Static Analysis of Embedded DSL-s

Aivar Annamaa
University of Tartu

aivar.annamaa@gmail.com

Joint work with Andrey Breslav, Varmo Vene, Jevgeni Kabanov

October 3rd, 2010
Rakari Theory Days

Problem

- DSL-s are often embedded as **string literals** in a GPL
  - SQL, RegEx, HTML
- It's flexible:
  - host-language compiler doesn't care
- It's **error prone**:
  - host-language compiler doesn't care
  - mistakes (may) pop up at **runtime**
  - **conditional concatenation** makes things hard

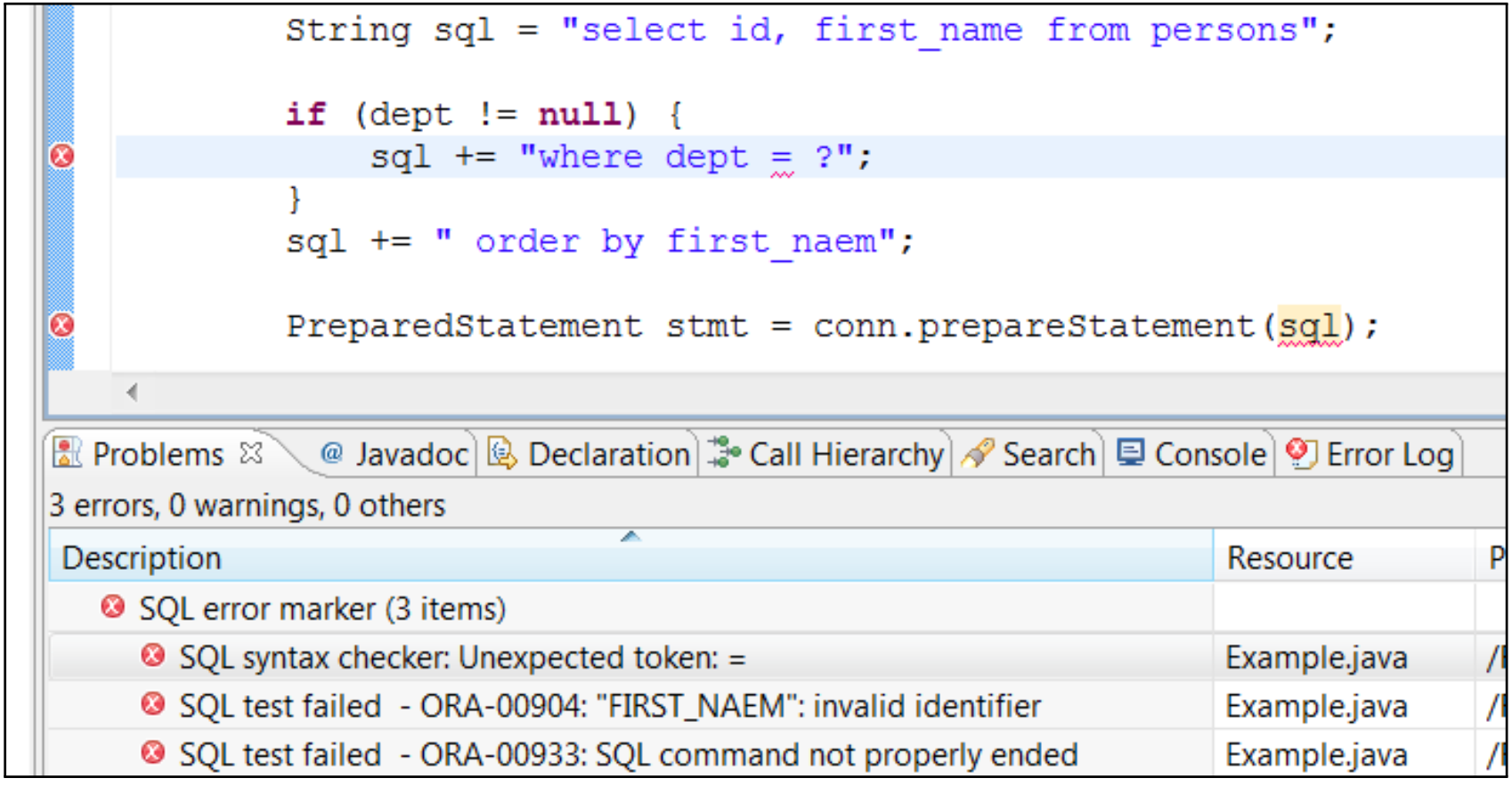
Example: SQL in Java

```
String sql = "select id, first_name from persons";

if (dept != null) {
    sql += "where dept = ?";
}
sql += " order by first_naem";

// runtime error ??
PreparedStatement stmt = conn.prepareStatement(sql);
```

A Solution: tool support



Outline of the talk

- Static analyzer for SQL embedded in Java
  - Properties & main phases
  - Collecting SQL strings as abstract strings
  - Testing concrete strings against actual database
  - Parsing abstract strings

(Desired) Properties of the analysis

- **Sound**: no errors found $\Rightarrow$ no errors at runtime
- **Efficient**: fast enough for on-line usage (while typing)
- **Precise**: no false alarms for common idioms of SQL construction
- Helpful in complex cases (loops, complex branching etc.)

Steps of analysis

1. Locate "**hotspots**" ie. arguments for "special" methods, eg. `prepareStatement(`**`sql`**`)`
2. Find **abstract value** of each hotspot
3. Verify each abstract value:
   - **test** concrete values against real DB
   - **parse** abstract values directly
4. Report any errors found

Abstract strings

Hotspot string expression is represented as **abstract string** – **regular set** of all possible values

```
AbsStr ::= Constant String
         | Sequence [AbsStr]
         | Choice [AbsStr]
         | Repetition AbsStr
```

Example hotspot and abstract string

```
String sql = "select id, name from persons";
if (dept != null) {
    sql += "where dept = ?";
}
sql += " order by name";

PreparedStatement stmt = conn.prepareStatement(sql);
```

```
Sequence [
    Constant "select id, name from persons",
    Choice [Constant "where dept = ?", Constant ""],
    Constant " order by name"
]
```

String collector. Intraprocedural analysis

Finds abstract value of hotspot expression

```
eval (StringLiteral s)  = Constant s
eval (Concat exp1 exp2) = Sequence [eval exp1, eval exp2]
eval (Name var)         =
     Choice [ eval(e) | e <- precedingAssignments(var)]
```

- Assignment in if-statement results in a `Choice`
- Concatenation in loops is evaluated in 2 steps:
  1. construct *extended* abstract string (context-free)
  2. widen to regular abstract string (may lose precision)

String collector. Parameters

If part of hotspot expression comes from method parameter:

1. locate all possible **callsites** of current method
2. evaluate respective **argument expression**
3. return **Choice** of resulting abstract strings

Depth of recursion, when evaluating arguments, is limited

String collector. Method calls

If hotspot expression includes a method call:

1. locate all possible **implementations**
2. evaluate **return expression(s)**
3. return **Choice** from different implementations

Testing concrete strings

- If abstract string for a hotspot is finite (no Repetition), then all possible **concrete strings** are generated
- In case of **Repetition**, repetition is performed once and twice
- Concrete strings are **tested** on actual schema using actual DB engine
- If testing (`prepareStatement`) fails, then error message from DB is forwarded to the user

Syntax analysis

- Why testing isn't enough?
  - Not sound for infinite abstract strings
  - Syntax analysis enables better positioning of error markers
  - Syntax analysis enables *content assist*
- What we want to know?
  - Does given abstract string $\subset$ SQL ?
- How to solve this?
  - Abstract lexing + abstract parsing

(Abstract) lexing

Preparatory work to simplify parsing

- Input: (*abstract*) character-string
- Output: (*abstract*) token-string (or failure)

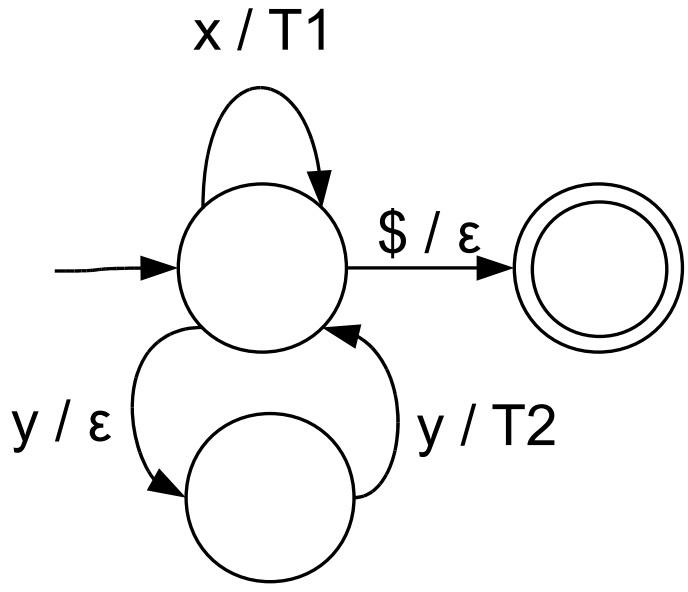
Abstract char-string → abstract token-string

```
Sequence [
  'f','r','o','m',' ','p','e','r','s','o','n','s',
  Choice [
    Sequence ['w','h','e','r','e',' ','d','e','p','t',$],
    $
]]
```

```
Sequence [
  Token FROM,
  Choice [
    Token (ID "persons"),
    Sequence [
      Token (ID "personswhere"),
      Token (ID "dept"),
  EOF
]]]
```

Lexing with Finite State Transducers

Extension to FA - transitions can generate some output



Using FST-s in abstract setting (1)

- Translate abstract char-string to abstract token-string
- Involves 3 automata: $M_{out} = P(M_{fst}, M_{in})$
  - M_{fst}: language specific lexer (eg. for SQL)
- Algorithm P "merges" M_{in} and M_{fst}
  - simulate M_{fst} on all paths through M_{in}
  - collect pairs of "matching" transitions
  - finish when fixpoint is reached
  - construct M_{out} from collected transition-pairs

Using FST-s in abstract setting (2)

Does it work?

- Fixpoint is found because there is finite number of transition-pairs
- Theory: Finite state transduction of a regular language is regular

Pragmatics

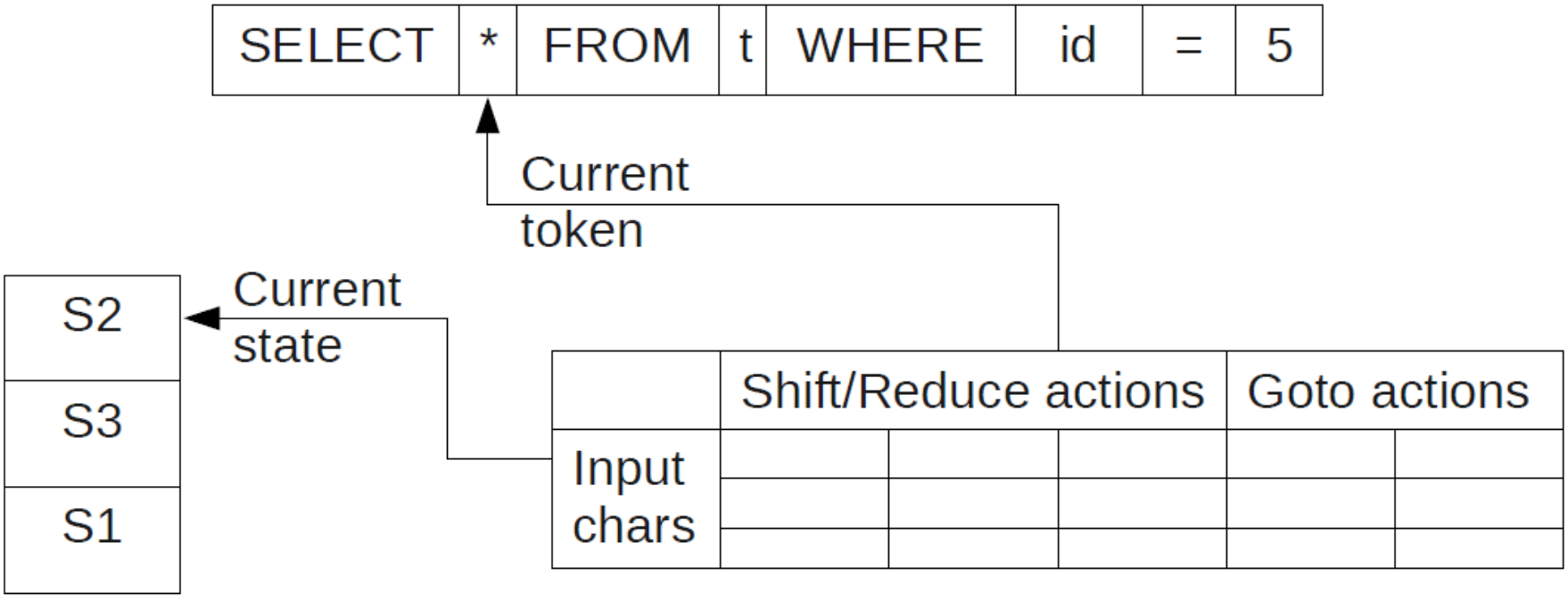
- M_{fst} for SQL is generated with standard tools (JFlex)

Abstract parsing

Does given abstract token-string represent a sublanguage of SQL?

- Inclusion of regular language in CF-language is undecidable in general
- Decidable for input without repetion
- Otherwise need to abstract (and lose precision)

Abstract LR parsing



- Use normal LR tables (generated by Bison)
- Intuition: when input string branches, then fork the parser
- Actually: explore all paths and record all different combinations of stack/input transitions until fixpoint

Abstract LR parsing with infinite input

If input includes repetition then fixpoint may not be found

- Solution: only remember top n items of the parsing stack
- Loss in precision:
  - eg. if $n = 3$, then expression `((((x))))` not recognized anymore

Conclusion

Ingredients:

- Interprocedural constant-propagation analysis
- Exhaustive testing of concrete strings
- Abstract lexing/parsing
- Cache for partial results

Results:

- Eclipse JDT plugin, implemented in Java
- Can analyse Compiere ERP system (300K LOC) in 3 min
  - 1343 hotspots
  - **12 bugs found** (10 syntax errors, 2 name misspellings)
  - **7 false alarms** (require path-sensitivity)
  - 129 unsupported hotspots (non-local mutable state, complex loops)
  - reanalyzing single file: ca. 0.5 sec