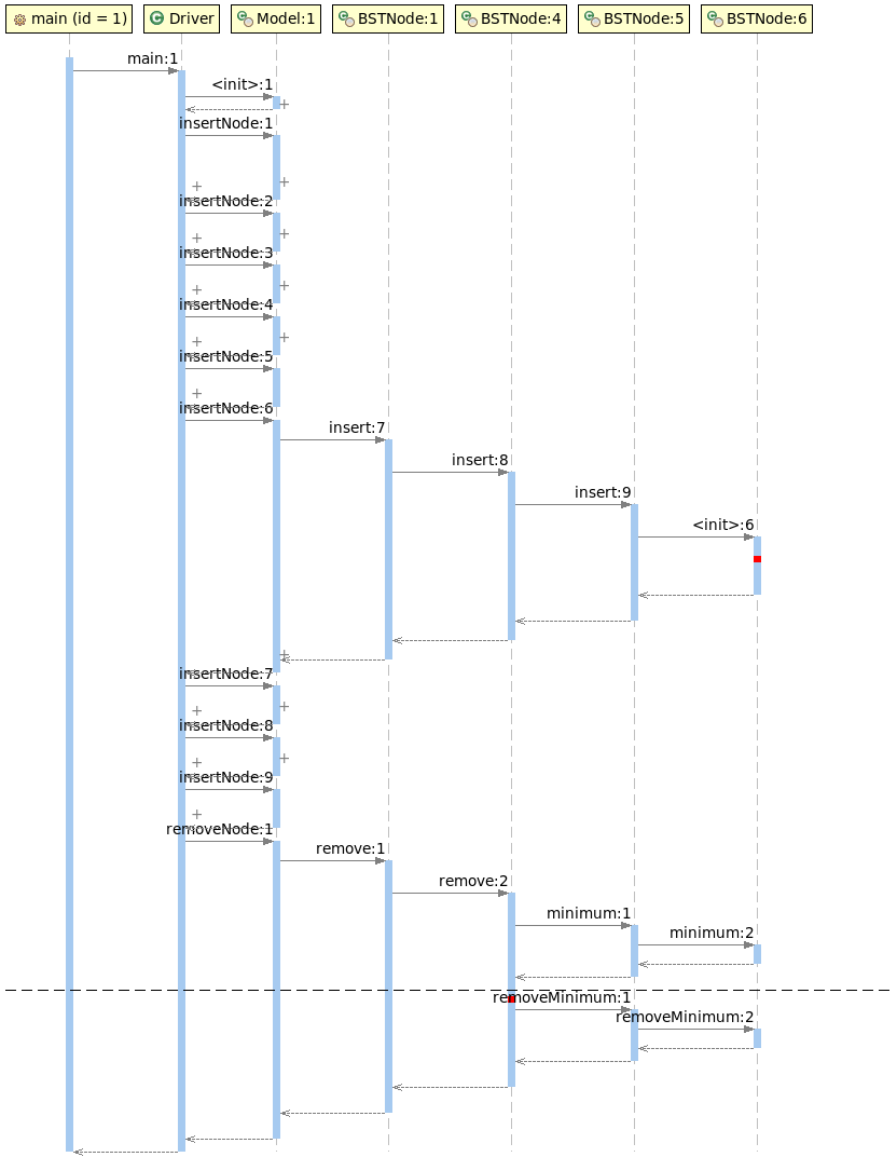
JDI event notifications

JDI event requests

JPDA Debugger

- Front-End
- Back-End
- Debuggee JVM

JIVE Focused Search Results



Next...

Project Structure

JIVE is implemented as a collection of Eclipse plugins.

- `edu.buffalo.cse.jive.feature` (meta)
  - plugin definitions, dependencies, requirements, etc;
- `edu.buffalo.cse.jive.launching` (hooks)
  - replaces debug launchers; provides an extended debugger;
- `edu.buffalo.cse.jive.core` (debugger)
  - extended debugger for JDI event handling; model updates;
- `edu.bsu.cs.jive` (types)
  - JIVE data model; JIVE data store; utilities;
- `edu.buffalo.cse.jive.core.adapter` (expose)
  - exposes Eclipse's Java Debug Target (JDT) for extension;
- `edu.buffalo.cse.jive.ui` (applications)
  - views; searches, queries, and query answers;

edu.buffalo.cse.jive.core

- Creates JIVE data models and registers them as JIVE event listeners.

edu.buffalo.cse.jive.core

- Creates JIVE data models and registers them as JIVE event listeners.
- Requests JDI events such as:
  - thread start/end, class prepare, method enter/exit, field change, step.

edu.buffalo.cse.jive.core

- Creates JIVE data models and registers them as JIVE event listeners.
- Requests JDI events such as:
  - thread start/end, class prepare, method enter/exit, field change, step.
- Receives and handles JDI event notifications:
  - updates internal data structures;
  - creates JIVE events;
  - notifies observers (i.e., models) on a separate thread;
  - models update themselves based on JIVE events.

edu.buffalo.cse.jive.core

- Creates JIVE data models and registers them as JIVE event listeners.
- Requests JDI events such as:
  - thread start/end, class prepare, method enter/exit, field change, step.
- Receives and handles JDI event notifications:
  - updates internal data structures;
  - creates JIVE events;
  - notifies observers (i.e., models) on a separate thread;
  - models update themselves based on JIVE events.
- Most debug tasks performed by Eclipse's debugger.

edu.buffalo.cse.jive.core

- Creates JIVE data models and registers them as JIVE event listeners.
- Requests JDI events such as:
  - thread start/end, class prepare, method enter/exit, field change, step.
- Receives and handles JDI event notifications:
  - updates internal data structures;
  - creates JIVE events;
  - notifies observers (i.e., models) on a separate thread;
  - models update themselves based on JIVE events.
- Most debug tasks performed by Eclipse's debugger.
- Heavily based on the observer pattern.

edu.buffalo.cse.jive.core

- Creates JIVE data models and registers them as JIVE event listeners.
- Requests JDI events such as:
  - thread start/end, class prepare, method enter/exit, field change, step.
- Receives and handles JDI event notifications:
  - updates internal data structures;
  - creates JIVE events;
  - notifies observers (i.e., models) on a separate thread;
  - models update themselves based on JIVE events.
- Most debug tasks performed by Eclipse's debugger.
- Heavily based on the observer pattern.
- Statistics: 26 files, 5KLoC.

edu.bsu.cs.jive

- Defines JIVE events, data models and their elements.

edu.bsu.cs.jive

- Defines JIVE events, data models and their elements.
- Implements data models using in-memory storage.
  - Encapsulates most concrete implementations using creational patterns.
  - Typically, a factory provides methods for creating model elements.
  - Many model elements accept some form of visitor object.

edu.bsu.cs.jive

- Defines JIVE events, data models and their elements.
- Implements data models using in-memory storage.
  - Encapsulates most concrete implementations using creational patterns.
  - Typically, a factory provides methods for creating model elements.
  - Many model elements accept some form of visitor object.
- Object Model (a.k.a. Contour Model) is the most complex.
  - Contours are transactional.
  - That is, every model change is encapsulated in a transaction.
  - Transactions can be committed/rolled back– the heart of replay mode.
  - Uses the state pattern in its implementation.

edu.bsu.cs.jive

- Defines JIVE events, data models and their elements.
- Implements data models using in-memory storage.
  - Encapsulates most concrete implementations using creational patterns.
  - Typically, a factory provides methods for creating model elements.
  - Many model elements accept some form of visitor object.
- Object Model (a.k.a. Contour Model) is the most complex.
  - Contours are transactional.
  - That is, every model change is encapsulated in a transaction.
  - Transactions can be committed/rolled back– the heart of replay mode.
  - Uses the state pattern in its implementation.
- Extensive use of the observer pattern.

edu.bsu.cs.jive

- Defines JIVE events, data models and their elements.
- Implements data models using in-memory storage.
  - Encapsulates most concrete implementations using creational patterns.
  - Typically, a factory provides methods for creating model elements.
  - Many model elements accept some form of visitor object.
- Object Model (a.k.a. Contour Model) is the most complex.
  - Contours are transactional.
  - That is, every model change is encapsulated in a transaction.
  - Transactions can be committed/rolled back– the heart of replay mode.
  - Uses the state pattern in its implementation.
- Extensive use of the observer pattern.
- Statistics (totals): 115 files, 15KLoC.

edu.bsu.cs.jive

- Defines JIVE events, data models and their elements.
- Implements data models using in-memory storage.
  - Encapsulates most concrete implementations using creational patterns.
  - Typically, a factory provides methods for creating model elements.
  - Many model elements accept some form of visitor object.
- Object Model (a.k.a. Contour Model) is the most complex.
  - Contours are transactional.
  - That is, every model change is encapsulated in a transaction.
  - Transactions can be committed/rolled back– the heart of replay mode.
  - Uses the state pattern in its implementation.
- Extensive use of the observer pattern.
- Statistics (totals): 115 files, 15KLoC.
- Statistics (contour): 42 files, 11KLoC.

edu.buffalo.cse.jive.ui

- Uses Eclipse's GEF framework and SWT.
  - Retrieve model elements to display their contents.
  - Compose UI elements using edit parts and figures.
  - You define actions for events (e.g., clicking on a menu item).

edu.buffalo.cse.jive.ui

- Uses Eclipse's GEF framework and SWT.
  - Retrieve model elements to display their contents.
  - Compose UI elements using edit parts and figures.
  - You define actions for events (e.g., clicking on a menu item).
- Template-based searches.
  - Pre-configured forms define required and/or optional fields.
  - Requested search is validated.
  - Model is searched– each search requires a visitor implementation.

edu.buffalo.cse.jive.ui

- Uses Eclipse's GEF framework and SWT.
  - Retrieve model elements to display their contents.
  - Compose UI elements using edit parts and figures.
  - You define actions for events (e.g., clicking on a menu item).
- Template-based searches.
  - Pre-configured forms define required and/or optional fields.
  - Requested search is validated.
  - Model is searched– each search requires a visitor implementation.
- SPJ queries.
  - Textual input with syntax validation.
  - A rudimentary query engine is in place.
  - It implements true joins, although in somewhat naïve fashion.

edu.buffalo.cse.jive.ui

- Uses Eclipse's GEF framework and SWT.
  - Retrieve model elements to display their contents.
  - Compose UI elements using edit parts and figures.
  - You define actions for events (e.g., clicking on a menu item).
- Template-based searches.
  - Pre-configured forms define required and/or optional fields.
  - Requested search is validated.
  - Model is searched– each search requires a visitor implementation.
- SPJ queries.
  - Textual input with syntax validation.
  - A rudimentary query engine is in place.
  - It implements true joins, although in somewhat naïve fashion.
- Extensive use of the observer pattern.

edu.buffalo.cse.jive.ui

- Uses Eclipse's GEF framework and SWT.
  - Retrieve model elements to display their contents.
  - Compose UI elements using edit parts and figures.
  - You define actions for events (e.g., clicking on a menu item).
- Template-based searches.
  - Pre-configured forms define required and/or optional fields.
  - Requested search is validated.
  - Model is searched– each search requires a visitor implementation.
- SPJ queries.
  - Textual input with syntax validation.
  - A rudimentary query engine is in place.
  - It implements true joins, although in somewhat naïve fashion.
- Extensive use of the observer pattern.
- Statistics: 195 files, 35KLoC.

Next...

Conclusion

- Summary.
- Future of JIVE.
- Are you interested?