First Step toward an Abstract Debugger
Serializing Everything in Yoyak

Jaeho Shin

2010-08-13
ROPAS Show&Tell
Contents

1. Why
2. How
3. Discussion
Why
Plan

1. Serialize things on filesystem
2. Make engine work with them
3. Make browser show them
4. Add interactive features
5. Do some Abstract Debugging!
Why Serialize?

To reveal everything inside the analyzer transparently to humans as well as machines.
Why Serialize?

To create a working ground for building independent tools that

- use computed abstraction for solving various problems
- visualize details of the on-going analysis
- control the behavior of the analyzer

Interactive or Automatic Refinement

Abstract State Browser

Error Checkers
Property Verifiers
Yoyak’s Working Ground

Source Code

Frontend Tools

Yoyak Directory

Abstract Interpretation Engine

Abstract State Browser

Error Checkers
Property Verifiers

Interactive or Automatic Refinement

Interactive or Automatic Refinement

...
Why on Filesystem?

Both Machine and Human Friendly

• Machines have rich sets of methods for handling files and directories
• Humans feel relatively comfortable navigating through directories and viewing files
How
Yoyak Directory

- A root directory that contains everything
  - Graphs
  - States
  - Table
  - Dump
  - Work list
  - ...

\[
\text{Summary} = \text{Graph} \times \text{Table} \times \text{Dump}
\]

\[
\text{Worklist} = \varphi(\text{Block}_P) \times \varphi(\text{Block}_P)
\]
JSON

- “Javascript Object Notation”
- Simple format for representing Semi-structured data
- Wide-spread
- Compact, compared to XML

"a "JSON" string"

1234  5.67e+89
true  null
false

```
{  
  "1": "one",  
  "two": 2,  
  "pi": 3.141592, 
  "first 8 primes":  
    [2, 3, 5, 7,  
     11, 13, 17, 19] 
}
```

```
[  
  1,  
  2.3,  
  "four"  
]
```

```
"a \"JSON\" string\n"
```

```
1234  5.67e+89
true  null
false
```
Serializing Graphs

- `/files/FILE/functions/FUNCTION/` (directory)
  - for each Control Flow Graph of `FUNCTION` in `FILE`
  - `bBLOCK` (JSON Object file)
    for details of a Basic Block, `BLOCK`
  - `edges` (JSON Object file)
    for all Flows-to and Returns-after-call Edges

\[
\text{Graph} = \text{Edges}
\]

\[
\begin{align*}
\text{CFG} & = \text{Flows} \times \text{Resumes} \\
\text{Flows} & = \text{Edges} \\
\text{Resumes} & = \text{Edges} \\
\text{Edges} & = \wp(\text{Block} \times \text{Block})
\end{align*}
\]
Serializing Blocks

- List of Commands (or Statements)
- Source code origin information

Excerpt of cjson.yoyak/files/cjson/test.c/functions/create_objects/b82:

```json
{
  "commands": [
    "alloc(.t.361[4,1])",
    "set(.t.361[0],73)",
    "set(.t.361[1],68)",
    "set(.t.361[2],115)",
    "set(.t.361[3],0)"
  ],
  "source": {
    "file": "cjson/test.c",
    "lines": [97, 97],
    "columns": [33, 33]
  }
}
```

```
b ∈ Block
b ::= CALL(e,e,e) | ENTRY_p | EXIT_p | c^* 
c ∈ Cmd
c ::= SET(e,e) | ESCAPE(e) | ASSUME(r)
   | ALLOC(e,s)_t | FREE(e)
e ∈ Expr
r ∈ Rel
s ∈ Shape
```
Serializing Edges

Excerpt of cjson.yoyak/files/cjson/test.c/functions/create_objects/edges:

```json
{
  "flows": {
    "b128": ["bEXIT"], "b127": ["b128"], "b126": ["b127"],
    "b125": ["b126"], "b124": ["b7"], "b123": ["b5"], "b122": ["b123"],
    ...
    "b9": ["b10"], "b7": ["b125"], "b5": ["b96", "b124"], "b4": ["b91", "b92"],
    "b3": ["b53"], "b1": ["b48", "b52"], "b0": ["b43", "b44"], "bENTRY": ["b9"]
  },
  "returns": {
    "b127": ["b128"], "b125": ["b126"], "b122": ["b123"], "b121": ["b122"],
    ...
    "b7": ["b125"], "b3": ["b53"]
  }
}
```

Edges $= \Phi(Block \times Block)$
Serializing States

- **/states/STATE** (JSON file)

- for an Abstract State identified with **STATE**

- **STATE** is the SHA1 sum of its contents

- (Content format varies depending on how abstract domain is represented)

Excerpt of cjson.yoyak/states/c7/bf/e6ebbeb9d2cc74882d725ccc79b8a69cb193:

```json
{
    "Var(child.224)": "Top",
    ...
    "Var(value.223)": "\tInterval = [0,+oo)\tFun = \tAddr = \tAblk = (l.1099, Off[0,+oo], Size[247,247], Stride[1,1], NullPosIBot) (l.1502, Off[0,+oo], Size[399,399], Stride[1,1], NullPosIBot) (l.710, Off[0,+oo], Size[153,153], Stride[1,1], NullPosIBot) (l.793, Off[0,+oo], Size[79,79], Stride[1,1], NullPosIBot) (l.848, Off[0,+oo], Size[51,51], Stride[1,1], NullPosIBot)\tSblk = ",
    ...
    "Var(.t.300)": "\tInterval = IBot\tFun = \tAddr = , Null\tAblk = (l.1518, Off[0,0], Size[20,20], Stride[1,1], NullPos[19,19])\tSblk = ",
    ...
    "Dyn(1.1518)": "\tInterval = [0,109]\tFun = \tAddr = \tAblk = \tSblk = ",
    ...
}
```

\[ Memory = \hat{Map} \times \hat{Alloc} \]
\[ Map = \hat{Addr}^{\text{fin}} \rightarrow \hat{Value} \]
Serializing State Table

- /files/FILE/functions/FUNCTION/bBLOCK.{inputs,outputs} (JSON Object files)
  - Input and output Abstract States for BLOCK
  - We keep track of trace token of each Abstract State

Excerpt of cjson.yoyak/files/cjson/test.c/functions/create_objects/b82.outputs:

```json
{
  "create_objects":
  "c7bfe6ebebb9d2cc74882d725ccc79b8a69cb193"
}
```

\[ Table \quad = \quad Block \xrightarrow{\text{fin}} Memory \]
Serializing Work List

- /engine/work/WORK (JSON Array file)
  - Represents “Work” to do for the work-list algorithm
  - Keeps track of which Block needs more computation
  - Work for a Block with different Trace tokens possible

```json
[
  "cjson/test.c>create_objects>b82",
  "create_objects"
]
```
Discussion
Discussion

- Namespace
- Persistency
- Size Matters
- Efficiency vs. Scalability
Namespace

Now we can easily identify things exactly with paths:

- `/files/cjson/cJSON.c/functions/pars_string.76/b23`
- `/files/cjson/test.c/functions/create_objects/b82.outputs`
- `/states/c7/bf/e6ebebb9d2cc74882d725ccc79b8a69cb193`
- `/engine/work/123`
- `...`
Persistency

With persistent storage of everything,

- **Pausing** and **continuing** a computation are possible with a much simpler implementation
- **How to start** another fixed-point computation **with augmented information** becomes apparent
## Size Matters

<table>
<thead>
<tr>
<th>Airac's Peak Memory Usage (x86_64)</th>
<th>Airac’s XML dump</th>
<th>Yoyak Directory</th>
<th>Gzip’ed Airac’s XML dump</th>
<th>Tar.gz’ed Yoyak Directory</th>
</tr>
</thead>
<tbody>
<tr>
<td>cJSON</td>
<td>~50,000 KB</td>
<td>1 File 26,812 KB</td>
<td>2,992 Files 18,948 KB</td>
<td>1 File 430 KB</td>
</tr>
<tr>
<td>3 Files</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4,072 LOC</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>459 SLOC</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sample2</td>
<td>~2,500 KB</td>
<td>1 File 11 KB</td>
<td>81 Files 220 KB</td>
<td>1 File 1 KB</td>
</tr>
<tr>
<td>2 Files</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>32 LOC</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>31 SLOC</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>uav-loop.c</td>
<td>~2,500 KB</td>
<td>1 File 12 KB</td>
<td>77 Files 220 KB</td>
<td>1 File 1 KB</td>
</tr>
<tr>
<td>1 File</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>25 LOC</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>23 SLOC</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Efficiency vs. Scalability

By keeping everything on the filesystem,

- All parts of computation must go through file operations, lots of file operations will make it incredibly inefficient!

But, remember it is always about trade-off:

- We can exceed physical, and even virtual memory limits!
- Distributing computation to multiple nodes can be simple as sharing the Yoyak Directory over network filesystems
Any Questions or Comments?
Thank you!
References


• JSON: Javascript Object Notation. [http://json.org](http://json.org)