A Graph-Based Semantics Workbench for Concurrent Asynchronous Programs

Alexander Heußner

Otto-Friedrich-Universität Bamberg

Chris Poskitt

ETH Zürich

Dagstuhl, November 2015

Analysis

(Verification) of **Evolving** Graph Structures

CP

- assertional reasoning for attributed GTS (see yesterday's talk)
- verification of (concurrent) object-oriented programs
- ...
- leveraging contracts in software correctness techniques

AH

- verification of dynamic message passing systems (graph grammars, partial order structures, treewidth et al., wqos, abstractions)
- verification of asynchronous concurrent systems
- ...
- reasoning for policies with resources via GTS

Today's topic:

Formalisations and analysis of different state-of-the-art concurrency abstractions for concurrent asynchronous (object-oriented) programs.

☺ We are talking about "real" source code here! ☺

Some Initial Problems here:

- models must be *highly* dynamic
  (e.g. dynamic generation of threads, channels, queues, stacks, wait/dependency relationships)
- expressiveness needed beyond "classical" automaton/Petri net models (e.g. complicated inter process and memory relations)
- semi-formal semantics / semantics "by implementation"
- different competing (and contradictory) semantics
- changes of semantic meta-model are common (and frequent)

Thus:

Not **one** semantic meta model but different competing and possibly contradictory models that also are rivaled by the

mental concurrency model

of the programmer.

Proposed Solution

➪ modular/parameterisable semantics (“semantic plug ins”)

➪ based on graph transformation systems

➪ formalise dynamic runtime semantics

- make scheduler explicit
- make queueing model explicit
- ...

➪ assume static semantics (typing, generics,. . .) already done

Prototypical implementation

- for SCOOP: an object-oriented message-passing language
- prototype based on a GTS model in the GROOVE tool
- plugin for official Eiffel Studio IDE (verification version)
  - get “flattened” source code
  - feedback errors to code display
- also standalone tool working on SCOOP graphs
- parameterisation by SCOOP’s two most recent (competing/contradictory) execution models

(Joint work with Claudio Corrodi.)

Let's take a closer look. . .

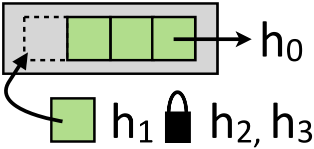
- example piece of SCOOP(-ish) code

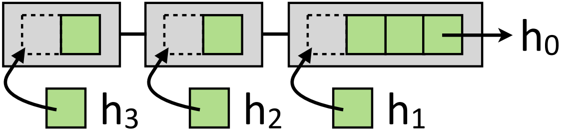
```
separate x,y
do
  x.set_colour(Green)
  y.set_colour(Green)
end
```

```
separate x,y
do
  x.set_colour(Indigo)
  a_colour = x.get_colour
  y.set_colour(a_colour)
end
```

- *separate objects* are associated with threads of execution that have exclusive responsibility for executing methods on them
- *separate block guarantees:* calls are queued as requests in program order; and no intervening requests are queued
- consider two different queueing semantics. . .

(blackboard demo)





Achieved so far…

➪ formalised the two execution models

☺ straightforward parameterisation by GTS rules and programs

➪ included "mental models" of engineers behind compilers and existing formal models in interviews

☺ diagrammatic representation and easy simulation

➪ simple analysis/verification tasks (simulation, deadlock detection,…) help to highlight discrepancies between models

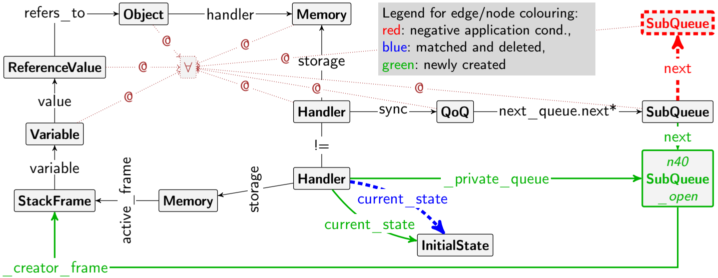
☺ "play" with different semantic meta-models

☺ highlighted a real inconsistency between the queuing semantics

☹ running time for large programs using the generic verification algorithms

A glimpse at the model

- example GTS rule modelling entering a separate block (private queues semantics):



➪ we use control programs to make the model's scheduler explicit (open to parameterisation) and to control atomicity

```
initialize_model;
while (progress & no_error) {
  for each handler p:
    alap handler_local_execution_step(p)+;
  try synchronisation_step;
}
recipe handler_local_execution_step (p){
  try separate_object_creation(p)+;
  else try assignment_to_variable(p)+;
  else try ... ;
  try clean_up_model+;
}
recipe synchronisation_step(){
  reserve_handlers | dequeue_task | ...;
}
...
// ---------- plug in ---------------------
recipe separate_object_creation(p){
  ...
}
...
```

Let's talk about Verification...

(ongoing work)

- generic abstractions for SCOOP graphs
- "well-structuredness" properties of SCOOP graphs
- relation to existing models, submodels, decidable subclasses, ...
- on-the-fly M2M to counting abstractions (Petri nets) etc.
- general concept of "semantics parameterised verfication"

Mid-term/Long-term goals:

- semantic workbench with series of tools usable for the software-engineer (who is writing concurrent software and/or writing compilers/libraries for concurrency abstractions)
- clearer connection to existing approaches (e.g. $\mathbb{K}$ etc.)
- structural comparison of concurrency abstractions from a graph perspective
- properly formalise and algorithmically attack "semantics parameterised analysis"