KOOL: An Application of Rewriting Logic to Language Prototyping and Analysis

Mark Hills and Grigore Roșu
{mhills, grosu}@cs.uiuc.edu

Department of Computer Science
University of Illinois at Urbana-Champaign

26 June 2007

Outline

The KOOL Language

KOOL is

- *object-oriented*: classes, methods, dynamic dispatch, exceptions; all values objects
- *dynamic*: dynamically typed, adding extensions for modifying code at runtime
- *concurrent*: multiple threads of execution, shared memory, locks acquired on objects
- *extensible*, with various features "plugged in": synchronized methods, semaphores, reflective capabilities

Design Motivations for KOOL

- Experiment with OO language features
- Experiment with *optional* and *pluggable* type systems
- Investigate interaction of language features with verification and analysis
- Create a language suitable for languages courses, without some “confusing” features from other languages

A Sample KOOL Program: Classes and Methods

```
class Factorial is
  method Fact(n) is
    if n = 0 then
      return 1;
    else
      return n * self.Fact(n-1);
    fi
  end
end

console << (new Factorial).Fact(200)
```

A Sample KOOL Program: Inheritance

```
class Point is
  var x,y;
  method Point(inx, iny) is
    x <- inx; y <- iny;
  end
  method toString is
    return ("x = " + x.toString() + " and y = " + y.toString());
  end
end

class ColorPoint extends Point is
  var c;
  method ColorPoint(inx, iny, inc) is
    super(inx,iny); c <- inc;
  end
  method toString is
    return (super.toString() + " and c = " + c.toString());
  end
end
```

Rewriting Logic Semantics of Programming Languages

- Rewriting logic is an extension of equational logic with support for concurrency
- Language semantics provides formal definitions of language features
- Rewriting logic semantics: formal language definitions using rewriting logic
- Definitions are executable with rewriting logic engines, like Maude

The Rewriting Logic Semantics Project

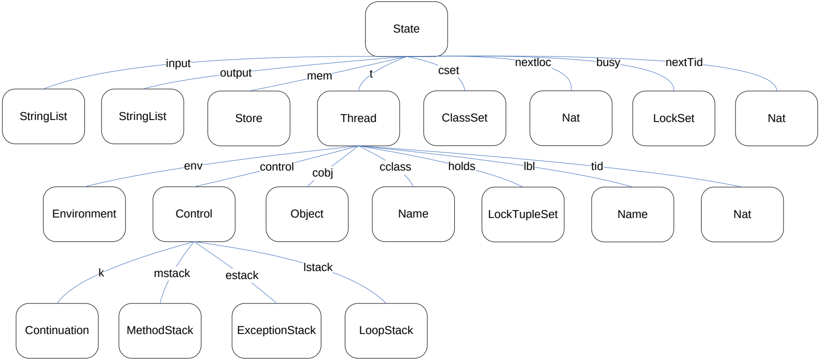
- KOOL is part of ongoing work on rewriting logic semantics
- Other work includes many languages and supporting tools, researchers at multiple universities
- Java, Beta, Scheme, Prolog, Haskell, PLAN, BC, CCS, MSR, ABEL, SILF, FUN, π-calculus, variants of λ-calculus, others

KOOL Program Representation

- States in KOOL represented as multisets of state components
- Multisets formed by putting components next to one another

```
op _ _ :  KState KState -> KState [assoc comm id:  empty]
```

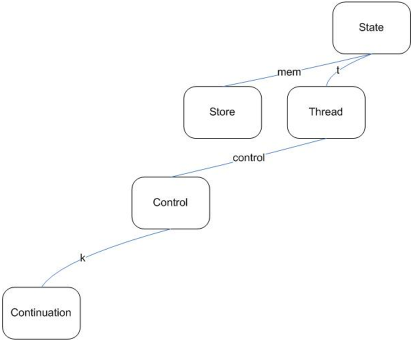
KOOL Program Representation



KOOL Program Representation: A Simple Term


```
stmt(if E then S else S' fi)
```

KOOL Program Representation: A More Complex Term




```
t(control(k(llookup(L) -> K) CS) TS) mem(Mem)
```

Sample KOOL Semantics

Equations represent non-competing transitions, and have the general form *eq* $l = r$ (unconditional) or *ceq* $l = r$ *if* c (conditional):

```
eq stmt(if E then S else S' fi) = exp(E) -> if(S,S') .
eq val(primBool(true)) -> if(S,S') = stmt(S) .
eq val(primBool(false)) -> if(S,S') = stmt(S') .
```

Sample KOOL Semantics

Equations represent non-competing transitions, and have the general form $eq\ l = r$ (unconditional) or $ceq\ l = r\ if\ c$ (conditional):

```
eq stmt(if E then S else S' fi) = exp(E) -> if(S,S') .
eq val(primBool(true)) -> if(S,S') = stmt(S) .
eq val(primBool(false)) -> if(S,S') = stmt(S') .
```

Rules represent transitions which may compete, and have the general form $rl\ l => r$ (unconditional) or $crl\ l => r\ if\ c$ (conditional):

```
crl t(control(k(llookup(L) -> K) CS) TS) mem(Mem) =>
    t(control(k(val(V) -> K) CS) TS) mem(Mem)
 if V := Mem[L] /\ V =/= undefined .
```

Sample KOOL Semantics

Equations represent non-competing transitions, and have the general form $eq\ l = r$ (unconditional) or $ceq\ l = r\ if\ c$ (conditional):

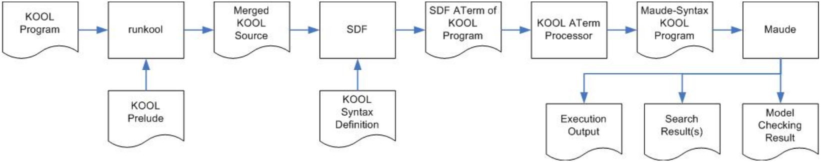
```
eq stmt(if E then S else S' fi) = exp(E) -> if(S,S') .
eq val(primBool(true)) -> if(S,S') = stmt(S) .
eq val(primBool(false)) -> if(S,S') = stmt(S') .
```

Rules represent transitions which may compete, and have the general form $rl\ l => r$ (unconditional) or $crl\ l => r\ if\ c$ (conditional):

```
crl t(control(k(llookup(L) -> K) CS) TS) mem(Mem) =>
    t(control(k(val(V) -> K) CS) TS) mem(Mem)
 if V := Mem[L] /\ V =/= undefined .
```

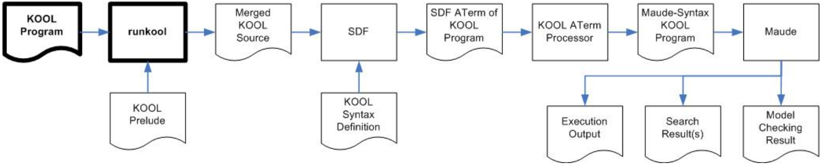
335 equations in semantics, 15 rules, 1406 lines

Running KOOL Programs



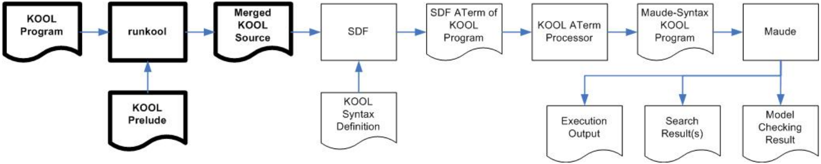
- Programs parsed, converted to Maude, and executed, with results displayed to user
- KOOL programs execute directly in the language semantics, defined using rewriting logic

Running KOOL Programs: Step 1



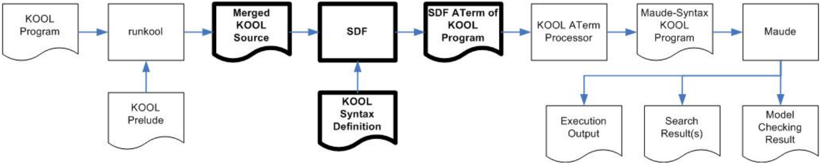
1. The KOOL program is created and `runkool` is invoked

Running KOOL Programs: Step 2



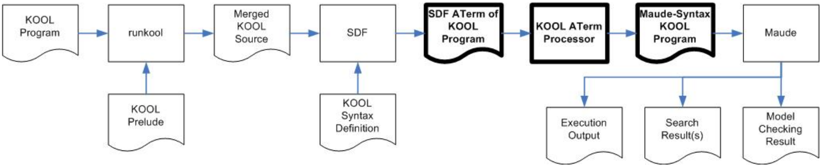
1. The KOOL program is created and `runkool` is invoked
2. `runkool` pulls in the standard prelude and generates a complete program

Running KOOL Programs: Step 3



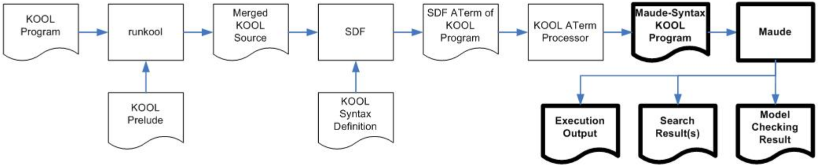
1. The KOOL program is created and `runkool` is invoked
2. `runkool` pulls in the standard prelude and generates a complete program
3. The program is parsed using SDF, generating an SDF ATerm

Running KOOL Programs: Step 4



1. The KOOL program is created and `runkool` is invoked
2. `runkool` pulls in the standard prelude and generates a complete program
3. The program is parsed using SDF, generating an SDF ATerm
4. A custom processor converts the ATerm into Maude syntax

Running KOOL Programs: Step 5



1. The KOOL program is created and `runkool` is invoked
2. `runkool` pulls in the standard prelude and generates a complete program
3. The program is parsed using SDF, generating an SDF ATerm
4. A custom processor converts the ATerm into Maude syntax
5. Maude runs the program, generating the proper output based on the requested execution mode

Running KOOL Programs: An Example

```
> runkool Factorial.kool
result String: "7886578673647905035523632139321850622951359776871732632
94742533244359449963403342920304284011984623904177212138919638830257642
79024263710506192662495282993111346285727076331723739698894392244562145
16642402540332918641312274282948532775242424075739032403212574055795686
60226031904170324062351700858796178922227896237038973747200000000000000
000000000000000000000000000000000000"
```

Outline

Analysis Overview

KOOL uses analysis capabilities of Maude to provide program analysis:

- **Search** allows a breadth-first search over the program state space
- **Model Checking** allows verification of finite-state systems using LTL formulae
- Rewriting logic *rules* determine size of state space/transitions between states

Breadth-First Search

- KOOL provides breadth-first search over output values "out-of-the-box"
- Can either find all output values or search for a specific value
- Can be useful for testing language extensions

Search Example: The Thread Game

KOOL version of a problem formulated by J. Moore

```
class ThreadGame is
  var x;

  method ThreadGame is
    x <- 1;
  end

  method Add is
    while true do x <- x + x; od
  end

  method Run is
    spawn(self.Add); spawn(self.Add);
    console << x;
  end
end
(new ThreadGame).Run
```

Thread Game Results

```
> runkool -t 5 ThreadGame.kool

Solution 1 (state 769)
SL:[StringList] --> "5"
```

Search Example: Synchronized Methods (1)

```
class WriteNum is
  var num;

  method WriteNum(n) is
    num <- n;
  end

  synchronized method set(n) is
    num <- n;
  end

  synchronized method write is
    console << "Start:" << num;
    self.set(num + 10);
    self.set(num - 8);
    console << "End:" << num;
  end
end
```

Search Example: Synchronized Methods (2)

```
class Driver is
  method run is
    var w1;
    w1 <- new WriteNum(10);
    spawn (w1.write);
    w1.set(20);
    spawn (w1.write);
  end
end

(new Driver).run
```

Results without Synchronization

```
> runkool -s --final Sync6.kool

Solution 1 (state 80383)
states: 80853  rewrites: 10112671 in 671633ms cpu (674345ms real)
    (15056 rewrites/second)
SL:[StringList] --> "Start:","20","End:","22","Start:","22","End:","24"

...

Solution 470 (state 80852)
states: 80853  rewrites: 10112671 in 671645ms cpu (674360ms real)
    (15056 rewrites/second)
SL:[StringList] --> "Start:","10","End:","Start:","20","End:","22","12"

No more solutions.
```

Results with Synchronization

```
> runkool -s --final Sync5.kool

Solution 1 (state 96)
states: 98  rewrites: 10390 in 612ms cpu (612ms real)
    (16976 rewrites/second)
SL:[StringList] --> "Start:","20","End:","22","Start:","22","End:","24"

Solution 2 (state 97)
states: 98  rewrites: 10390 in 612ms cpu (612ms real)
    (16976 rewrites/second)
SL:[StringList] --> "Start:","10","End:","12","Start:","20","End:","22"

No more solutions.
```

Model Checking

- KOOL uses Maude to provide basic model checking capabilities
- Extended with labeled statements; labels can be used in LTL formulae
- Runtime allows custom Maude modules with new LTL properties to be loaded and used during verification

Dining Philosophers

```
class Philosopher is
  method Run(id,left,right) is
    while true do
      // thinking here...
      hungry:
        acquire left;
        acquire right;
      eating:
        release left;
        release right;
    od
  end
end
```

Model Checking the Dining Philosophers

```
> runkool DP.kool -m ... model checking arguments ...
```

- Model checking arguments generally include formula to check
- When formula doesn't hold, a counterexample is generated
- When formula holds, `true` is returned

Outline

1. Rewriting Logic Semantics and KOOL

2. Analysis in KOOL with Rewriting Logic

3. Conclusion

Conclusions

- KOOL is a full-featured, pure OO language defined using rewriting logic
- Rewriting logic provides a semantics-based interpreter for running KOOL programs almost for free
- Rewriting logic and KOOL provide analysis capabilities useful for model checking, search, and testing language extensions

Future Work

- Provide GC for KOOL, which should help improve memory performance and provide a more realistic memory model
- Plug type systems into KOOL, allowing multiple type systems to be used on a single KOOL program
- Further investigate analysis performance optimization (some work on this is already done – see *On Formal Analysis of OO Languages using Rewriting Logic: Designing for Performance*, Hills and Roșu, FMOODS'07, LNCS Volume 4468, pp 107–121, 2007)

Related Work

- Rewriting Logic Semantics: *The Rewriting Logic Semantics Project*, Meseguer and Roșu, TCS, Volume 373(3), pp 217–237, 2007.
- *Formal Analysis of Java Programs in JavaFAN*, Farzan, Chen, Meseguer, and Roșu, CAV'04, LNCS Volume 3114, pp 501–505, 2004.
- *Using Maude and its strategies for defining a framework for analyzing Eden semantics*, Hidalgo-Herrero, Verdejo, and Ortega-Mallén, WRS'06, ENTCS, to appear.
- *Compiling language definitions: the ASF+SDF compiler*, van den Brand, Heering, Klint, and Olivier, ACM TOPLAS, Volume 24(4), pp 334–368, 2002.