On Formal Analysis of OO Languages using Rewriting Logic: Designing for Performance [⋆]

Mark Hills and Grigore Roşu

Department of Computer Science
University of Illinois at Urbana-Champaign, USA
201 N Goodwin Ave, Urbana, IL 61801
{mhills, grosu}@cs.uiuc.edu

Abstract. Rewriting logic provides a powerful, flexible mechanism for language definition and analysis. This flexibility in design can lead to problems during analysis, as different designs for the same language feature can cause drastic differences in analysis performance. This paper describes some of these design decisions in the context of KOOL, a concurrent, dynamic, object-oriented language. Also described is a general mechanism used in KOOL to support model checking while still allowing for ongoing, sometimes major, changes to the language definition.

keywords: object-oriented languages, programming language semantics, analysis, model checking, rewriting logic

1 Introduction

With the increase in multi-core systems, concurrency is becoming a more important topic in programming languages and formal methods research. Rewriting logic [16, 15], an extension of equational logic with support for concurrency, provides a computational logic for defining, reasoning about, and executing concurrent systems. While these can be fairly simple systems, entire programming languages, such as object-oriented languages, can be defined as rewrite theories, allowing tools designed to work with generic rewrite specifications to work with the defined programming languages as well.

While there has been much work on analysis and verification techniques with rewriting logic [18, 19, 6, 17], this work has mainly focused on laying the theoretical foundations and on proofs of concept. Exceptions to this include work on program verification for Java [7], Java bytecode in the JVM [8], and CML [3], a concurrent extension to the ML programming language.

Even with these papers focused on real languages, very little information is given on *why* certain design decisions were made. For the language designer looking to define object-oriented languages using rewriting logic, this is a major shortcoming. Since even small changes to a rewriting logic definition can have major impacts on the ability to analyze programs, making appropriate decisions when defining the language is vitally important. In addition, little information

⋆ Supported by NSF CCF-0448501 and NSF CNS-0509321.

is available about specifically object-oriented definitions; while the work on Java [7] obviously qualifies, the JVM operates at a much lower level, and the model of computation used by CML, based around the strict functional language ML, differs from that used by standard object-oriented languages.

In this paper, we have set out to fill this gap by providing information on increasing the analysis performance of rewrite logic definitions for object-oriented languages, specifically in the context of Maude [4, 5], a high-performance rewriting logic engine. We start in Section 2 by providing a brief introduction to rewriting logic, showing the relationship between rewriting logic and term rewriting and explaining the crucial distinction between equations and rules. Section 3 then provides a brief introduction to KOOL, a concurrent, object-oriented language that will be the focus of the experiments in this paper.

In Section 4, we highlight the search capabilities of Maude by showing some examples of its use. Search provides a breadth-first search over a program's state space, providing an ability to search for program states matching certain conditions (output of a certain value, safety condition violation) that, due to the potentially infinite state space of the program, may not be possible with model checking. Section 5 then discusses model checking of OO programs in rewriting logic, using the classic dining philosophers problem. To improve the performance of search and model checking, Section 6 discusses two potential performance improvements important in the context of object-oriented languages: auto-boxing of scalar values for use in a pure object-oriented language, and optimizing memory access for analysis performance. Section 7 concludes the paper.

2 Rewriting Logic

This section provides a brief introduction to term rewriting and rewriting logic. Term rewriting is a standard computational model supported by many systems; rewriting logic [16, 15] organizes term rewriting modulo equations as a complete logic and serves as a foundation for the semantics of programming languages [19–21].

2.1 Term Rewriting

Term rewriting is a method of computation that works by progressively changing (rewriting) a term. This rewriting process is defined by a number of rules – potentially containing variables – which are each of the form: $l \to r$. A rule can apply to the entire term being rewritten or to a subterm of the term. First, a match within the current term is found. This is done by finding a substitution, θ, from variables to terms such that the left-hand side of the rule, l, matches part or all of the current term when the variables in l are replaced according to the substitution. The matched subterm is then replaced by the result of applying the substitution to the right-hand side of the rule, r. Thus, the part of the current term matching $\theta(l)$ is replaced by $\theta(r)$. The rewriting process continues as long as it is possible to find a subterm, rule, and substitution such that $\theta(l)$

matches the subterm. When no matching subterms are found, the rewriting process terminates, with the final term being the result of the computation. Rewriting can continue forever, a necessity for emulating computation.

There exist a plethora of term rewriting engines, including ASF [24], Elan [1], Maude [4, 5], OBJ [9], Stratego [25], Tom [23, 14], and others. Rewriting is also a fundamental part of existing languages and theorem provers. Term rewriting is inherently parallel, since non-overlapping parts of a term can be rewritten at the same time, and thus fits well with current trends in architecture and systems.

2.2 Rewriting Logic

Rewriting logic is a computational logic built upon equational logic which provides support for concurrency. In equational logic, a number of *sorts* (types) and *equations* are defined. The equations specify which terms are considered to be equal. All equal terms can then be seen as members of the same equivalence class of terms, a concept similar to that from the λ calculus with equivalence classes based on α and β equivalence. Rewriting logic provides *rules* in addition to equations, used to transition between equivalence classes of terms. This allows for concurrency, where different orders of evaluation could lead to non-equivalent results, such as in the case of data races. The distinction between rules and equations is crucial for analysis, since terms which are equal according to equational deduction can all be collapsed into the same analysis state. Rewriting logic is connected to term rewriting in that all the equations and rules of rewriting logic, of the form $l = r$ and $l \Rightarrow r$, respectively, can be transformed into term rewriting rules by orienting them properly (necessary because equations can be used for deduction in either direction), transforming both into $l \rightarrow r$. This provides a means of taking a definition in rewriting logic and a term and "executing" it.

In this paper we focus on the use of Maude [4, 5], a rewriting logic language and engine. Beyond the ability to execute a program based on a rewriting logic definition, Maude provides several capabilities which make it useful for defining languages and performing formal analysis of programs. Maude allows commutative and associative operations with identities, allowing straight-forward definitions of language features which make heavy use of sets and lists, such as sets of classes and methods and lists of computational tasks. Maude's support for rewriting logic provides a natural way to model concurrency, with potentially competing tasks (memory accesses, lock acquisition, etc) defined as rules. Also, Maude provides built-in support for model checking and breadth-first state space exploration, which will be explored further starting in Section 4.

3 KOOL

KOOL is a concurrent, dynamic, object-oriented language, loosely inspired by, but not identical to, the Smalltalk language [10, 2]. KOOL includes support for standard imperative features, such as assignment, conditionals, and loops

Program	P ::=	$C^*\ E$
Class	C ::=	class X is D^* M^* end | class X extends X' is D^* M^* end
Decl	D ::=	var $\{X,\}^+$;
Method	M ::=	method X is D^* S end | method X ($\{X',\}^+$) is D^* S end
Expression	E ::=	X | I | F | B | Ch | Str | (E) | new X | new X ($\{E,\}^+$) | self | E X_{op} E' | $E.X(())^?$ | $E.X(\{E,\}^+$) | super() | super.$X(())^?$ | super.$X(\{E,\}^+$) | super($\{E,\}^+$)
Statement	S ::=	E <- E'; | begin D^* S end | if E then S else S' fi | if E then S fi | try S catch X S end | throw E ; | for X <- E to E' do S od | while E do S od | break; | continue; | return; | return E; | S S' | E; | assert E; | X: | typecase E of Cs^+ (else S)$^?$ end
Case	Cs ::=	case X of S

$X \in$ *Name*, $I \in$ *Integer*, $F \in$ *Float*, $B \in$ *Boolean*, $Ch \in$ *Char*, $Str \in$ *String*, $X_{op} \in$ *Operator Names*

Fig. 1. KOOL Syntax

with break and continue. KOOL also includes support for many familiar object-oriented features: all values are objects; all operations are carried out via message sends; message sends use dynamic dispatch; single inheritance is used, with a designated root class named `Object`; methods are all public, while fields are all private outside of the owning object; and scoping is static, yet declaration order for classes and methods is unimportant (all methods in a class see all other methods in the same class, for instance, and all classes see all other classes). KOOL allows for the run-time inspection of object types via a typecase construct, and includes support for exceptions with a standard try/catch mechanism.

3.1 KOOL Syntax

The syntax of KOOL is shown in Figure 1. The lexical definitions of literals are not included in the figure to limit clutter, but are standard (for instance, booleans include both true and false, strings are surrounded with double quotes and characters with single quotes, etc). Single line and block comments are both supported, using the same syntax as Java or C++, with the addition that block comments can be nested. Message sends are specified in a Java-like syntax except for methods named after operators, which are always binary and can be used infix (such as `a + b` instead of `a.+(b)`). Because of this, very few operators are predefined, and operators all have the same precedence and associativity. Sends with no parameters do not require parens except for calls to parent constructors which do not take parameters, which are of the form `super()`. Finally, semicolons are used as statement terminators, not separators, and are only needed where the end of a statement may be ambiguous – at the end of an assignment, for instance, or at the end of each statement inside a branch of a conditional, but not at the end of the conditional itself, which ends with `fi`.

```
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)
```

Fig. 2. Recursive Factorial, KOOL

To get a feel for the language, a sample program is presented in Figure 2. A new class `Factorial` is defined with a method `Fact` that calculates the factorial of the parameter `n`. After the class definition is the main program expression, which creates a new object of class `Factorial`, invokes method `Fact` with the parameter `200`, and then writes the output to the predefined `console` object using the output operation, `<<` (borrowed from C++). This operation invokes the `toString` method on its parameter and returns itself as the method result, allowing chaining of output operations (such as `console << "Value = " << 3`).

3.2 KOOL Semantics

The semantics of KOOL is defined using Maude equations and rules, with the current program represented as a "soup" of sometimes nested terms representing the current computation, memory, the environment, locks held, etc. A visual representation of this term, the state infrastructure, is shown in Figure 4; state components needed specifically for concurrency are shaded.

```
 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') .

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 .
```

Fig. 3. Sample KOOL Rules

Figure 3 shows examples of the equations and rules which make up the KOOL semantics. The first three equations (represented with `eq`) process a conditional in the expected way: the first computes the value of the guard, saving the branches for later use, while the second and third execute the appropriate branch based on whether the guard was `true` or `false`. The fourth, a conditional rule (represented with `crl`), represents the lookup of a memory location. The rule states that, if the next computation step in this thread is to look up the value at location `L`, and if that value is `V`, and if `V` is not undefined (i.e. `L` is a properly defined location), the result of the computation is the value `V`. `CS` and `TS` match the unreferenced parts of the control and thread state, respectively.

We are continually extending the language, and are also using KOOL as a basis for both teaching and research in language semantics, object-oriented languages, analysis, and verification.

3.3 KOOL Implementation

There is an implementation of KOOL available at our website [12], as well as a web-based interface to run and analyze KOOL programs such as those presented

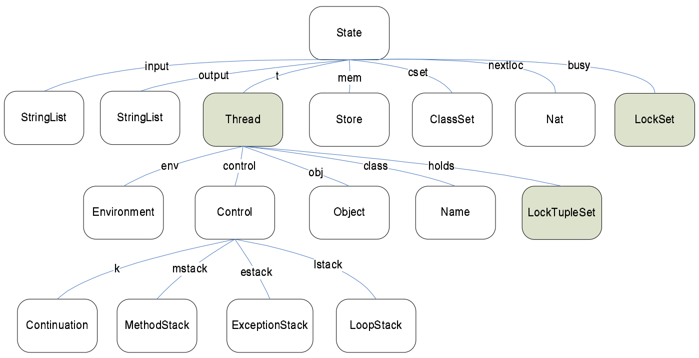

Fig. 4. KOOL State Infrastructure

here. There is also a companion technical report [11] that explains the syntax and semantics in detail.

KOOL programs are generally run using the `runkool.sh` script. Flags accepted include `-s` to perform a search and `-t` to search for a specific value. For model checking, the `mckool.sh` script is used instead of `runkool.sh` for simple runs, while `mccust.sh` is used for model checking runs that require custom Maude files. Flags accepted include `-m` to run the model checker in assertion-checking mode and `-c` to pass a custom model-checking formula. These three scripts will be combined in the near future. A KOOL program is first parsed using SDF [24] and a custom C program, translating from KOOL syntax into a form more easily handled by Maude and including any necessary headers and footers (to indicate a search execution instead of a normal execution, for instance). Maude then executes this generated text, writing the result to the console.

4 Breadth-first Search in KOOL

The thread game is a concurrency problem defined as follows: take a single variable, say `x`, initialized to 1. In two threads, repeat the assignment `x <- x + x` forever. In another thread, output the value of `x`. What values is it possible to output? As has been proved [22], it is possible to output any natural number ≥ 1. A KOOL version of the thread game is shown in Figure 5.

To check to see if a specific value can be output, one could run the program. Given enough runs, the value of interest may be generated, but this is highly inefficient. Model checking will not help here either, since this

```
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
```

Fig. 5. Thread Game, KOOL

is an infinite state system, and the value may not be along the first (depth-first) search path chosen. Maude's search capability can be used, though, either to enumerate possible values (obviously not all possible values here) or to search for a specific value. For instance, searching for 10 yields a result, indicating that 10 is one of the possible values; a sample run showing this is shown in Figure 6.

```
./runkool.sh examples/ThreadGame.kool -t 10
... term omitted ...
Solution 1 (state 2294)
states: 3381  rewrites: 310427 in 14388ms cpu
SL:[StringList] --> "10"
```

Fig. 6. Thread Game Sample Run

Another example of the usefulness of search is illustrated by the program in Figure 7. This program is finite state, so all possible results can be enumerated. When search is used here, requesting all possible final results, three are returned: both 100 and 200 can be output, and an assertion can be thrown if the thread running `Changer` sets the value to 200 between the time the value is set to 100 and the time the next line, with the `assert` statement, is executed.

```
class WrappedInt is
  var wval;

  method WrappedInt(n) is
    wval <- n;
  end

  method setWVal(n) is
    wval <- n;
  end

  method toString is
    return wval.toString();
  end

  method =(n) is
    return wval = n;
  end
end
```

```
class Changer is
  method Run(n) is
    n.setWVal(200);
  end
end

class Main is
  method Run is
    var x;
    x <- new WrappedInt(5);
    spawn ((new Changer).Run(x));
    x.setWVal(100);
    assert(x = 100);
    console << x;
  end
end
```

```
./runkool.sh examples/Spawn7.kool -s
... term omitted ...
Solution 1 (state 1964)
states: 2147  rewrites: 93343 in 6112ms cpu (6110ms real) (15271 rewrites/second)
SL:[StringList] --> "100"

Solution 2 (state 2430)
states: 2441  rewrites: 100482 in 6748ms cpu (6748ms real) (14889 rewrites/second)
SL:[StringList] --> "200"

Solution 3 (state 2490)
states: 2491  rewrites: 101721 in 6852ms cpu (6852ms real) (14844 rewrites/second)
SL:[StringList] --> "AssertException thrown: Assertion triggered"
```

Fig. 7. Assertions, KOOL

5 Model Checking KOOL

A canonical example for concurrency is the Dining Philosophers problem. A simple version of this problem, with just two philosophers, is shown written in KOOL in Figure 8. In KOOL, locks can be acquired on any object. Here we create a `Fork` class with no methods or properties; we can create objects of this class and then acquire locks on the objects, representing taking a fork. The `Philosopher` class just contains a single method, `Run`, which enters an infinite loop that cycles through two states: hungry (wants to acquire forks) and eating (has acquired forks). Once a philosopher eats, it releases the locks, putting down the forks. The `Main` class also contains a `Run` method; this method creates the necessary forks and philosophers, and then uses the `spawn` statement to run each philosopher in its own thread.

```
class Fork is
end

class Philosopher is
  method Run(id,left,right) is
    while (true) do
      hungry:
      acquire left;
      acquire right;

      eating:
      skip;
      release left;
      release right;
    od
  end
end
```

```
class Main is
  var l1, l2;
  var p1, p2;

  method Run is
    l1 <- new Fork;
    l2 <- new Fork;

    p1 <- new Philosopher;
    p2 <- new Philosopher;

    spawn(p1.Run(1,l1,l2));
    spawn(p2.Run(2,l2,l1));
  end
end

(new Main).Run
```

Fig. 8. Dining Philosophers

We would like to determine if this program can deadlock. Using Maude's model checking capabilities, we can write properties over the program state which can then be used in LTL formulae. For instance, we could create a property named `deadlocked`, and then write a formula like "`[]~deadlocked`" (it's always the case that we are not deadlocked). A problem with this is that the program state is very complex; it contains all current class definitions, runtime information for each thread, global information for the program (such as memory), and other bookkeeping information. It isn't always obvious how to properly write a property using this information. Here, for instance, we would need to detect when we are trying to acquire a fork, and say that when we start trying to acquire forks we always are able to acquire both, allowing the philosopher to eat. Another problem is that, if we change the state definition as we are modifying the language design, we risk having to change defined properties to match the new state, breaking the modularity of language definitions.

A solution that resolves these problems is to use *label statements*, shown in Figure 8 as identifiers followed by a colon (such as `hungry:` or `eating:`) to assist in model checking. This idea is used by other model checkers as well – SPIN [13], for instance, also uses labels. The language semantics then include a rule (not an equation, since this takes us into a non-equivalent state which should be detected during verification) which sets a component of the thread state to the value of the label when the label is encountered. This allows properties to be stated directly in terms of the labels – here, for instance, freedom from deadlock means that upon reaching the `hungry` label it is always the case that the thread eventually reaches the `eating` label. This requires much less detailed knowledge about the state, since only label names, included in the program source, need to be known. It also insulates model checking from state changes, as long as the part of the state dealing with labels is not modified. The tradeoff is a potential degradation of performance, since the label semantics are defined in terms of rules, and rule application adds additional states to the state space. In cases where additional performance is needed, it is still possible to write predicates directly against the state, avoiding the use of labels. Again, though, these predicates may be quite complicated, and may require ongoing maintenance as the language evolves.

Using this notion of progress for deadlock freedom, the appropriate LTL formula for the two philosopher problem is then:

$$\texttt{progress(2,hungry,eating)} \vee \texttt{progress(3,hungry,eating)}$$

where 2 and 3 are the thread IDs and `progress(n,l1,l2)` means that thread `n` eventually reaches `l2` whenever it reaches `l1`. Thread IDs are needed since LTL lacks quantification – i.e. there is no way to say that, $\forall n,$`progress(n,l1,l2)`. The thread running first has ID 1, and each spawn adds 1 to this.

```
while true do
  hungry:
  if (id % 2 = 0) then
    acquire left;
    acquire right;
  else
    acquire right;
    acquire left;
  fi

  eating:
  skip;
  release left;
  release right;
od
```

Fig. 9. Dining Philosophers, Deadlock-Free

Running the model checker with this program and formula, we will get a counterexample, since it is in fact possible to deadlock (when the first philosopher grabs the first fork and the second grabs the second). Times for the model checker to find counterexamples, by philosopher count, are given in Figure 13. A fix to the code in the `Philosopher` class `Run` method is shown in Figure 9, with "odd" philosophers taking the forks in one order and "even" philosophers in the other. Unfortunately, due to the initial language design, which focused more on executability and less on verification, it is not possible to verify this fix with the model checker – it will run for a time and then crash. This will be addressed in Section 6, where modifications to the design to improve verification performance will be explored. A sample run, after these modifications are in place, is shown for 2 philosophers, using the deadlock-free version of the code, in Figure 10. Here, `mccust` allows a custom model-checking script to be run, in this case one that defines `someWillEat`, and `-c` allows the LTL property to be specified. The result, `true`, means that the property holds.

```
./mccust.sh examples/DP2-nodeadlock.kool mcdp.maude -c "someWillEat(2,3)"
... term omitted ...
result Bool: true
```

Fig. 10. Model Checking Sample Run

6 Tuning the Model

The ability to model check and search programs using language definitions in rewriting logic is very closely tied to the performance of the definition. There are two general classes of performance improvement: improvements that impact execution speed, and improvements that impact analysis speed, which may even slightly reduce typical execution speed. Two examples of improvements are presented here. First, auto-boxing is introduced to the language. This allows operations on scalar types, which are represented in KOOL as objects, to be performed directly on the underlying values for many operations (standard arithmetic operations, for instance), while still allowing method calls to be used on an object representation of the scalar where needed. Second, memory is segregated into two pools, a shared and an unshared pool. Rules are used when accessing or modifying memory in the shared pool, since these changes could lead to data races, while equations are used for equivalent operations on the unshared pool. This follows the intuition that changes to unshared memory locations in a thread cannot cause races. This change may or may not improve execution performance, but has a dramatic impact on analysis performance.

6.1 Auto-boxing

In KOOL, all values, including those typically represented as scalars in languages like Java, are objects. This means that a number like 5 is represented as an object, and an expression like 5 + 7 is represented as a method call. Primitive operations are defined which extract the primitive values "hidden" in the objects (i.e. the actual number 5, versus the object that represents it), perform the operation on these primitive values, and create a new object representing the result. This provides a "pure" object-oriented model, but requires additional overhead, including additional accesses to memory to retrieve the primitive values and create the new object for the result. Since memory accesses are modeled as rules in the definition, this also increases model checking and search time by increasing the number of states that need to be checked.

To improve performance, auto-boxing can be added to KOOL. This allows values such as 5 to be represented as scalars – i.e. directly as the primitive values. A number of operations can then be performed directly on the primitive representation, without having to go through the additional steps described above. For numbers, this includes arithmetic and logical operations, which are some of the most common operations applied to these values. Operations which cannot be performed directly can still be treated as message sends; the scalar value is automatically converted to an object representing the same value, which can then act as a message target to handle the method. Since boxing can occur automatically, by default values, including those generated as the result of primitive operations, are left un-boxed – in scalar form. This all happens behind the scenes, allowing KOOL programs to remain unchanged.

An example of the rule changes to enable auto-boxing are found in Figure 11. The first equation is without auto-boxing. Here, when a floating point number is encountered, a new floating point object of class `Float` is automatically created, with the value then "hidden" inside it. In the second equation, this value is instead wrapped inside a value wrapper – no new object is created. The next equation shows an example of an intercepted method call – invoking `+` with two float values will cause the built-in float `+` operation to be used, instead of requiring a method call to process the request. In the last equation, the boxing step is shown – here, a method outside of those handled directly on scalars has been called with the floating-point number as the target, in which case a new object will be created just like in the first equation (`[owise]` will ensure that we will try this as a last resort). The `blockWList` operation is used to hold the other values until the `Float` object is created, at which point they can be used in the invocation (the necessity for this is based on the structure of the definition, not on the fact that the floating point number is boxed).

Auto-boxing has a significant impact on performance. Figure 13 shows the updated figures for verification times with this change in place. Not only is this faster than the solution without auto-boxing in all cases, but it is now also possible to verify deadlock freedom for up to 5 philosophers, which was not possible with the prior definition.

```
  eq k(exp(f(F)) -> K) = k(exp(new Float(empty)) -> storePrim(primFloat(F)) -> K) .
--------------------------------------------------------------------------------------------
  eq k(exp(f(F)) -> K) = k(val(fv(F)) -> K) .
  eq k(val(fv(F),fv(F'')) -> toInvoke(n('+)) -> K) = k(val(fv(F + F')) -> K) .
  eq k(val(fv(F),Vl) -> toInvoke(Xm) -> K) =
     k(newPrimFloat(primFloat(F)) -> blockWList(Vl) -> toInvoke(Xm) -> K) [owise] .
```

Fig. 11. Example Definition Changes, Auto-boxing

6.2 Memory Pools

Memory in the KOOL definition is represented using a single global store for an entire program. This is fairly efficient for normal execution, but for model checking and search this can be more expensive than needed. This is because all interaction with the store must use rules, since multiple threads could compete to access the same memory location at the same time. However, many memory accesses don't compete – for instance, when a new thread is started by spawning a method call, the method's instance variables are only seen by this new thread, not by the thread that spawned it. What is needed, then, is a modification to the definition that will allow rules to be used where they are needed – for memory accesses that could compete – while allowing equations to be used for the rest.

To do this, memory in KOOL can be split into two pools: a shared memory pool, containing all memory accessible by more than one thread at some point during execution, and a non-shared memory pool, containing memory that is known to be accessed by at most one thread. To add this to the definition, an additional global state component is added to represent the shared memory pool, and the appropriate rules are modified to perform memory operations against the proper memory pool. Properly moving memory locations between the pools does require care, however, since accidentally leaving memory in the non-shared pool could cause errors during verification.

The strategy we take here is a conservative one: any memory location that *could* be accessed by more than one thread, regardless of whether this *actually* happens during execution, will be moved into the shared pool. There are two scenarios to consider. In the first, the spawn statement executes a message send. In this scenario, locations accessible through the message target (an object), as well as locations accessible through the actual parameters of the call, are all moved into the shared pool. Note that accessible here is transitive – an object passed as a parameter may contain references to other objects, all of which could be reached through the containing object. In many cases this will be more conservative than necessary; however, there are many situations, such as multiple spawns of message sends on the same object, and spawns of message sends on `self`, where this will be needed.

The second scenario is where the spawn statement is used to spawn a new thread containing an arbitrary expression. Here, all locations accessible in the current environment need to be moved to the shared pool. This includes all locations accessible in the currently executing method, including locations for instance variables, as well as any accessible through `self`. This covers all cases, including those with message sends embedded in larger expressions (since the target is in scope, either directly or through another object reference, it will be moved to the shared pool).

This strategy leads to a specific style of programming that should improve verification performance: message sends should be spawned, not arbitrary expressions, and needed information should be passed in the spawn statement to the target, instead of set through setters or in the constructor. This is because the object-level member variables will be shared, while instance variables and formal parameters will not. This brings up a subtle but important distinction – the objects referenced by the formal parameters will be shared, but not the references, which are local to the method, meaning that no verification performance penalty is paid until the code needs to "look inside" the referenced object. Looking inside does not include retrieving the object to use it in a lock acquisition statement (acquisition itself is a rule).

Figure 12 shows one of the two rules changed to support the memory pools (the other, for assignment, is similar), as well as part of the location reassignment logic. The first rule, which is the original lookup rule, retrieves a value `V` from a location `L` in memory `Mem`. The location must exist, which accounts for the condition – if `L` does not exist, looking up the current value with `Mem[L]` will

```
 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 .
-------------------------------------------------------------------------------------------
 ceq 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 .

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

 ceq t(control(k(reassign(L,Ll) -> K) CS) TS) mem(Mem) smem(SMem) =
     t(control(k(reassign(Ll,Ll') -> K) CS) TS) mem(unset(Mem,L)) smem(SMem[L <- V])
  if V := Mem[L] /\ V =/= undefined /\ Ll' := valLocs(V) .

 ceq t(control(k(reassign(L,Ll) -> K) CS) TS) mem(Mem) smem(SMem) =
     t(control(k(reassign(Ll) -> K) CS) TS) mem(Mem) smem(SMem)
  if V := SMem[L] /\ V =/= undefined .

  eq k(reassign(empty) -> K) = k(K) .
```

Fig. 12. Example Definition Changes, Memory Pools

return `undefined`. `CS` and `TS` match the rest of the `control` and `thread` states, respectively. The second and third equation and rule replace this first to support the shared and unshared memory pools. The second is now an equation, since the memory under consideration is not shared. The third is a rule, since the memory is shared. This shared pool is represented with a new part of the state, `smem`. The last three equations represent the reassignment of memory locations from the unshared to the shared pool, triggered on thread creation and assignment to shared memory locations. In the first, the location and value are moved to the shared pool, as well as all locations reachable through the value `V` – none for scalars, but all directly reachable inside an object for non-scalars. As these locations are processed, locations reachable through the values at those locations will also be added. The second represents the case where the location is already shared – in this case, we just move on to the next location. The third just ends the recursive process. Even with circular data structures (i.e. objects which holds references to one another) this will terminate, since locations reachable through an object are only added when the location for that object is initially shared.

This strategy could be improved with additional bookkeeping. As can be seen in the reassignment rules, there is currently no tracking of which threads share which location, which means that once a location is shared it never becomes unshared. Tracking this information could potentially allow a finer-grained sharing mechanism as well, versus the fairly course sharing mechanism in place now. However, even with this mechanism, we still see some significant improvements in verification performance. These can be seen in Figure 13. Note that, in every case, adding the shared pool increases performance, in many cases dramatically. It also allows additional verification – checking for a counterexample works for 8 philosophers, and verifying deadlock freedom in the fixed solution can be done for up to 7 philosophers.

Ph	No Optimizations			Auto-boxing			Auto-boxing + Memory Pools		
	States	Counter	DeadFree	States	Counter	DeadFree	States	Counter	DeadFree
2	61	0.645	NA	35	0.64	0.798	7	0.621	0.670
3	1747	0.723	NA	244	0.694	3.610	30	0.637	1.287
4	47737	1.132	NA	1857	1.074	40.279	137	0.782	5.659
5	NA	6.036	NA	14378	4.975	501.749	634	1.629	34.415
6	NA	68.332	NA	111679	49.076	NA	2943	7.395	218.837
7	NA	895.366	NA	867888	555.791	NA	13670	47.428	1478.747
8	NA	NA	NA	NA	NA	NA	63505	325.151	NA

Single 3.40 GHz Pentium 4, 2 GB RAM, OpenSuSE 10.1, kernel 2.6.16.27-0.6-smp, Maude 2.2.
Times in seconds, Ph is philosopher count, Counter is time to generate counter-example, DeadFree is time to verify the program is deadlock free, state count based on Maude search results, NA means the process either crashed or was abandoned after consuming most system memory.

Fig. 13. Dining Philosophers Verification Time

7 Conclusions and Future Work

In this paper we have shown how rewriting logic can be used for verification and analysis of a non-trivial concurrent object-oriented language. We have also shown ways in which run-time and verification performance can be improved, in this case by adding auto-boxing of scalar values in a pure object-oriented language and by segregating accesses of shared and non-shared memory locations. We believe the ideas presented here can be used during the design of other rewriting logic definitions of object-oriented languages as a means to improve performance.

There is much future work in this area, some of which was touched on in the paper. Better methods of sharing and un-sharing memory would help in the analysis of longer running programs, and could potentially be used for other purposes as well, such as in the analysis of garbage collection schemes. Also, while we achieve a reduction in the state space by the use of equations to collapse equivalent states, work on techniques like partial order reduction in the context of rewriting logic specifications would help to improve performance further.

References

1. P. Borovanský, C. Kirchner, H. Kirchner, P.-E. Moreau, and C. Ringeissen. An overview of ELAN. *ENTCS*, 15, 1998.
2. Byte. Issue on Smalltalk. *Byte Magazine*, 6(8), August 1981.
3. F. Chalub and C. Braga. A Modular Rewriting Semantics for CML. In *Proceedings of the 8th. Brazilian Symposium on Programming Languages*, May 2004.
4. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Quesada. Maude: specification and programming in rewriting logic. *Theoretical Computer Science*, 285:187–243, 2002.

5. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. The Maude 2.0 System. In *Proceedings of RTA'03*, volume 2706, pages 76–87. Springer LNCS, 2003.
6. S. Eker, J. Meseguer, and A. Sridharanarayanan. The Maude LTL Model Checker. In F. Gadducci and U. Montanari, editors, *Proc. 4th. Intl. Workshop on Rewriting Logic and its Applications*, volume 71 of *ENTCS*. Elsevier, 2002.
7. A. Farzan, F. Chen, J. Meseguer, and G. Roşu. Formal analysis of Java programs in JavaFAN. In *Proceedings of CAV'04*, volume 3114 of *LNCS*, pages 501–505. Springer, 2004.
8. A. Farzan, J. Meseguer, and G. Roşu. Formal JVM code analysis in JavaFAN. In *Proceedings of AMAST'04*, volume 3116 of *LNCS*, pages 132–147, 2004.
9. J. Goguen, T. Winkler, J. Meseguer, K. Futatsugi, and J.-P. Jouannaud. Introducing OBJ. In *Software Engineering with OBJ: algebraic specification in action.* Kluwer, 2000.
10. A. Goldberg and D. Robson. *Smalltalk-80: the language and its implementation.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983.
11. M. Hills and G. Roşu. KOOL: A K-based Object-Oriented Language. Technical Report UIUCDCS-R-2006-2779, University of Illinois at Urbana-Champaign, 2006.
12. M. Hills and G. Rosu. KOOL Language Homepage. http://fsl.cs.uiuc.edu/KOOL.
13. G. J. Holzmann. The Model Checker SPIN. *IEEE Trans. Softw. Eng.*, 23(5):279–295, 1997.
14. C. Kirchner, P.-E. Moreau, and A. Reilles. Formal validation of pattern matching code. In *Proceedings of PPDP'05*, pages 187–197. ACM Press, 2005.
15. N. Martí-Oliet and J. Meseguer. Rewriting logic: roadmap and bibliography. *Theoretical Computer Science*, 285:121–154, 2002.
16. J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.
17. J. Meseguer. Software specification and verification in rewriting logic. In M. Broy and M. Pizka, editors, *Models, Algebras, and Logic of Engineering Software, Marktoberdorf, Germany, July 30 – August 11, 2002*, pages 133–193. IOS Press, 2003.
18. J. Meseguer, M. Palomino, and N. Martí-Oliet. Equational abstractions. in Proc. CADE-19, Springer LNCS, Vol. 2741, 2–16, 2003.
19. J. Meseguer and G. Roşu. Rewriting Logic Semantics: From Language Specifications to Formal Analysis Tools . In *Proceedings of IJCAR'04*, pages 1–44. Springer LNAI 3097, 2004.
20. J. Meseguer and G. Roşu. The rewriting logic semantics project. *Theoretical Computer Science*, to appear, 2006.
21. J. Meseguer and G. Roşu. The rewriting logic semantics project. In *Proceedings of SOS'05*, volume 156 of *ENTCS*, pages 27–56. Elsevier, 2006.
22. J. S. Moore. http://www.cs.utexas.edu/users/moore/publications/thread-game.html.
23. P.-E. Moreau, C. Ringeissen, and M. Vittek. A pattern matching compiler for multiple target languages. In *Proceedings of CC'03*, volume 2622 of *LNCS*, pages 61–76. Springer, 2003.
24. M. G. J. van den Brand, J. Heering, P. Klint, and P. A. Olivier. Compiling language definitions: the ASF+SDF compiler. *ACM TOPLAS*, 24(4):334–368, 2002.
25. E. Visser. Program Transf. with Stratego/XT: Rules, Strategies, Tools, and Systems. In *Domain-Specific Program Generation*, pages 216–238, 2003.

A KOOL Language Definition

This appendix contains information on the definition of KOOL. Here we show mainly the changes from the prior published definition of KOOL, available in a companion technical report [11].

A.1 SDF Parser Definition

The SDF parser now includes an `Assert` module for runtime assertions. Labels are defined in the `Stmt` module, with constructor `LabelStmt`. The `Main` module is extended to include the `Assert` module.

```
module Assert
imports Stmt Exp
exports
  context-free syntax
    "assert" Exp ";" -> Stmt { cons("Assert") }
  lexical syntax
    "assert" -> Name { reject }

module Stmt
imports Name Exp
exports
  sorts Stmt
  context-free syntax
    Exp ";" -> Stmt { cons("StmtExp") }
    "skip" ";" -> Stmt { cons("Skip") }
    Name ":" -> Stmt { cons("LabelStmt") }
  lexical syntax
    "skip" -> Name { reject }

module Main
imports Program Class Block SeqComp Assignment Conditional Loop
        Self Super Send New Number Exception TypeCase Bool Char
        String Comments Threads Prim Assert
```

A.2 Rewriting Logic Semantics

Changes shown below are modules either added or modified to support assertions (not in the already-available language definition, and just mentioned in passing in this paper), labels, auto-boxing, and memory pools.

Added Syntax The `ASSERT-SYNTAX` module adds the syntax for assertions, while `MAIN-SYNTAX` is extended to include this module.

```
fmod ASSERT-SYNTAX is
  including STMT-SYNTAX .
  including EXP-SYNTAX .

  op assert_; : Exp -> Stmt .
  op label : Name -> Stmt .
endfm

fmod MAIN-SYNTAX is
  including PROGRAM-SYNTAX .
  including CLASS-SYNTAX .
  including BLOCK-SYNTAX .
  including SEQUENCE-SYNTAX .
  including ASSIGNMENT-SYNTAX .
  including CONDITIONAL-SYNTAX .
  including LOOP-SYNTAX .
  including SELF-SYNTAX .
  including SUPER-SYNTAX .
  including SEND-SYNTAX .
  including NEW-SYNTAX .
  including SCALARS-SYNTAX .
  including EXCEPTION-SYNTAX .
  including TYPECASE-SYNTAX .
  including PRIMITIVES-SYNTAX .
  including CONCURRENCY-SYNTAX .
  including ASSERT-SYNTAX .
endfm
```

State Changes The `STATE` module has been extended to include additional details needed in the thread state, mainly the current thread label, as well as the additional, shared memory pool `smem`.

```
fmod STATE is
  including CONTINUATION .
  including VALUE .
  including LOCATION .
  including ENVIRONMENT .
  including STORE .
  including METHOD .
  including CLASS .
  including OBJ-ENVIRONMENT .
  including OBJECT .
  including STRING-LIST .
  including LOCK .

  sorts KState KStateList .
  subsort KState < KStateList .

  op empty : -> KState .
  op __ : KState KState -> KState [assoc comm id: empty] .

  op empty : -> KStateList .
  op _,_ : KStateList KStateList -> KStateList [assoc id: empty] .

  sorts MStackTuple MStackTupleList .
  subsort MStackTuple < MStackTupleList .
  op empty : -> MStackTupleList .
  op _,_ : MStackTupleList MStackTupleList -> MStackTupleList [assoc id: empty] .
  op [_,_,_,_,_] : Continuation KState Env Object Name -> MStackTuple .

  sorts EStackTuple EStackTupleList .
  subsort EStackTuple < EStackTupleList .
  op empty : -> EStackTupleList .
  op _,_ : EStackTupleList EStackTupleList -> EStackTupleList [assoc id: empty] .
  op [_,_,_,_,_,_] : Continuation KState Env Object Name Continuation -> EStackTuple .

  sorts LStackTuple LStackTupleList .
  subsort LStackTuple < LStackTupleList .
  op empty : -> LStackTupleList .
  op _,_ : LStackTupleList LStackTupleList -> LStackTupleList [assoc id: empty] .
  op [_,_,_,_] : Continuation KState Env Continuation -> LStackTuple .

  *** Control state components (nested in thread)
  op control : KState -> KState [format (b! o)] .
  op k : Continuation -> KState [format (y! o)] .
  op mstack : MStackTupleList -> KState [format (mu! o)] .
  op estack : EStackTupleList  -> KState [format (mu! o)] .
  op lstack : LStackTupleList -> KState [format (mu! o)] .

  *** Thread state components (nested in top-level)
  op t : KState -> KState [format (r! o)] .
  op env : Env -> KState [format (r! o)] .
  op cobj : Object -> KState [format (r! o)] .
  op cclass : Name -> KState [format (r! o)] .
  op holds : LockTupleSet -> KState [format (r! o)] .
  op tid : Nat -> KState [format (r! o)] .
  op lbl : Name -> KState [format (r! o)] .

  op io : KState -> KState [format (r! o)] .
  op input : StringList -> KState [format (r! o)] .
  op output : StringList -> KState [format (r! o)] .

  op counters : KState -> KState [format (r! o)] .
  op nextOid : Nat -> KState [format (r! o)] .
  op nextTid : Nat -> KState [format (r! o)] .
  op nextLoc : Nat -> KState [format (r! o)] .
  op tc : Nat -> KState [format (r! o)] .

  *** Top-level state components
  op cset : ClassSet -> KState [format (r! o)] .
  op mem : Store -> KState [format (r! o)] .
  op smem : Store -> KState [format (r! o)] .
  op busy : LockSet -> KState [format (r! o)] .
  op threads : KState -> KState [format (r! o)] .
  op aflag : Bool -> KState [format (r! o)] .

***
*** Placeholder for the starter environment
***
  op baseenv : -> Env .

endfm
```

Memory Operations The `MEMORY-OPS` module is new, and isolates all memory operations. This has changed to support shared memory, with the addition of operations on the shared memory pool and logic to reassign locations from one pool to the other. Many operations here were removed from another module, the text of which is not shown here.

```
mod MEMORY-OPS is
  including STATE .

  vars X Xc Xc' : Name . var Xl : Names . vars K K' : Continuation . vars N N' : Nat .
  vars CS CS' cns : KState . vars V V' : Value . var Vl : ValueList . vars Env Env' : Env .
  vars Mem SMem : Store . var L : Location . vars SL SL' : KStateList .
  vars O O' : Object . vars C C' : IClass . vars Ll Ll' : LocationList .
  var CI : ClassItem . var Cs : ClassSet . var OE : ObjEnv .
  var TS TS' ts : KState .

  op lassign : Location -> Continuation .
 ceq t(control(k(val(V) -> lassign(L) -> K) CS) TS) mem(Mem) =
    t(control(k(K) CS) TS) mem(Mem[L <- V])
  if Mem[L] =/= undefined .

 ceq t(control(k(val(V) -> lassign(L) -> K) CS) TS) smem(Mem) =
    t(control(k(reassign(valLocs(V)) -> sstore(L,V) -> K) CS) TS) smem(Mem)
  if Mem[L] =/= undefined .

  op sstore : Location Value -> Continuation .
  rl t(control(k(sstore(L,V) -> K) CS) TS) smem(Mem) =>
```

```
     t(control(k(K) CS) TS) smem(Mem[L <- V]) .

 crl t(control(k(val(V) -> lassign(L) -> K) CS) TS) smem(Mem) =>
     t(control(k(reassign(valLocs(V)) -> K) CS) TS) smem(Mem[L <- V])
  if Mem[L] =/= undefined .

  op llookup : Location -> Continuation .
 ceq 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 .

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

  op reassign : LocationList -> Continuation .

***
*** If L represents a location of a value in non-shared memory,
*** move it over to shared memory and also move over any reachable
*** values.
***
 ceq t(control(k(reassign(L,Ll) -> K) CS) TS) mem(Mem) smem(SMem) =
     t(control(k(reassign(Ll,Ll') -> K) CS) TS) mem(unset(Mem,L)) smem(SMem[L <- V])
  if V := Mem[L] /\ V =/= undefined /\ Ll' := valLocs(V) .

***
*** If L represents a location of a value in shared memory,
*** ignore it -- there is no need to move it over, as it is already
*** shared.
***
 ceq t(control(k(reassign(L,Ll) -> K) CS) TS) mem(Mem) smem(SMem) =
     t(control(k(reassign(Ll) -> K) CS) TS) mem(Mem) smem(SMem)
  if V := SMem[L] /\ V =/= undefined .

***
*** Inductive halt condition for reassignment
***
  eq k(reassign(empty) -> K) = k(K) .

  op bind : Names -> Continuation .
 ceq t(control(k(val(Vl) -> bind(Xl) -> K) CS) env(Env) TS) mem(Mem) counters(nextLoc(N) cns) =
     t(control(k(K) CS) env(Env[Xl <- Ll]) TS) mem(Mem[Ll <- Vl]) counters(nextLoc(N + N') cns)
  if N' := len(Xl) /\ Ll := locs(N,N') .

 ceq t(control(k(bind(Xl) -> K) CS) env(Env) TS) mem(Mem) counters(nextLoc(N) cns) =
     t(control(k(K) CS)  env(Env[Xl <- Ll]) TS) mem(Mem[Ll <*- nil]) counters(nextLoc(N + N') cns)
  if N' := len(Xl) /\ Ll := locs(N,N') .

  op valLocs : ValueList -> LocationList .
  eq valLocs(o(oenv(OE) O),Vl) = oenvLocs(OE), valLocs(Vl) .
  eq valLocs(V,Vl) = empty, valLocs(Vl) [owise] .
  eq valLocs(empty) = empty .

  op oenvLocs : ObjEnv -> LocationList .
  eq oenvLocs([X,Env] OE) = envLocs(Env), oenvLocs(OE) .
  eq oenvLocs(empty) = empty .

  op envLocs : Env -> LocationList .
  eq envLocs(_`,_(Env, X |-> L)) = L, envLocs(Env) .
  eq envLocs(empty) = empty .

endm
```

Send Semantics Changes The `SEND-SEMANTICS` module has been modified to account for auto-boxing. Method calls that can work without boxing are intercepted, with operations working on the primitive values directly. Sends that must have an object target are boxed.

```
mod SEND-SEMANTICS is
  including STATE-HELPERS .
  including EXP-SEMANTICS .
  including STMT-SEMANTICS .
  including SEND-SYNTAX .
  including EXCEPTION-SYNTAX .
  including NEW-SYNTAX .
  including PRIM-BASIC .

  vars Xm Xc Xc' : Name . vars K Km : Continuation .
  vars Xs Xs' : Names . vars Dl Dl' : DeclList .
  vars Ds Ds' : DeclSet .
  var E : Exp . var Es : Exps . var Vl : ValueList .
  var CS : KState . var CI : ClassItem . var Cs : ClassSet .
  vars O O' : Object . var Env : Env . var IC : IClass .
  var Ms : MethodSet . var TS : KState . var Mi : MethodItem .

***
*** First, evaluate the call target (which should evaluate to an
*** object) and the parameters (which can be empty).
  op toInvoke : Name -> ContinuationItem .
  op toInvokeAndWrap : Name -> ContinuationItem .
  eq exp(E . Xm ( Es )) = exp(E,Es) -> toInvoke(Xm) .

***
*** 12/18/2006 MAH
*** Put in the "hook" here to capture scalar calls and treat them
*** differently than non-scalar calls. This allows us to keep the
*** same form for all calls and allows us to use the same operators
*** for scalars and non-scalars (like + for integer addition and
*** string concatenation).
***

  vars I I' : Int . vars F F' : Float . vars B B' : Bool . var V : Value .

  eq k(val(iv(I),iv(I')) -> toInvoke(n('+)) -> K) = k(val(iv(I + I')) -> K) .
  eq k(val(iv(I),iv(I')) -> toInvoke(n('-)) -> K) = k(val(iv(I - I')) -> K) .
  eq k(val(iv(I),iv(I')) -> toInvoke(n('*)) -> K) = k(val(iv(I * I')) -> K) .
  eq k(val(iv(I),iv(I')) -> toInvoke(n('/)) -> K) = k(val(iv(I quo I')) -> K) .
  eq k(val(iv(I),iv(I')) -> toInvoke(n('%)) -> K) = k(val(iv(I rem I')) -> K) .

  eq k(val(iv(I),fv(F')) -> toInvoke(n('+)) -> K) = k(val(fv(float(I) + F')) -> K) .
  eq k(val(iv(I),fv(F')) -> toInvoke(n('-)) -> K) = k(val(fv(float(I) - F')) -> K) .
  eq k(val(iv(I),fv(F')) -> toInvoke(n('*)) -> K) = k(val(fv(float(I) * F')) -> K) .
  eq k(val(iv(I),fv(F')) -> toInvoke(n('/)) -> K) = k(val(fv(float(I) / F')) -> K) .

  eq k(val(fv(F),iv(I')) -> toInvoke(n('+)) -> K) = k(val(fv(F + float(I'))) -> K) .
  eq k(val(fv(F),iv(I')) -> toInvoke(n('-)) -> K) = k(val(fv(F - float(I'))) -> K) .
  eq k(val(fv(F),iv(I')) -> toInvoke(n('*)) -> K) = k(val(fv(F * float(I'))) -> K) .
  eq k(val(fv(F),iv(I')) -> toInvoke(n('/)) -> K) = k(val(fv(F / float(I'))) -> K) .

  eq k(val(fv(F),fv(F')) -> toInvoke(n('+)) -> K) = k(val(fv(F + F')) -> K) .
  eq k(val(fv(F),fv(F')) -> toInvoke(n('-)) -> K) = k(val(fv(F - F')) -> K) .
  eq k(val(fv(F),fv(F')) -> toInvoke(n('*)) -> K) = k(val(fv(F * F')) -> K) .
  eq k(val(fv(F),fv(F')) -> toInvoke(n('/)) -> K) = k(val(fv(F / F')) -> K) .

  eq k(val(iv(I),iv(I')) -> toInvoke(n('<)) -> K) = k(val(bv(I < I')) -> K) .
  eq k(val(iv(I),iv(I')) -> toInvoke(n('<=)) -> K) = k(val(bv(I <= I')) -> K) .
  eq k(val(iv(I),iv(I')) -> toInvoke(n('>)) -> K) = k(val(bv(I > I')) -> K) .
  eq k(val(iv(I),iv(I')) -> toInvoke(n('>=)) -> K) = k(val(bv(I >= I')) -> K) .
  eq k(val(iv(I),iv(I')) -> toInvoke(n('=)) -> K) = k(val(bv(I == I')) -> K) .
  eq k(val(iv(I),iv(I')) -> toInvoke(n('!=)) -> K) = k(val(bv(I =/= I')) -> K) .

  eq k(val(iv(I),fv(F')) -> toInvoke(n('<)) -> K) = k(val(bv(float(I) < F')) -> K) .
  eq k(val(iv(I),fv(F')) -> toInvoke(n('<=)) -> K) = k(val(bv(float(I) <= F')) -> K) .
  eq k(val(iv(I),fv(F')) -> toInvoke(n('>)) -> K) = k(val(bv(float(I) > F')) -> K) .
  eq k(val(iv(I),fv(F')) -> toInvoke(n('>=)) -> K) = k(val(bv(float(I) >= F')) -> K) .
  eq k(val(iv(I),fv(F')) -> toInvoke(n('=)) -> K) = k(val(bv(float(I) == F')) -> K) .
  eq k(val(iv(I),fv(F')) -> toInvoke(n('!=)) -> K) = k(val(bv(float(I) =/= F')) -> K) .

  eq k(val(fv(F),iv(I')) -> toInvoke(n('<)) -> K) = k(val(bv(F < float(I'))) -> K) .
  eq k(val(fv(F),iv(I')) -> toInvoke(n('<=)) -> K) = k(val(bv(F <= float(I'))) -> K) .
  eq k(val(fv(F),iv(I')) -> toInvoke(n('>)) -> K) = k(val(bv(F > float(I'))) -> K) .
  eq k(val(fv(F),iv(I')) -> toInvoke(n('>=)) -> K) = k(val(bv(F >= float(I'))) -> K) .
  eq k(val(fv(F),iv(I')) -> toInvoke(n('=)) -> K) = k(val(bv(F == float(I'))) -> K) .
  eq k(val(fv(F),iv(I')) -> toInvoke(n('!=)) -> K) = k(val(bv(F =/= float(I'))) -> K) .

  eq k(val(fv(F),fv(F')) -> toInvoke(n('<)) -> K) = k(val(bv(F < F')) -> K) .
  eq k(val(fv(F),fv(F')) -> toInvoke(n('<=)) -> K) = k(val(bv(F <= F')) -> K) .
  eq k(val(fv(F),fv(F')) -> toInvoke(n('>)) -> K) = k(val(bv(F > F')) -> K) .
  eq k(val(fv(F),fv(F')) -> toInvoke(n('>=)) -> K) = k(val(bv(F >= F')) -> K) .
  eq k(val(fv(F),fv(F')) -> toInvoke(n('=)) -> K) = k(val(bv(F == F')) -> K) .
  eq k(val(fv(F),fv(F')) -> toInvoke(n('!=)) -> K) = k(val(bv(F =/= F')) -> K) .

  eq k(val(iv(I),Vl) -> toInvoke(Xm) -> K) =
     k(newPrimInt(primInt(I)) -> boxWList(Vl) -> toInvoke(Xm) -> K) [owise] .

  eq k(val(fv(F),Vl) -> toInvoke(Xm) -> K) =
     k(newPrimFloat(primFloat(F)) -> boxWList(Vl) -> toInvoke(Xm) -> K) [owise] .

  eq k(val(bv(B),Vl) -> toInvoke(Xm) -> K) =
     k(newPrimBool(primBool(B)) -> boxWList(Vl) -> toInvoke(Xm) -> K) [owise] .

  op boxWList : ValueList -> ContinuationItem .

  eq k(val(V) -> boxWList(Vl) -> K) = k(val(V,Vl) -> K) .
***
*** Second, now that we have an object back that we can invoke on,
*** save the current context and start searching for the method
*** in the dynamic class of the object.
*** Note that we clear the loop stack so we cannot break or continue
*** out of a method called inside a loop.
***
  var ts : KState .
  op invoke : Name ValueList -> ContinuationItem .
  eq t(control(k(val(o(myclass(Xc) O), Vl) -> toInvoke(Xm) -> K) CS) cclass(Xc') cobj(O') env(Env) TS)
       cset(cls(cname(Xc) CI) Cs) =
     t(control(k(pushMStack(K,CS,Env,O',Xc') -> clearLStack -> invoke(Xm,Vl)) CS)
       cclass(Xc) cobj(myclass(Xc) O) env(baseenv) TS)
       cset(cls(cname(Xc) CI) Cs) [owise] .

  eq t(control(k(val(o(myclass(Xc) O), Vl) -> toInvokeAndWrap(Xm) -> K) CS)
       cclass(Xc') cobj(O') env(Env) TS)
       cset(cls(cname(Xc) CI) Cs) =
     t(control(k(pushMStack(discard -> val(o(myclass(Xc) O)) -> K,CS,Env,O',Xc') -> clearLStack ->
                 invoke(Xm,Vl)) CS)
       cclass(Xc) cobj(myclass(Xc) O) env(baseenv) TS)
       cset(cls(cname(Xc) CI) Cs) .

***
*** If we invoke on a nil reference (like var x ; x + 5) throw an exception
***
  eq k(val(nil, Vl) -> toInvoke(Xm) -> K) =
     k(stmt(throw (new NilPointerException(s("Attempted to invoke on a nil reference"))) ;) -> K) .

  eq k(val(nil, Vl) -> toInvokeAndWrap(Xm) -> K) =
     k(stmt(throw (new NilPointerException(s("Attempted to invoke on a nil reference"))) ;) -> K) .

***
*** If we find the method to invoke, bind the formals and any declarations and
*** invoke the method body.
***
 ceq t(control(k(invoke(Xm,Vl) -> K) CS) cclass(Xc) TS)
```

```
      cset(cls(cname(Xc) mthds(mthd(mname(Xm) mparams(Dl) mdecls(Ds) mbody(Km) Mi) Ms) CI) Cs) =
    t(control(k(val(Vl) -> bind(declListNames(Dl)) -> bind(declSetNames(Ds)) -> Km -> K) CS) cclass(Xc) TS)
      cset(cls(cname(Xc) mthds(mthd(mname(Xm) mparams(Dl) mdecls(Ds) mbody(Km) Mi) Ms) CI) Cs)
  if len(Vl) == len(declListNames(Dl)) .
 ceq t(control(k(invoke(Xm,Vl) -> K) CS) cclass(Xc) TS)
      cset(cls(cname(Xc) mthds(mthd(mname(Xm) mparams(Dl) mdecls(Ds) mbody(Km) Mi) Ms) CI) Cs) =
    t(control(k(stmt(throw (new InvalidSignatureException(s(string(getQid(Xm))))) ;) -> K) CS) cclass(Xc) TS)
      cset(cls(cname(Xc) mthds(mthd(mname(Xm) mparams(Dl) mdecls(Ds) mbody(Km) Mi) Ms) CI) Cs)
  if len(Vl) =/= len(declListNames(Dl)) .

  op getQid : Name -> Qid .
  eq getQid(n(Q)) = Q .

***
*** If we don't find the method to invoke in the current class, switch context to the parent
*** class and keep looking. Change the name to match the parent class name when the method
*** and class have the same name (we are calling a constructor).
***
  eq t(control(k(invoke(Xm,Vl) -> K) CS) cclass(Xc) TS) cset(cls(cname(Xc) pname(Xc') CI) Cs) =
     t(control(k(invoke(if Xm == Xc then Xc' else Xm fi,Vl) -> K) CS) cclass(Xc') TS)
     cset(cls(cname(Xc) pname(Xc') CI) Cs) [owise] .

***
*** If we don't find the method at all, throw a MethodNotFound exception.
***
  var Q : Qid .
  eq t(control(k(invoke(n(Q),Vl) -> K) CS) cclass(n('Object)) TS) cset(Cs) =
     t(control(k(stmt(throw (new MethodNotFoundException(s(string(Q)))) ;) -> K) CS)
       cclass(n('Object)) TS) cset(Cs) [owise] .
endm
```

Primitives Semantics Changes The PRIMITIVES-SEMANTICS module has been modified to account for auto-boxing. Now unboxed values are returned by primitive operations by default, versus creating new objects to represent the values of primitive operations.

```
mod PRIMITIVES-SEMANTICS is
  including PRIM-BASIC .
  including NEW-SEMANTICS .
  including EXCEPTION-SYNTAX .
  protecting MAP{Int,Continuation} * (sort Map{Int,Continuation} to PrimMap) .
  protecting MAP{Int,Value} * (sort Map{Int,Value} to PrimInts, op undefined to noPrimInt) .
  protecting MAP{Bool,Value} * (sort Map{Bool,Value} to PrimBools, op undefined to noPrimBool) .

  vars I I' : Int . vars F F' : Float . vars C C' : Char .
  vars B B' : Bool . vars S S' : String . var L : Location .
  vars O O' : Object . var OE : ObjEnv .
  var K : Continuation . vars TS CS : KState . var Sl : StringList .
  var N : Nat . vars E E' E1 E2 E3 : Exp .
  vars V V' V1 V2 V3 : Value . var Xc : Name .

***
*** Handle primitive invocations from the language syntax. First, get back prim index
*** and prim args.
***
  op primInvokeNullary : -> ContinuationItem .
  eq k(exp(primInvokeNullary E) -> K) = k(exp(E) -> primInvokeNullary -> K) .

  op primInvokeUnary : -> ContinuationItem .
  eq k(exp(primInvokeUnary E E1) -> K) = k(exp(E,E1) -> fetchPrimBinary -> primInvokeUnary -> K) .

  op primInvokeBinary : -> ContinuationItem .
  eq k(exp(primInvokeBinary E E1 E2) -> K) = k(exp(E,E1,E2) -> fetchPrimTernary -> primInvokeBinary -> K) .

***
*** Now use prim index to pick correct primitive operation
```

```
***
  var CI : Continuation . var PM : PrimMap . var ts : KState .

 ceq t(control(k(val(primInt(I)) -> primInvokeNullary -> K) CS) TS) primMap(PM) =
    t(control(k(CI -> K) CS) TS) primMap(PM)
  if CI := PM[I] .
 ceq t(control(k(val(primInt(I),V) -> primInvokeUnary -> K) CS) TS) primMap(PM) =
    t(control(k(val(V) -> CI -> K) CS) TS) primMap(PM)
  if CI := PM[I] .
 ceq t(control(k(val(primInt(I),V,V') -> primInvokeBinary -> K) CS) TS)primMap(PM) =
    t(control(k(val(V,V') -> CI -> K) CS) TS) primMap(PM)
  if CI := PM[I] .

 ceq t(control(k(val(iv(I)) -> primInvokeNullary -> K) CS) TS) primMap(PM) =
    t(control(k(CI -> K) CS) TS) primMap(PM)
  if CI := PM[I] .
 ceq t(control(k(val(iv(I),V) -> primInvokeUnary -> K) CS) TS) primMap(PM) =
    t(control(k(val(V) -> CI -> K) CS) TS) primMap(PM)
  if CI := PM[I] .
 ceq t(control(k(val(iv(I),V,V') -> primInvokeBinary -> K) CS) TS) primMap(PM) =
    t(control(k(val(V,V') -> CI -> K) CS) TS) primMap(PM)
  if CI := PM[I] .

***
*** Retrieve primitive components from objects.
***
  op fetchPrimUnary : -> ContinuationItem .
  eq control(k(val(o(myclass(Xc) O)) -> fetchPrimUnary -> K) CS) cobj(O') =
    control(k(olookup(pval,Xc) -> resetObjectTo(O') -> K) CS) cobj(myclass(Xc) O) .

***
*** In case the primitive is an int (used to select from the prim map), convert
*** it to a primitive here
***
  eq control(k(val(iv(I)) -> fetchPrimUnary -> K) CS) =
    control(k(val(primInt(I)) -> K) CS) .

  op fetchPrimBinary : -> ContinuationItem .
  op fetchPrimBinary : Value -> ContinuationItem .
  eq k(val(V,V') -> fetchPrimBinary -> K) =
    k(val(V') -> fetchPrimUnary -> fetchPrimBinary(V) -> K) .
  eq k(val(V') -> fetchPrimBinary(V) -> K) =
    k(val(V) -> fetchPrimUnary -> val(V') -> K) .

  op fetchPrimTernary : -> ContinuationItem .
  op fetchPrimTernary : Value -> ContinuationItem .
  eq k(val(V1,V2,V3) -> fetchPrimTernary -> K) =
    k(val(V2,V3) -> fetchPrimBinary -> fetchPrimTernary(V1) -> K) .
  eq k(val(V2,V3) -> fetchPrimTernary(V1) -> K) =
    k(val(V1) -> fetchPrimUnary -> val(V2,V3) -> K) .

***
*** Store primitive values back into an object
***
  op storePrim : Value -> ContinuationItem .
  eq control(k(val(o(myclass(Xc) O)) -> storePrim(V) -> K) CS) cobj(O') =
    control(k(val(V) -> oassignTo(pval,Xc) -> resetObjectTo(O') -> val(o(myclass(Xc) O)) -> K) CS)
      cobj(myclass(Xc) O) .

***
*** Primitive arithmetic and relational ops for ints, including
*** coersions to floats.
***
*** 12/18/2006 MAH
*** Now return scalars; these will be boxed if need be
***
  ops primIntB+ primIntB- primIntB* primIntB/ primIntB% : -> ContinuationItem .
  ops primInt> primInt>= primInt< primInt<= primInt= primInt!= : -> ContinuationItem .
  eq k(val(primInt(I),primInt(I')) -> primIntB+ -> K) = k(val(iv(I + I')) -> K) .
  eq k(val(primInt(I),primInt(I')) -> primIntB- -> K) = k(val(iv(I - I')) -> K) .
  eq k(val(primInt(I),primInt(I')) -> primIntB* -> K) = k(val(iv(I * I')) -> K) .
  eq k(val(primInt(I),primInt(I')) -> primIntB/ -> K) = k(val(iv(I quo I')) -> K) .
  eq k(val(primInt(I),primInt(I')) -> primIntB% -> K) = k(val(iv(I rem I')) -> K) .

  eq k(val(primInt(I),primFloat(F')) -> primIntB+ -> K) = k(val(fv(float(I) + F')) -> K) .
  eq k(val(primInt(I),primFloat(F')) -> primIntB- -> K) = k(val(fv(float(I) - F')) -> K) .
  eq k(val(primInt(I),primFloat(F')) -> primIntB* -> K) = k(val(fv(float(I) * F')) -> K) .
  eq k(val(primInt(I),primFloat(F')) -> primIntB/ -> K) = k(val(fv(float(I) / F')) -> K) .

  eq k(val(primInt(I),primInt(I')) -> primInt> -> K) = k(val(bv(I > I')) -> K) .
  eq k(val(primInt(I),primInt(I')) -> primInt>= -> K) = k(val(bv(I >= I')) -> K) .
  eq k(val(primInt(I),primInt(I')) -> primInt< -> K) = k(val(bv(I < I')) -> K) .
  eq k(val(primInt(I),primInt(I')) -> primInt<= -> K) = k(val(bv(I <= I')) -> K) .
  eq k(val(primInt(I),primInt(I')) -> primInt= -> K) = k(val(bv(I == I')) -> K) .
  eq k(val(primInt(I),primInt(I')) -> primInt!= -> K) = k(val(bv(I =/= I')) -> K) .

  eq k(val(primInt(I),primFloat(F')) -> primInt> -> K) = k(val(bv(float(I) > F')) -> K) .
  eq k(val(primInt(I),primFloat(F')) -> primInt>= -> K) = k(val(bv(float(I) >= F')) -> K) .
  eq k(val(primInt(I),primFloat(F')) -> primInt< -> K) = k(val(bv(float(I) < F')) -> K) .
  eq k(val(primInt(I),primFloat(F')) -> primInt<= -> K) = k(val(bv(float(I) <= F')) -> K) .
  eq k(val(primInt(I),primFloat(F')) -> primInt= -> K) = k(val(bv(float(I) == F')) -> K) .
  eq k(val(primInt(I),primFloat(F')) -> primInt!= -> K) = k(val(bv(float(I) =/= F')) -> K) .

***
*** Primitive arithmetic and relational ops for floats, including
*** coersions from ints on the second arguments.
***
  ops primFloatB+ primFloatB- primFloatB* primFloatB/ : -> ContinuationItem .
  ops primFloat> primFloat>= primFloat< primFloat<= primFloat= primFloat!= : -> ContinuationItem .

  eq k(val(primFloat(F),primFloat(F')) -> primFloatB+ -> K) = k(val(fv(F + F')) -> K) .
  eq k(val(primFloat(F),primFloat(F')) -> primFloatB- -> K) = k(val(fv(F - F')) -> K) .
  eq k(val(primFloat(F),primFloat(F')) -> primFloatB* -> K) = k(val(fv(F * F')) -> K) .
  eq k(val(primFloat(F),primFloat(F')) -> primFloatB/ -> K) = k(val(fv(F / F')) -> K) .

  eq k(val(primFloat(F),primInt(I')) -> primFloatB+ -> K) = k(val(fv(F + float(I'))) -> K) .
  eq k(val(primFloat(F),primInt(I')) -> primFloatB- -> K) = k(val(fv(F - float(I'))) -> K) .
  eq k(val(primFloat(F),primInt(I')) -> primFloatB* -> K) = k(val(fv(F * float(I'))) -> K) .
  eq k(val(primFloat(F),primInt(I')) -> primFloatB/ -> K) = k(val(fv(F / float(I'))) -> K) .

  eq k(val(primFloat(F),primFloat(F')) -> primFloat> -> K) = k(val(bv(F > F')) -> K) .
  eq k(val(primFloat(F),primFloat(F')) -> primFloat>= -> K) = k(val(bv(F >= F')) -> K) .
  eq k(val(primFloat(F),primFloat(F')) -> primFloat< -> K) = k(val(bv(F < F')) -> K) .
  eq k(val(primFloat(F),primFloat(F')) -> primFloat<= -> K) = k(val(bv(F <= F')) -> K) .
  eq k(val(primFloat(F),primFloat(F')) -> primFloat= -> K) = k(val(bv(F == F')) -> K) .
  eq k(val(primFloat(F),primFloat(F')) -> primFloat!= -> K) = k(val(bv(F =/= F')) -> K) .

  eq k(val(primFloat(F),primInt(I')) -> primFloat> -> K) = k(val(bv(F > float(I'))) -> K) .
  eq k(val(primFloat(F),primInt(I')) -> primFloat>= -> K) = k(val(bv(F >= float(I'))) -> K) .
  eq k(val(primFloat(F),primInt(I')) -> primFloat< -> K) = k(val(bv(F < float(I'))) -> K) .
  eq k(val(primFloat(F),primInt(I')) -> primFloat<= -> K) = k(val(bv(F <= float(I'))) -> K) .
  eq k(val(primFloat(F),primInt(I')) -> primFloat= -> K) = k(val(bv(F == float(I'))) -> K) .
  eq k(val(primFloat(F),primInt(I')) -> primFloat!= -> K) = k(val(bv(F =/= float(I'))) -> K) .

***
*** Primitive read/write ops for strings, numeric values, and bools
***
  var ios : KState .

  ops primRead primReadInt primReadFloat primReadBool primWrite primConcat primStrLen : -> ContinuationItem .
  rl t(control(k(val(primString(S)) -> primWrite -> K) CS) TS) io(output(Sl) ios) =>
     t(control(k(K) CS) TS) io(output(Sl,S) ios) .
  rl t(control(k(primRead -> K) CS) TS) io(input(S,Sl) ios) =>
     t(control(k(exp(new String(empty)) -> storePrim(primString(S)) -> K) CS) TS) io(input(Sl) ios) .
  rl t(control(k(primReadInt -> K) CS) TS) io(input(S,Sl) ios) =>
```

```
     t(control(k(newPrimInt(primInt(trunc(rat(S,10)))) -> K) CS) TS) io(input(Sl) ios) .
  rl t(control(k(primReadFloat -> K) CS) TS) io(input(S,Sl) ios) =>
     t(control(k(exp(new Float(empty)) -> storePrim(primFloat(float(S))) -> K) CS) TS) io(input(Sl) ios) .
  rl t(control(k(primReadBool -> K) CS) TS) io(input("true",Sl) ios) =>
     t(control(k(newPrimBool(primBool(true)) -> K) CS) TS) io(input(Sl) ios) .
  rl t(control(k(primReadBool -> K) CS) TS) io(input("false",Sl) ios) =>
     t(control(k(newPrimBool(primBool(false)) -> K) CS) TS) io(input(Sl) ios) .
  eq k(val(primString(S),primString(S')) -> primConcat -> K) =
     k(exp(new String(empty)) -> storePrim(primString(S + S')) -> K) .
  eq k(val(primString(S)) -> primStrLen -> K) = k(newPrimInt(primInt(length(S))) -> K) .

***
*** Ops to convert primitive values to strings
***
  op toStringInt : -> ContinuationItem .
  eq k(val(primInt(I)) -> toStringInt -> K) = k(exp(new String(empty)) -> storePrim(primString(string(I,10))) -> K) .

  op toStringFloat : -> ContinuationItem .
  eq k(val(primFloat(F)) -> toStringFloat -> K) = k(exp(new String(empty)) -> storePrim(primString(string(F))) -> K) .

  op toStringBool : -> ContinuationItem .
  eq k(val(primBool(true)) -> toStringBool -> K) = k(exp(new String(empty)) -> storePrim(primString("true")) -> K) .
  eq k(val(primBool(false)) -> toStringBool -> K) = k(exp(new String(empty)) -> storePrim(primString("false")) -> K) .

***
*** Ops to convert strings to primitive values
***
  op toIntString : -> ContinuationItem .
  eq k(val(primString(S)) -> toIntString -> K) = k(newPrimInt(primInt(trunc(rat(S,10)))) -> K) .

  op toFloatString : -> ContinuationItem .
  eq k(val(primString(S)) -> toFloatString -> K) = k(exp(new Float(empty)) -> storePrim(primFloat(float(S))) -> K) .

  op toBoolString : -> ContinuationItem .
  eq k(val(primString("true")) -> toBoolString -> K) = k(newPrimBool(primBool(true)) -> K) .
  eq k(val(primString("false")) -> toBoolString -> K) = k(newPrimBool(primBool(false)) -> K) .

***
*** Ops to speed up primitives, so we only create each one once -- especially
*** useful for invariant primitive objects like integers and booleans, where we
*** cannot change the primitive value from the outside
***
  op primInts : PrimInts -> KState [format (mu! o)] .

  var PIs : PrimInts .

 ceq t(control(k(newPrimInt(primInt(I)) -> K) CS) TS) primInts(PIs) =
     t(control(k(val(V) -> K) CS) TS) primInts(PIs)
  if V := PIs[I] /\ V =/= noPrimInt .
 ceq t(control(k(newPrimInt(primInt(I)) -> K) CS) TS) primInts(PIs) =
     t(control(k(exp(new Integer(empty)) -> storePrim(primInt(I)) -> addPI(I) -> K) CS) TS) primInts(PIs)
  if PIs[I] == noPrimInt .

  op addPI : Int -> ContinuationItem .
  eq t(control(k(val(V) -> addPI(I) -> K) CS) TS) primInts(PIs) =
     t(control(k(val(V) -> K) CS) TS) primInts(insert(I,V,PIs)) .

  op primBools : PrimBools -> KState [format (mu! o)] .

  var PBs : PrimBools .

 ceq t(control(k(newPrimBool(primBool(B)) -> K) CS) TS) primBools(PBs) =
     t(control(k(val(V) -> K) CS) TS) primBools(PBs)
  if V := PBs[B] /\ V =/= noPrimBool .
 ceq t(control(k(newPrimBool(primBool(B)) -> K) CS) TS) primBools(PBs) =
     t(control(k(exp(new Boolean(empty)) -> storePrim(primBool(B)) -> addPB(B) -> K) CS) TS) primBools(PBs)
  if PBs[B] == noPrimBool .
  op addPB : Bool -> ContinuationItem .
  eq t(control(k(val(V) -> addPB(B) -> K) CS) TS) primBools(PBs) =
     t(control(k(val(V) -> K) CS) TS) primBools(insert(B,V,PBs)) .

  eq t(control(k(newPrimFloat(primFloat(F)) -> K) CS) TS) =
     t(control(k(exp(new Float(empty)) -> storePrim(primFloat(F)) -> K) CS) TS) .
***
*** Define the primitive map, which maps numbers to primitive functions.
***
  op primMap : PrimMap -> KState [format (mu! o)] .

***
*** The initial map of built-in primitives.
***
  op Prims : -> PrimMap .
  eq Prims = (1 |-> primIntB+), (2 |-> primIntB-), (3 |-> primIntB*), (4 |-> primIntB/),
             (5 |-> primIntB%), (6 |-> primInt<), (7 |-> primInt<=), (8 |-> primInt>),
             (9 |-> primInt>=), (10 |-> primInt=), (11 |-> primInt!=), (12 |-> toStringInt),
             (13 |-> primFloatB+), (14 |-> primFloatB-), (15 |-> primFloatB*), (16 |-> primFloatB/),
             (17 |-> primFloat<), (18 |-> primFloat<=), (19 |-> primFloat>), (20 |-> primFloat>=),
             (21 |-> primFloat=), (22 |-> primFloat!=), (23 |-> toStringFloat), (24 |->  toStringBool),
             (25 |->  primWrite), (26 |->  primRead), (27 |->  primReadInt), (28 |->  primReadFloat),
             (29 |->  primReadBool), (30 |->  primConcat), (31 |->  primStrLen), (32 |->  toIntString),
             (33 |->  toFloatString), (34 |->  toBoolString) .


endm
```

Scalars Semantics Changes The `SCALARS-SEMANTICS` module has been modified to not box scalar values by default.

```
***
*** 12/18/2006 MAH
*** Overriding the automatic conversion of scalars into objects. Instead, we will
*** keep scalars for integers, floats, and booleans, and we will then box and unbox
*** them when needed.
***
mod SCALARS-SEMANTICS is
  including STATE-HELPERS .
  including EXP-SEMANTICS .
  including SCALARS-SYNTAX .
  including NEW-SEMANTICS .
  including PRIMITIVES-SEMANTICS .

  var I : Int . var F : Float . var B : Bool .
  var C : Char . var S : String . var K : Continuation .

***
*** When we find a string or character scalar, create a new object for it
***
  eq k(exp(c(C)) -> K) = k(exp(new Char(empty)) -> storePrim(primChar(C)) -> K) .
  eq k(exp(s(S)) -> K) = k(exp(new String(empty)) -> storePrim(primString(S)) -> K) .

***
*** When we find an integer, float, or boolean scalar, keep it in scalar form,
*** boxing it later if need be (to call methods on it, for instance)
***
  eq k(exp(i(I)) -> K) = k(val(iv(I)) -> K) .
  eq k(exp(f(F)) -> K) = k(val(fv(F)) -> K) .
  eq k(exp(b(B)) -> K) = k(val(bv(B)) -> K) .

***  eq k(exp(i(I)) -> K) = k(newPrimInt(primInt(I)) -> K) .
***  eq k(exp(f(F)) -> K) = k(newPrimFloat(primFloat(F)) -> K) .
***  eq k(exp(b(B)) -> K) = k(newPrimBool(primBool(B)) -> K) .

endm
```

Conditional Semantics Changes The `CONDITIONAL-SEMANTICS` module has been modified to allow scalar values for booleans to be used in the conditional to determine which branch to take.

```
***
*** 12/18/2006 MAH
*** Added unboxed values for true and false for selection of proper if clause.
***
mod CONDITIONAL-SEMANTICS is
  including STATE-HELPERS .
  including EXP-SEMANTICS .
  including STMT-SEMANTICS .
  including SCALARS-SEMANTICS .
  including CONDITIONAL-SYNTAX .
  including PRIMITIVES-SEMANTICS .

  op skip : -> Stmt .
  op if : Stmt Stmt -> ContinuationItem .

  var E : Exp . vars S S' : Stmt . var V : Value .
  var K : Continuation . var O : Object .

  eq if E then S fi = if E then S else skip fi .
  eq k(stmt(skip) -> K) = k(K) .
  eq stmt(if E then S else S' fi) = exp(E) -> if(S,S') .
  eq val(o(O)) -> if(S,S') = val(o(O)) -> fetchPrimUnary -> if(S,S') .
  eq val(primBool(true)) -> if(S,S') = stmt(S) .
  eq val(primBool(false)) -> if(S,S') = stmt(S') .
  eq val(bv(true)) -> if(S,S') = stmt(S) .
  eq val(bv(false)) -> if(S,S') = stmt(S') .
endm
```

Typecase Semantics Changes The `TYPECASE-SEMANTICS` module has been modified to allow the typecase statement to work with unboxed values.

```
mod TYPECASE-SEMANTICS is
  including STATE-HELPERS .
  including EXP-SEMANTICS .
  including STMT-SEMANTICS .
  including TYPECASE-SYNTAX .
  including CLASS .
  including SCALARS-SEMANTICS .

  var E : Exp . var Cs : Cases . var C : Case . var EC : ElseCase .
  vars Xc Xc' Xc'' : Name . var O : Object .
  vars CSet CSet' CSet'' : ClassSet .
  vars Xs Xs' Xs'' : NameSet . var S : Stmt .
  var K : Continuation . var CS : KState . var CI : ClassItem .
  var TS : KState . var ts : KState .
  var I : Int . var F : Float . var B : Bool .

  op typecase : Cases -> ContinuationItem .
  op elsecase : ElseCase -> ContinuationItem .
  op noelse : -> ContinuationItem .
  op case : Case -> ContinuationItem .
  op getInherits : Name -> ContinuationItem .
  op buildInherits : Name -> ContinuationItem .
  op inherits : NameSet -> Continuation .
  op discardelse : -> ContinuationItem .

  eq stmt(typecase E of Cs end) = exp(E) -> typecase(Cs) -> noelse .
  eq stmt(typecase E of Cs EC end) = exp(E) -> typecase(Cs) -> elsecase(EC) .

  eq val(iv(I)) -> typecase(Cs) = getInherits(Integer) -> typecase(Cs) .
  eq val(fv(F)) -> typecase(Cs) = getInherits(Float) -> typecase(Cs) .
  eq val(bv(B)) -> typecase(Cs) = getInherits(Boolean) -> typecase(Cs) .
  eq val(o(O myclass(Xc))) -> typecase(Cs) = getInherits(Xc) -> typecase(Cs) .
  eq inherits(Xs) -> typecase(C, Cs) = inherits(Xs) -> case(C) -> typecase(Cs) .
  eq inherits(Xc Xs) -> case(case Xc of S) -> typecase(Cs) = stmt(S) -> discardelse .
  eq inherits(Xc Xs) -> case(case Xc' of S) -> typecase(Cs) = inherits(Xc Xs) -> typecase(Cs) [owise] .
  eq inherits(Xc Xs) -> typecase(empty) = inherits(Xc Xs) .
  eq inherits(Xc Xs) -> elsecase(else S) = stmt(S) .
  eq inherits(Xc Xs) -> noelse -> K = K .
  eq discardelse -> elsecase(EC) -> K = K .
  eq discardelse -> noelse -> K = K .

***
*** If we have already calculated the inherits set, just grab it
*** back out of the class definition.
 ceq t(control(k(getInherits(Xc) -> K) CS) TS) cset(cls(cname(Xc) inheritsSet(Xs) CI) CSet) =
     t(control(k(inherits(Xs) -> K) CS) TS) cset(cls(cname(Xc) inheritsSet(Xs) CI) CSet)
  if Xs =/= (emptyset).NameSet .

***
*** If not, start to calculate it back up towards the root. The advantage here is we
*** can build the entire path at once, so we don't have to revisit this later. Note:
*** we specify Object's inheritsSet in the class definition, so this will not
*** recurse forever or "fall off" the end.
***
 ceq t(control(k(getInherits(Xc) -> K) CS) TS) cset(cls(cname(Xc) pname(Xc') inheritsSet(Xs) CI) CSet) =
     t(control(k(getInherits(Xc') -> buildInherits(Xc) -> K) CS) TS) cset(cls(cname(Xc) pname(Xc')
       inheritsSet(Xs) CI) CSet)
  if Xs == (emptyset).NameSet .

***
*** When we have calculated the name set back up towards the root, save it
*** and pass it on.
***
  eq t(control(k(inherits(Xs) -> buildInherits(Xc) -> K) CS) TS) cset(cls(cname(Xc) inheritsSet(Xs') CI) CSet) =
     t(control(k(inherits(Xs Xc) -> K) CS) TS) cset(cls(cname(Xc) inheritsSet(Xs Xc) CI) CSet) .
endm
```

Concurrency Semantics Changes The `CONCURRENCY-SEMANTICS` module has been modified to work properly with memory pools.

```
mod CONCURRENCY-SEMANTICS is
  including STATE-HELPERS .
  including STMT-SEMANTICS .
  including EXP-SEMANTICS .
  including CONCURRENCY-SYNTAX .
  including SEND-SEMANTICS .
  including SUPER-SEMANTICS .

  var E : Exp . var Es : Exps . vars Xm Xc : Name . var Vl : ValueList .
  var K : Continuation . vars CS TS SS cns ts : KState . var LTS : LockTupleSet .
  var V : Value . var LS : LockSet . vars N M N' : Nat . var O : Object .
  var CI : ClassItem . var Cs : ClassSet . var Env : Env . var L : Location .
  var X : Name . var OE : ObjEnv .


***
*** Standard spawn -- run a method call in a new thread
***
  eq t(control(k(stmt(spawn E . Xm ( Es ) ;) -> K) CS) holds(LTS) TS) =
     t(control(k(exp(E,Es) -> toInvokeAndSpawn(Xm) -> K) CS) holds(LTS) TS) .

***
*** Shared memory strategy -- we will use a very simple shared memory strategy
*** for spawned threads. All locations reachable through the target object,
*** plus all locations reachable through the value list Vl, will be made
*** shared. This is conservative, albeit overly general, and works in
*** difficult cases -- spawning calls on self, multiple spawns on the same
*** object, etc. This means that the best performance will be using variables
*** local to a method, since those will not be shared by default.
***
*** One question that may come up -- why reassign twice? This ensures that,
*** before either thread accesses memory, reallocation has taken place, the risk
*** being that the spawning thread could access memory before it is marked shared
***
  op toInvokeAndSpawn : Name -> ContinuationItem .
  rl t(control(k(val(o(myclass(Xc) O), Vl) -> toInvokeAndSpawn(Xm) -> K) CS) TS) cset(cls(cname(Xc) CI) Cs)
       counters(nextTid(N) tc(N') cns) =>
     t(control(k(reassign(valLocs(o(myclass(Xc) O), Vl)) -> K) CS) TS)
     t(control(k(reassign(valLocs(o(myclass(Xc) O), Vl)) -> invoke(Xm,Vl) -> die)
       mstack(empty) estack(empty) lstack(empty))
       cclass(Xc) cobj(myclass(Xc) O) env(baseenv) holds(empty) tid(N) lbl(n('init)))
       cset(cls(cname(Xc) CI) Cs) counters(nextTid(s(N)) tc(s(N')) cns) .
  eq k(val(nil, Vl) -> toInvokeAndSpawn(Xm) -> K) =
     k(stmt(throw (new NilPointerException(s("Attempted to invoke on a nil reference"))) ;) -> K) .

***
*** When we spawn a new thread, just hand control over to the existing method
*** invocation continuation items. Start with empty stacks, so we cannot
*** "throw" our way out of a thread, for instance.
***
*** MAH 1/3/2007 Now an "arbitrary" spawn, not covering method calls. This allows
*** arbitrary expressions to be spawned in a new thread, like 1 + 2, if this is needed
*** for some reason. This can be expensive for analysis, since this makes all
*** locations reachable from the current context shared.
***
  rl t(control(k(stmt(spawn E ;) -> K) CS) cobj(O) env(Env) holds(LTS) tid(N) TS) counters(nextTid(M) tc(N') cns) =>
     t(control(k(K) CS) holds(LTS) tid(N) cobj(O) env(Env) TS)
     t(control(k(reassign(valLocs(o(O)),envLocs(Env)) -> exp(E) -> die) mstack(empty) estack(empty) lstack(empty))
       holds(empty) cobj(O) env(Env) tid(M) TS) counters(nextTid(s(M)) tc(s(N')) cns) .

***
*** To acquire a lock, we need to check to see if we already hold it. If
*** so, increment the counter. If not, wait until it is available.
***
  op acquire : -> ContinuationItem .
  eq stmt(acquire E ;) = exp(E) -> acquire .
  eq control(k(val(V) -> acquire -> K) CS) holds(LTS [lk(V),N]) =
     control(k(K) CS) holds(LTS [lk(V),s(N)]) .
 crl t(control(k(val(V) -> acquire -> K) CS) holds(LTS) TS) busy(LS) =>
     t(control(k(K) CS) holds(LTS [lk(V),1]) TS) busy(LS lk(V))
  if notin(LS,lk(V)) .

***
*** To release a lock, decrement the counter, unless it is already 1. If
*** it is, remove it from the busy set as well.
***
  op release : -> ContinuationItem .
  eq stmt(release E ;) = exp(E) -> release .
  eq t(control(k(val(V) -> release -> K) CS) holds(LTS [lk(V),1]) TS) busy(LS lk(V)) =
     t(control(k(K) CS) holds(LTS) TS) busy(LS) .
  eq t(control(k(val(V) -> release -> K) CS) holds(LTS [lk(V),s(N)]) TS) busy(LS lk(V)) =
     t(control(k(K) CS) holds(LTS [lk(V),N]) TS) busy(LS lk(V)) [owise] .

***
*** When a thread dies, remove it. Also, release all its
*** locks.
***
  op die : -> ContinuationItem .
  eq t(control(k(val(V) -> die) CS) holds(LTS) TS) busy(LS) counters(tc(s(N)) cns) =
     busy(LS - LTS) counters(tc(N) cns) .
  eq t(control(k(val(V) -> popMStack -> die) CS) holds(LTS) TS) busy(LS) counters(tc(s(N)) cns) =
     busy(LS - LTS) counters(tc(N) cns) .

endm
```

Assert Semantics The ASSERT-SEMANTICS module is new, and provides support for both the assert statement and labels.

```
mod ASSERT-SEMANTICS is
  including STATE-HELPERS .
  including STMT-SEMANTICS .
  including EXP-SEMANTICS .
  including ASSERT-SYNTAX .
  including PRIMITIVES-SEMANTICS .

  var E : Exp . var K : Continuation . vars X X' : Name .
  vars CS TS ts : KState . var V : Value . var B : Bool .

  eq k(stmt(assert(E) ;) -> K) = k(exp(E) -> assert -> K) .

  op assert : -> Continuation .
  eq t(control(k(val(primBool(true)) -> assert -> K) CS) TS) = t(control(k(K) CS) TS) .
  rl t(control(k(val(primBool(false)) -> assert -> K) CS) TS) aflag(B) =>
     t(control(k(stmt(throw (new AssertException(s("Assertion triggered"))) ;) -> K) CS) TS) aflag(true) .
  eq t(control(k(val(bv(true)) -> assert -> K) CS) TS) = t(control(k(K) CS) TS) .
  rl t(control(k(val(bv(false)) -> assert -> K) CS) TS) aflag(B) =>
     t(control(k(stmt(throw (new AssertException(s("Assertion triggered"))) ;) -> K) CS) TS) aflag(true) .
 crl t(control(k(val(V) -> assert -> K) CS) TS) aflag(B) =>
     t(control(k(stmt(throw (new AssertException(s("Non-boolean assert condition!"))) ;) -> K) CS) TS) aflag(true)
  if isBool(V) == false .
  rl t(control(k(stmt(label(X)) -> K) CS) lbl(X') TS) => t(control(k(K) CS) lbl(X) TS) .

  op isBool : Value -> Bool .
  eq isBool(primBool(B)) = true .
  eq isBool(bv(B)) = true .
  eq isBool(V) = false [owise] .

endm
```

Program Semantics Changes The PROGRAM-SEMANTICS module has been modified to include the new ASSERT-SEMANTICS module.

```
mod PROGRAM-SEMANTICS is
  including STATE-HELPERS .
  including PROGRAM-SYNTAX .
  including CLASS-SEMANTICS .
  including EXP-SEMANTICS .
  including NEW-SEMANTICS .
  including PRIMITIVES-SEMANTICS .
  including SCALARS-SEMANTICS .
  including SELF-SEMANTICS .
  including SEND-SEMANTICS .
  including NAME-SEMANTICS .
  including ASSIGNMENT-SEMANTICS .
  including BLOCK-SEMANTICS .
  including SEQUENCE-SEMANTICS .
  including CONDITIONAL-SEMANTICS .
  including SUPER-SEMANTICS .
  including EXCEPTION-SEMANTICS .
  including TYPECASE-SEMANTICS .
  including LOOP-SEMANTICS .
  including CONCURRENCY-SEMANTICS .
  including ASSERT-SEMANTICS .
endm
```

Main Semantics Changes The MAIN-SEMANTICS module has been modified to correctly set up the initial state with added state components.

```
mod MAIN-SEMANTICS is
  including PROGRAM-SEMANTICS .
  including PROGRAM-PRELUDE .
  including MAIN-SYNTAX .

  var P : Program . vars TS CS S ios ts : KState .
  var Sl : StringList . var E : Exp . var Cs : Classes .

  op eval : Program -> [StringList] .
  op eval* : Program -> [StringList] .
  op evalI : Program StringList -> [StringList] .
  op evalI* : Program StringList -> [StringList] .
  op evalS : Program -> KState .
  op stop : -> ContinuationItem .

***
*** This case handles the scenario where an exception is thrown back to the
*** top -- i.e. there is no handler. It is here since everything we need
*** is already visible at this point.
***
  var Vl : ValueList . var K : Continuation . var Xc' : Name .
  var O' : Object . var Env : Env . var V : Value .

  op join : ValueList -> ContinuationItem .
  eq val(V) -> join(Vl) = val(V,Vl) .

  eq control(k(val(Vl) -> popAndRunEStack -> K) estack(empty) CS)
      env(Env) cobj(O') cclass(Xc') =
     control(k(exp(n('console)) -> join(Vl) -> toInvoke(n('<<)) -> discard -> stop) estack(empty) CS)
       env(baseenv) cobj(consoleobj) cclass(Console) .

  op initState : Program StringList -> KState .
  eq initState(Cs E, Sl) =
              t(control(k(exp(E) -> discard -> stop) mstack(empty) estack(empty) lstack(empty))
              env(baseenv)  cobj(consoleobj) cclass(Console) holds(empty) tid(1) lbl(n('init)))
              io(input(Sl) output(empty))
      smem(preludeStore) mem(empty)  primInts(empty) primBools(empty) cset(process(Cs)
              preludeClasses) busy(empty)
      primMap(Prims) counters(nextOid(2) nextLoc(1) nextTid(2) tc(1)) aflag(false)  .

  eq eval(Cs E) = getOutput(initState(Cs E, empty)) .
  eq eval*(Cs E) = getOutput*(initState(Cs E, empty)) .
  eq evalS(Cs E) = getFinalKState(initState(Cs E, empty)) .
  eq evalI(Cs E,Sl) = getOutput(initState(Cs E, Sl)) .
  eq evalI*(Cs E,Sl) = getOutput*(initState(Cs E, Sl)) .

  op getOutput : KState -> [StringList] .
  op getOutput* : KState -> [StringList] .

  eq getOutput(t(control(k(stop) CS) TS) io(output(Sl) ios) S) = Sl .
  eq getOutput*(t(control(k(stop) CS) TS) io(output(Sl) ios) tc(1) S) = Sl .

  op getFinalKState : KState -> KState .
  eq getFinalKState(t(control(k(stop) CS) TS) S) = t(control(k(stop) CS) TS) S .
endm
```

KOOL Model Checking Module `KOOL-MODELCHECK` has been added to provide basic model checking primitives for use in constructing model checking formulas, mainly related to assertions and labels.

```
mod KOOL-MODELCHECK is
  including MODEL-CHECKER .
  including MAIN-SEMANTICS .
  subsort KState < State .
  op assertTriggered : State -> [Bool] .
  op assertFailed : -> Prop .

  vars S TS ios : KState . var N : Nat . vars X X' : Name .

  eq assertTriggered(S aflag(true)) = true .
  eq assertTriggered(S aflag(false)) = false .

  eq S |= assertFailed = assertTriggered(S) .

  op labeled : Nat Name -> Prop .
  eq t(tid(N) lbl(X) TS) S |= labeled(N,X) = true .

  op progress : Nat Name Name -> Prop .
  eq progress(N,X,X') = labeled(N,X) |-> labeled(N,X') .
endm
```

KOOL Model Checking, Dining Philosophers This Module is a custom model checking module used to provide functionality specifically for the dining philosophers.

```
mod KOOL-MODELCHECK-DP is
  including KOOL-MODELCHECK .

  vars N M : Nat .

  op someWillEat : Nat Nat -> Prop .
  eq someWillEat(N,N) = progress(N,n('hungry),n('eating)) .
  eq someWillEat(N,M) = progress(N,n('hungry),n('eating)) \/ someWillEat(s(N),M) [owise] .
endm
```