Verifying Concurrency Runtimes using Graph Transformation Systems

Claudio Corrodi\textsuperscript{1}, Chris Poskitt\textsuperscript{2}, Alexander Heußner\textsuperscript{3}

\textsuperscript{1}Software Composition Group, University of Bern, Switzerland
\textsuperscript{2}Singapore University of Technology and Design, Singapore
\textsuperscript{3}Software Technologies Research Group, University of Bamberg, Germany
Chair of Software Engineering

Software Technologies Research Group

[Images of three individuals]
Concurrency Made Easy

O-O Concurrency models

Verification

Testing

Robotics
Concurrency Made Easy

- O-O Concurrency models
- Verification
- Testing
- Robotics
SCOOP

Simple Concurrent Object-Oriented Programming

Goal: Raise concurrency abstractions from error-prone (lock based) models to O-O programming
SCOOP
SCOOP

eat (left, right: separate FORK)
  do
    left.pick_up
    right.pick_up
    print ("I am eating!")
    left.put_down
    right.put_down
  end

Separate block: No intervening calls between “pick_up” and “put_down”
Execution Models

“Request Queues”
Separate block guarantees?
Performant?
Execution Models

“Request Queues”
Separate block guarantees?
Performant? ✗

✔

P2, P3
FORK
P1 P2, P3
Execution Models

“Request Queues”
Separate block guarantees? ✔
Performant? ✗

“Queues of Queues”
Separate block guarantees? ✔
Performant? ✗
Execution Models

“Request Queues”
Separate block guarantees? ✔
Performant? ✗

“Queues of Queues”
Separate block guarantees? ✔
Performant? ✔
Execution Models

“Distributed SCOOP”

Extension of “Queues of Queues” model
Correctness

No race conditions?

Absence of deadlocks?
Correctness

No race conditions?

Is this still a solution?

```plaintext
eat (left, right: separate FORK)
  do
    print ("I am eating!")
  end
```

Absence of deadlocks?
Our work

Can we model and simulate—modularly—competing semantics for a language like SCOOP, and analyse them for semantic discrepancies?
Approach

• Model runtimes as graph transformation systems

• Modular / parameterisable semantics

• Analyse parameterised GTS against representative programs in GROOVE
Graph Transformation Systems

Configuration / state graph

Item value = 0 → next → Item value = 1 → next → Item value = 2 → next → Item value = 3
Graph Transformation Systems

Configuration / state graph

Transformation rule
Graph Transformation Systems

Configuration / state graph

Transformation rule
Graph Transformation Systems

Configuration / state graph

Transformation rule
Graph Transformation Systems

State-space exploration

Nondeterministic application of any matching rule

(labeled transition system)
SCOOP GTS

Static part: control flow graphs

ParameterMapping
  index = 1
  name = "left"

ParameterMapping
  index = 2
  name = "right"

InitialState
  class = "PHILOSOPHER"
  procedure = "eat"

ParameterExpression
  name = "left"

ActionCommand
  procedure = "use"

ControlState
  to_state
  target

ActionCommand
  procedure = "use"
  to_state
  target

FinalState
  to_state
SCOOP GTS

Dynamic part: Handlers and memory state

Static part: control flow graphs
Detecting errors

**Error**
message = "Mutual exclusion error. Both philosophers have entered the eat method."
<table>
<thead>
<tr>
<th>Graph</th>
<th>Runtime</th>
<th>Configurations</th>
<th>Time [s]</th>
</tr>
</thead>
<tbody>
<tr>
<td>DP 2 eager</td>
<td>QoQ</td>
<td>5,863</td>
<td>25.5</td>
</tr>
<tr>
<td></td>
<td>RQ</td>
<td>4,219</td>
<td>18.2</td>
</tr>
<tr>
<td></td>
<td>DSCOOP</td>
<td>13,046</td>
<td>52.9</td>
</tr>
<tr>
<td>DP 2 lazy</td>
<td>QoQ</td>
<td>9,609</td>
<td>40.8</td>
</tr>
<tr>
<td></td>
<td>RQ</td>
<td>5,679</td>
<td>23.5</td>
</tr>
<tr>
<td></td>
<td>DSCOOP</td>
<td>18,874</td>
<td>73.0</td>
</tr>
<tr>
<td>DP 3 eager</td>
<td>QoQ</td>
<td>227,797</td>
<td>1,480.6</td>
</tr>
<tr>
<td></td>
<td>RQ</td>
<td>99,198</td>
<td>436.3</td>
</tr>
<tr>
<td></td>
<td>DSCOOP</td>
<td>523,513</td>
<td>2,726.0</td>
</tr>
<tr>
<td>DP 3 lazy</td>
<td>QoQ</td>
<td>444,689</td>
<td>2,424.9</td>
</tr>
<tr>
<td></td>
<td>RQ</td>
<td>170,249</td>
<td>1,090.1</td>
</tr>
<tr>
<td></td>
<td>DSCOOP</td>
<td>1,288,663</td>
<td>5,999.5</td>
</tr>
<tr>
<td>PC 20</td>
<td>QoQ</td>
<td>50,286</td>
<td>575.0</td>
</tr>
<tr>
<td></td>
<td>RQ</td>
<td>12,890</td>
<td>141.6</td>
</tr>
<tr>
<td></td>
<td>DSCOOP</td>
<td>90,434</td>
<td>997.7</td>
</tr>
</tbody>
</table>
Towards Practical Graph-Based Verification for an Object-Oriented Concurrency Model

Alexander Heußner
University of Bamberg, Germany

Christopher M. Poskitt
Benjamin Morandi
Department of Computer Science
ETH Zürich, Switzerland

Claudio Corrodi

To harness the power of multi-core and distributed platforms, and to make the development of concurrent software more accessible to software engineers, different object-oriented concurrency models such as SCOOP have been proposed. Despite the practical importance of analysing SCOOP programs, there are currently no general verification approaches that operate directly on program code without additional annotations. One reason for this is the multitude of partially conflicting semantic formalisations for SCOOP (either in theory or by-implementation). Here, we propose a simple graph transformation system (GTS) based run-time semantics for SCOOP that grasps the most common features of all known semantics of the language. This run-time model is implemented in the state-of-the-art GTS tool GROOVE, which allows us to simulate, analyse, and verify a subset of SCOOP.
Publications

A Graph-Based Semantics Workbench for Concurrent Asynchronous Programs

Claudio Corrodi\textsuperscript{1,2}, Alexander Heu\ss{}ner\textsuperscript{3}, and Christopher M. Poskitt\textsuperscript{1,4}

\textsuperscript{1} Department of Computer Science, ETH Zürich, Switzerland
\textsuperscript{2} Software Composition Group, University of Bern, Switzerland
\textsuperscript{3} Software Technologies Research Group, University of Bamberg, Germany
\textsuperscript{4} Singapore University of Technology and Design, Singapore

Abstract. A number of novel programming languages and libraries have been proposed that offer simpler-to-use models of concurrency than threads. It is challenging, however, to devise execution models that successfully realise their abstractions without forfeiting performance or introducing unintended behaviours. This is exemplified by Scoop—a concurrent object-oriented message-passing language—which has seen multiple semantics proposed and implemented over its evolution. We propose...
A Semantics Comparison Workbench for Concurrent, Asynchronous, Distributed Programs

Claudio Corrodi\textsuperscript{1}, Alexander Heu\ss{}ner\textsuperscript{2}, and Christopher M. Poskitt\textsuperscript{3}

\textsuperscript{1}Software Composition Group, University of Bern, Switzerland
\textsuperscript{2}Software Technologies Research Group, University of Bamberg, Germany
\textsuperscript{3}Singapore University of Technology and Design, Singapore

Abstract. A number of high-level languages and libraries have been proposed that offer novel and simple to use abstractions for concurrent, asynchronous, and distributed programming. The execution models that realise them, however, often change over time—whether to improve performance, or to extend them to new language features—potentially affecting behavioural and safety properties of existing programs. This is exemplified by SCOOP, a message-passing approach to concurrent object-oriented programming that has
Acknowledgments

• Many slides are adapted from related similar presentations given by Chris Poskitt and Alexander Heußner

• Dining philosophers image taken from Wikipedia

• D-SCOOP figure taken from “An Interference-Free Programming Model for Network Objects” (Schill, Poskitt, Meyer; 2016)