Garbage Collection for Monitoring Parametric Properties

**Dongyun Jin**, Patrick Meredith, Dennis Griffith, Grigore Rosu

University of Illinois at Urbana-Champaign
Outline

- Motivation
- Property Static Analysis
- Double Lazy Garbage Collection
- Evaluation
- Conclusion
Monitoring

- Scalable technique for ensuring software reliability
- Observe run of a program

- Execute & Observe
- Event 1
- Event 2
- Event 3
- Event 4
- Analyze

- Analyze execution trace against desired properties
- React/Report using handlers (if needed)
Applications of Monitoring

- Development
  - Debugging
  - Testing
- Deployment
  - Security
  - Reliability
  - Runtime Verification
Parametric Properties

- Properties referring to object instances
- The following property describes a bad behavior between Each Collection $c$ and Iterator $i$:

```
create($c, i$) \rightarrow update($c$) \rightarrow next($i$) \rightarrow update($c$)
```

- Generalize typestates
  - Typestates are parametric properties with one parameter
Parametric Property Monitoring Systems

- **Tracematches** [de Moor et al. 05]
  - Regular Patterns
- **PQL** [Lam et al. 05]
  - Context-Free Patterns
- **PTQL** [Aiken et al. 05]
  - SQL-like language
- **Eagle then RuleR** [Barringer et al. 04]
  - Rule-Based Rewrite Properties (e.g. Context-Free Patterns, ..)
- **MOP** [Rosu et al. 03]
  - Formalism Independent Approach
- many others
Parametric Monitoring in MOP

- Keep one monitor for each parameter instance
  - A parameter instance binds parameters to objects
  - E.g., \((c \mapsto c_2, i \mapsto i_3)\)

- Each monitor knows nothing of parameters; operates exclusively on only one trace slice
Parametric Monitoring in MOP

- Keep one monitor for each parameter instance
  - A parameter instance binds parameters to objects
  - E.g., \((c \leftrightarrow c_2, i \leftrightarrow i_3)\)
- Each monitor knows nothing of parameters; operates exclusively on only one trace slice

Trace: \(create(c_1, i_1)\) \(create(c_1, i_2)\) \(next(i_1)\) \(update(c_1)\) \(next(i_2)\)
Parametric Monitoring in MOP

- Keep one monitor for each parameter instance
  - A parameter instance binds parameters to objects
  - E.g., \((c \leadsto c_2, \ i \leadsto i_3)\)
- Each monitor knows nothing of parameters; operates exclusively on only one trace slice

<table>
<thead>
<tr>
<th>Trace:</th>
<th>create((c_1, i_1))</th>
<th>create((c_1, i_2))</th>
<th>next((i_1))</th>
<th>update((c_1))</th>
<th>next((i_2))</th>
</tr>
</thead>
<tbody>
<tr>
<td>((c_1, i_1)) Trace Slice:</td>
<td>create</td>
<td>next</td>
<td>update</td>
<td></td>
<td></td>
</tr>
<tr>
<td>((c_1, i_2)) Trace Slice:</td>
<td>create</td>
<td>update</td>
<td>next</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Parametric Monitoring in MOP

- Keep one monitor for each parameter instance
  - A parameter instance binds parameters to objects
  - E.g., \((c \leftrightarrow c_2, i \leftrightarrow i_3)\)
- Each monitor knows nothing of parameters; operates exclusively on only one trace slice

<table>
<thead>
<tr>
<th>Trace:</th>
<th>create((c_1, i_1)) create((c_1, i_2)) next((i_1)) update((c_1)) next((i_2))</th>
</tr>
</thead>
<tbody>
<tr>
<td>((c_1, i_1)) Trace Slice:</td>
<td>create next update</td>
</tr>
<tr>
<td>((c_1, i_2)) Trace Slice:</td>
<td>create update next</td>
</tr>
</tbody>
</table>
Parametric Monitoring in MOP

- Keep one monitor for each parameter instance
  - A parameter instance binds parameters to objects
  - E.g., \((c \leftrightarrow c_2, i \leftrightarrow i_3)\)
- Each monitor knows nothing of parameters; operates exclusively on only one trace slice

<table>
<thead>
<tr>
<th>Trace:</th>
<th>create((c_1, i_1))</th>
<th>create((c_1, i_2))</th>
<th>next((i_1))</th>
<th>update((c_1))</th>
<th>next((i_2))</th>
</tr>
</thead>
<tbody>
<tr>
<td>((c_1, i_1)) Trace Slice:</td>
<td>create</td>
<td>next</td>
<td>update</td>
<td></td>
<td></td>
</tr>
<tr>
<td>((c_1, i_2)) Trace Slice:</td>
<td>create</td>
<td>update</td>
<td>next</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Challenge: Large number of monitors
Previously...

- Benchmark based on DaCapo 9.12 [Blackburn et al. 06]
  - On average for 65 cases (13 examples × 5 properties)
- Tracematches [de Moor et al. 05]
  - One of the most memory efficient monitoring systems
  - 142% Runtime overhead
  - 12% Memory overhead
- MOP [Rosu et al. 03]
  - 33% Runtime overhead
  - 140% Memory overhead

1: Except 6(of 65) crashes
Worst Cases – Runtime Overhead

From DaCapo 9.12 and DaCapo 2006-10 MR2

- bloat, HasNext: 448% (Tracematches), 2119% (MOP)
- bloat, UnsafeIterator: 569% (Tracematches), 19194% (MOP)
- avrora, UnsafeIterator: 311% (Tracematches), 637% (MOP)
- bloat, UnsafeMapIterator: 1203% (MOP)
- pmd, UnsafeMapIterator: OOM
- bloat, UnsafeSyncColl: 746% (MOP), ~4M Monitors
- bloat, UnsafeSyncMap: 716% (MOP), 1942% (MOP)
Worst Cases – Memory Overhead

From DaCapo 9.12 and DaCapo 2006-10 MR2

- **bloat, HasNext**: Tracematches 57%, MOP 294%, 1059%
- **bloat, UnsafeIterator**: Tracematches 5%, MOP 2896%, OOM
- **avrora, UnsafeIterator**: Tracematches 5%, MOP 2799%
- **bloat, UnsafeMapIterator**: Tracematches 39%, MOP 2896%, 1082%, OOM
- **pmd, UnsafeMapIterator**: Tracematches 41%, MOP 2510%, 1030%, OOM
- **bloat, UnsafeSyncColl**: Tracematches 39%, MOP 2510%
- **bloat, UnsafeSyncMap**: Tracematches 41%, MOP 1030%
Goal: Reduce the Runtime/Memory Overhead

- Static
  - Program Analysis
    [Bodden et al. 09, 10], [Dwyer et al. 10], ...
  - Property Analysis

- Dynamic

- By using information from property analysis, garbage collect monitors when they become useless during monitoring
Outline

- Motivation
- Property Static Analysis
- Double Lazy Garbage Collection
- Evaluation
- Conclusion
Property Static Analysis

- Property analysis tells us when monitors are unnecessary
- Our analysis is formalism independent; we only show it for FSM monitors (see paper for CFG monitors)

![Diagram showing the sequence of events: update(c), next(i), update(c), next(i), create(c, i), update(c), next(i).]

<table>
<thead>
<tr>
<th>The Last Event</th>
<th>Parameter instances that need to be alive</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Property Static Analysis

- Property analysis tells us when monitors are unnecessary.
- Our analysis is formalism independent; we only show it for FSM monitors (see paper for CFG monitors).

### The Last Event

<table>
<thead>
<tr>
<th>The Last Event</th>
<th>Parameter instances that need to be alive</th>
</tr>
</thead>
<tbody>
<tr>
<td>create</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Property Static Analysis

- Property analysis tells us when monitors are unnecessary
- Our analysis is formalism independent; we only show it for FSM monitors (see paper for CFG monitors)

<table>
<thead>
<tr>
<th>The Last Event</th>
<th>Parameter instances that need to be alive</th>
</tr>
</thead>
<tbody>
<tr>
<td>create</td>
<td></td>
</tr>
</tbody>
</table>

![Diagram](image-url)
Property Static Analysis

- Property analysis tells us when monitors are unnecessary
- Our analysis is formalism independent; we only show it for FSM monitors (see paper for CFG monitors)

<table>
<thead>
<tr>
<th>The Last Event</th>
<th>Parameter instances that need to be alive</th>
</tr>
</thead>
<tbody>
<tr>
<td>create</td>
<td></td>
</tr>
<tr>
<td>update(c)</td>
<td></td>
</tr>
<tr>
<td>next(i)</td>
<td></td>
</tr>
<tr>
<td>update(c)</td>
<td></td>
</tr>
<tr>
<td>next(i)</td>
<td></td>
</tr>
</tbody>
</table>

Needed Events
Property Static Analysis

- Property analysis tells us when monitors are unnecessary.
- Our analysis is formalism independent; we only show it for FSM monitors (see paper for CFG monitors).

```
create(c, i)

update(c)  next(i)  update(c)  next(i)

The Last Event  Parameter instances that need to be alive
create  \{c, i\}
```
Property Static Analysis

- Property analysis tells us when monitors are unnecessary
- Our analysis is formalism independent; we only show it for FSM monitors (see paper for CFG monitors)

<table>
<thead>
<tr>
<th>The Last Event</th>
<th>Parameter instances that need to be alive</th>
</tr>
</thead>
<tbody>
<tr>
<td>create</td>
<td>{c, i}</td>
</tr>
</tbody>
</table>

If any of these parameters are garbage collected, this monitor cannot trigger; can be collected
Property Static Analysis

- Property analysis tells us when monitors are unnecessary
- Our analysis is formalism independent; we only show it for FSM monitors (see paper for CFG monitors)

<table>
<thead>
<tr>
<th>The Last Event</th>
<th>Parameter instances that need to be alive</th>
</tr>
</thead>
<tbody>
<tr>
<td>create</td>
<td>{c, i}</td>
</tr>
<tr>
<td>update</td>
<td></td>
</tr>
</tbody>
</table>
Property Static Analysis

- Property analysis tells us when monitors are unnecessary
- Our analysis is formalism independent; we only show it for FSM monitors (see paper for CFG monitors)

<table>
<thead>
<tr>
<th>The Last Event</th>
<th>Parameter instances that need to be alive</th>
</tr>
</thead>
<tbody>
<tr>
<td>create</td>
<td>{c, i}</td>
</tr>
<tr>
<td>update</td>
<td></td>
</tr>
</tbody>
</table>
Property Static Analysis

- Property analysis tells us when monitors are unnecessary.
- Our analysis is formalism independent; we only show it for FSM monitors (see paper for CFG monitors).

The Last Event | Parameter instances that need to be alive
--- | ---
create | \( \{c, i\} \)
update | 

Needed Events
Property Static Analysis

- Property analysis tells us when monitors are unnecessary.
- Our analysis is formalism independent; we only show it for FSM monitors (see paper for CFG monitors).

<table>
<thead>
<tr>
<th>The Last Event</th>
<th>Parameter instances that need to be alive</th>
</tr>
</thead>
<tbody>
<tr>
<td>create</td>
<td>{c, i}</td>
</tr>
<tr>
<td>update</td>
<td>{i}</td>
</tr>
</tbody>
</table>
Property Static Analysis

- Property analysis tells us when monitors are unnecessary.
- Our analysis is formalism independent; we only show it for FSM monitors (see paper for CFG monitors).

### Needed Events

- **create** \(\{c, i\}\)
- **update** \(\{i\}\)

### The Last Event

<table>
<thead>
<tr>
<th>The Last Event</th>
<th>Parameter instances that need to be alive</th>
</tr>
</thead>
<tbody>
<tr>
<td>create</td>
<td>({c, i})</td>
</tr>
<tr>
<td>update</td>
<td>({i})</td>
</tr>
</tbody>
</table>
Property Static Analysis

- Property analysis tells us when monitors are unnecessary
- Our analysis is formalism independent; we only show it for FSM monitors (see paper for CFG monitors)

Needed Events

<table>
<thead>
<tr>
<th>The Last Event</th>
<th>Parameter instances that need to be alive</th>
</tr>
</thead>
<tbody>
<tr>
<td>create</td>
<td>{c, i}</td>
</tr>
<tr>
<td>update</td>
<td>{i}</td>
</tr>
<tr>
<td></td>
<td>{c, i}</td>
</tr>
</tbody>
</table>

Needed Events
Property Static Analysis

- Property analysis tells us when monitors are unnecessary.
- Our analysis is formalism independent; we only show it for FSM monitors (see paper for CFG monitors).

<table>
<thead>
<tr>
<th>The Last Event</th>
<th>Parameter instances that need to be alive</th>
</tr>
</thead>
<tbody>
<tr>
<td>create</td>
<td>{c, i}</td>
</tr>
<tr>
<td>update</td>
<td>{i}</td>
</tr>
<tr>
<td></td>
<td>{c, i}</td>
</tr>
<tr>
<td>next</td>
<td>{c, i}</td>
</tr>
</tbody>
</table>
Property Static Analysis

- Property analysis tells us when monitors are unnecessary
- Our analysis is formalism independent; we only show it for FSM monitors (see paper for CFG monitors)

<table>
<thead>
<tr>
<th>The Last Event</th>
<th>Parameter instances that need to be alive</th>
</tr>
</thead>
<tbody>
<tr>
<td>create</td>
<td>( {c, i} )</td>
</tr>
<tr>
<td>update</td>
<td>( {i} ) ( {c, i} )</td>
</tr>
<tr>
<td>next</td>
<td>( {c, i} )</td>
</tr>
</tbody>
</table>

If any of these are dead, then the current monitor can be collected.
Outline

- Motivation
- Property Static Analysis
- **Double Lazy Garbage Collection**
- Evaluation
- Conclusion
Double Lazy Garbage Collection

- **Indexing Trees** map parameter instances to monitors

- **Lazy** propagation of info about dead parameter instances
  - Eager propagation yields high runtime overhead (like TM)
  - **Idea:** propagate when monitors are updated for other reasons

- **Lazy** removal of useless monitors
  - Removal is expensive (monitor can belong to many trees)
  - **Idea:** just mark unnecessary monitors, then clean up later

(technical; see paper for details and proof of correctness)
Outline

- Motivation
- Property Static Analysis
- Double Lazy Garbage Collection
- **Evaluation**
- Conclusion
Evaluation – Overall

- On average, based on the DaCapo 9.12 Benchmark
  - 13 Programs and 5 Properties

<table>
<thead>
<tr>
<th></th>
<th>Tracematches</th>
<th>MOP</th>
<th>RV (MOP + Garbage Collection)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Runtime Overhead</td>
<td>142%&lt;sup&gt;1&lt;/sup&gt;</td>
<td>33%</td>
<td>15%</td>
</tr>
<tr>
<td>Memory Overhead</td>
<td>12%&lt;sup&gt;1&lt;/sup&gt;</td>
<td>140%</td>
<td>39%</td>
</tr>
</tbody>
</table>

<sup>1</sup>: Except 6(of 65) crashes
Evaluation – Worst Cases Runtime Overhead

From DaCapo 9.12 and DaCapo 2006-10 MR2

- **bloat, HasNext**
  - Tracematches: 116%
  - MOP: 448%
  - RV: 2119%

- **bloat, UnsafeIterator**
  - Tracematches: 251%
  - MOP: 569%
  - RV: 19194%

- **avrora, UnsafeIterator**
  - Tracematches: 118%
  - MOP: 311%
  - RV: 637%

- **bloat, UnsafeMapIterator**
  - Tracematches: 178%
  - MOP: 1203%
  - RV: OOM

- **pmd, UnsafeMapIterator**
  - Tracematches: 188%
  - MOP: 571%
  - RV: OOM

- **bloat, UnsafeSyncColl**
  - Tracematches: 212%
  - MOP: 746%
  - RV: 1359%

- **bloat, UnsafeSyncMap**
  - Tracematches: 130%
  - MOP: 716%
  - RV: 1942%

Approximate MOP: ~ 4M Monitors
Evaluation – Worst Cases Memory Overhead

From DaCapo 9.12 and DaCapo 2006-10 MR2

<table>
<thead>
<tr>
<th>Scenario</th>
<th>Tracematches</th>
<th>MOP</th>
<th>RV</th>
</tr>
</thead>
<tbody>
<tr>
<td>bloat, HasNext</td>
<td>1059%</td>
<td>1059%</td>
<td>1059%</td>
</tr>
<tr>
<td>bloat, UnsafeIterator</td>
<td>294%</td>
<td>169%</td>
<td>1512%</td>
</tr>
<tr>
<td>avrora, UnsafeIterator</td>
<td>236%</td>
<td>236%</td>
<td>2799%</td>
</tr>
<tr>
<td>bloat, UnsafeMapIterator</td>
<td>1045%</td>
<td>1045%</td>
<td>1045%</td>
</tr>
<tr>
<td>pmd, UnsafeMapIterator</td>
<td>1082%</td>
<td>1082%</td>
<td>1082%</td>
</tr>
<tr>
<td>bloat, UnsafeSyncColl</td>
<td>2510%</td>
<td>2510%</td>
<td>2510%</td>
</tr>
<tr>
<td>bloat, UnsafeSyncMap</td>
<td>1030%</td>
<td>1030%</td>
<td>1030%</td>
</tr>
</tbody>
</table>

OOM: Out of Memory
Conclusions

- Garbage collection for parametric monitoring
  - Formalism-independent
  - Directed by property static analysis
- Double Lazy Garbage Collection
  - Lazy propagation of info
  - Lazy removal of monitors
- Evaluation
  - Lowest runtime overhead parametric monitoring system
  - Reasonable memory overhead