FMOODS'07, LNCS 4468, pp 107-121, 2007

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
http://fsl.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.

Key words: object-oriented languages, language design, 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 [14, 13], 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 [16, 17, 5, 15], much of this work has not focused on programming languages, or has used simpler, sometimes trivial, languages. Exceptions to this include work on program verification for Java [6], Java bytecode in the JVM [7], and CML [2], 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

FMOODS'07, LNCS 4468, pp 107-121, 2007

is available about specifically object-oriented definitions; while the work on Java [6] 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 start filling this gap by providing information on increasing the analysis performance of rewrite logic definitions for object-oriented languages, specifically in the context of Maude [3, 4], 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 [14, 13] organizes term rewriting modulo equations as a complete logic and serves as a foundation for programming language semantics [17, 18].

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: $l \rightarrow r$. 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

FMOODS'07, LNCS 4468, pp 107-121, 2007

process terminates, with the final term being the result of the computation. Rewriting, like other methods of computation, can continue forever.

There exist a plethora of term rewriting engines, including ASF [21], Elan [1], Maude [3, 4], OBJ [8], Stratego [22], 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 [3, 4], 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 identity elements, 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 [9]. KOOL includes support for standard imperative features, such as assignment, conditionals, and loops 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;

FMOODS'07, LNCS 4468, pp 107-121, 2007

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: | spawn E ; | acquire E ; | release E ; | 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

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. 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). 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. 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

FMOODS'07, LNCS 4468, pp 107-121, 2007

To get a feel for the language, a sample program is shown 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`).

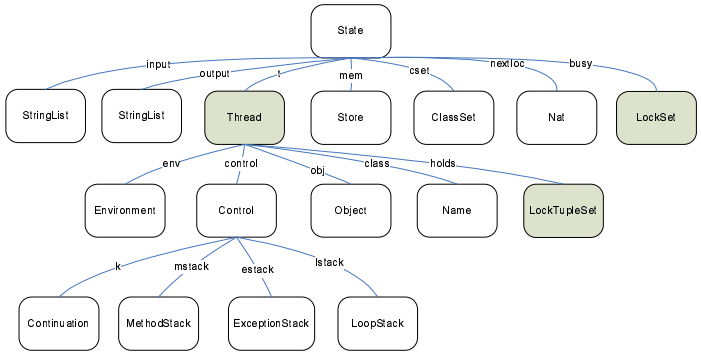

Fig. 3. KOOL State Infrastructure

3.2 KOOL Semantics

The semantics of KOOL is defined using Maude equations and rules, with the current program state 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 3; state components needed specifically for concurrency are shaded.

Figure 4 shows examples of the equations and rules which make up the KOOL semantics. Lists of computations, called continuations, are formed using the `->` operator, with the head of the list to the left. The first three equations (represented with `eq`) process a conditional. The first indicates the value of the guard expression `E` must be computed before a branch is selected. The guard is put at the left end of the list, where it will be computed by rules specific to the type of expression, while the branches `S` and `S'` are saved for later use by putting them into an `if` continuation item. The

```
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. 4. Sample KOOL Rules

FMOODS'07, LNCS 4468, pp 107-121, 2007

second and third equations execute the appropriate branch based on whether the guard evaluated to `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` (`:=` binds `V` to the result of reducing `Mem[L]`, the memory lookup operation), 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, while `K` represents the rest of the computation in this thread. Note that, since the fourth rule represents a side-effect, it can only be applied when it is the next computation step in the thread (it is at the head of the continuation), while the first three, which don't involve side-effects, can be applied at any time.

3.3 KOOL Implementation

There is an implementation of KOOL available at our website [11], as well as a web-based interface to run and analyze KOOL programs such as those presented here. There is also a companion technical report [10] that explains the syntax and semantics in detail. When run, a KOOL program is first parsed using SDF [21], which we have found better supports the complexities of real programming-language syntax than the parser included with Maude. SDF generates an abstract syntax tree that is then turned into Maude "prefix" form using a custom C program. The `runkool` program coordinates this process and handles the invocation of Maude, running in different modes (execution, search, etc.) based on command-line parameters and returning the program output.

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 [19], it is possible to output any natural number ≥ 1. In KOOL, spawn is used to execute an arbitrary expression, often a message send, in a new thread. Threads are the main unit of concurrency in KOOL, with each thread containing its own execution context (current class, environment, etc), and all threads accessing a shared store. A KOOL version of the thread game is shown in Figure 5.

```
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

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

FMOODS'07, LNCS 4468, pp 107-121, 2007

inefficient. Model checking will not help here either, since this 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 presented in Figure 6.

```
./runkool 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 examples/Spawn7.kool -s
... term and some stats omitted ...
Solution 1 (state 1964)
SL:[StringList] --> "100"

Solution 2 (state 2430)
SL:[StringList] --> "200"

Solution 3 (state 2490)
SL:[StringList] --> "AssertException thrown: Assertion triggered"
```

Fig. 7. Assertions and Search in KOOL

FMOODS'07, LNCS 4468, pp 107-121, 2007


```
class Fork is
end

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

      eating:
      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 in 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 using `acquire`. 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 using `release`, 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.

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 "`[]`$\backsim$`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 by looking into the computation directly, meaning we would need to base the property on the definition of lock acquisition, and formulate this in terms of acquiring a pair of locks. 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 possible solution in this case is to use Maude's search capabilities, described in Section 4, but this is not

FMOODS'07, LNCS 4468, pp 107-121, 2007

a general solution, since other properties of interest (starvation, for instance) cannot be checked in this way.

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 [12], 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:
  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 12. 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 due to resource exhaustion. This will be addressed in Section 6, where modifications to the design to improve verification performance will be explored. With these modifications in place, the model checker will return `true` given the LTL formula for deadlock freedom shown above.

FMOODS'07, LNCS 4468, pp 107-121, 2007

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, both of which have appeared in various forms in programming languages but not, to our knowledge, in rewriting logic language specifications. 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. Although mainly useful in dynamic languages like KOOL, this technique can also be used to perform automatic coercions between scalar and object types in statically-typed languages. 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

FMOODS'07, LNCS 4468, pp 107-121, 2007

operations, are left un-boxed, in scalar form. This all happens behind the scenes, allowing KOOL programs to remain unchanged.

```
  eq k(exp(f(F)) -> K) = k(newPrimFloat(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)) -> boxWList(Vl) -> toInvoke(Xm) -> K) [owise] .
```

Fig. 10. Example Definition Changes, Auto-boxing

An example of the rule changes to enable auto-boxing is found in Figure 10. The first equation is without auto-boxing. Here, when a floating point number `F` is encountered, a new floating point object of class `Float` is created to represent `F` using `newPrimFloat`. Any operations on this object, such as adding two floats, will involve a message send. The next three rules are with auto-boxing enabled. In the second equation, instead of creating a new object for `F`, we return a scalar value. The third equation shows an example of an intercepted method call. When a method is called, the target and all arguments are evaluated, with the method name held in the `toInvoke` continuation item. Here, `+` has been invoked with a target and argument that both evaluate to scalar float values, so we will use the built-in float `+` operation instead of requiring a method call. In the fourth equation, the boxing step is shown – here, a method outside of those handled directly on scalars has been called with the floating-point scalar value 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). Once created, the new object, and the values being sent as arguments (held in `boxWList`), will be used to perform a standard method call.

Auto-boxing has a significant impact on performance. Figure 12 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.

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 interactions 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

FMOODS'07, LNCS 4468, pp 107-121, 2007

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. Correctly 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 to move locations to the shared pool 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, including those for instance variables and those 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, not arbitrary expressions, should be spawned, 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 parameters themselves, which are local to the method, meaning that no verification performance penalty is paid until the code needs to "look inside" the referenced objects. Looking inside does not include retrieving a referenced object for use in a lock acquisition statement (however, acquisition itself is a rule).

Figure 11 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

FMOODS'07, LNCS 4468, pp 107-121, 2007


```
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. 11. 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 `L` and its value are in the unshared pool, and are moved to the shared pool. If the value is an object, all locations it holds references to are also added to the list of locations that must be processed. The second represents the case where the location is already in the shared pool. In this case, nothing is done with the location. The third equation applies only when all locations have been processed, indicating we should continue with the computation (with `K`).

This strategy could be improved with additional bookkeeping. For instance, no information on which threads share which locations is currently tracked. Tracking this information could potentially allow a finer-grained sharing mechanism, and could also allow memory to be un-shared when threads terminate. However, even with the current strategy, we still see some significant improvements in verification performance. These can be seen in Figure 12. 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.

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

FMOODS'07, LNCS 4468, pp 107-121, 2007

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. 12. Dining Philosophers Verification Times

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. There has also been some work, in the context of real-time systems, on using different state representations at different points in evaluation to improve analysis performance [20]; it would be interesting to see if similar techniques could be used in language definitions, where the lack of time steps would make it more challenging to determine when the state could be reconfigured. Finally, a method to determine that specification transformations are *semantics preserving* would be valuable, especially if it could be done automatically using the language specifications.

Acknowledgments. We thank the anonymous reviewers for their helpful comments, which have improved the quality of this paper. We also thank the NSF for their support through grants NSF CCF-0448501 and NSF CNS-0509321.

References

1. P. Borovanský, C. Kirchner, H. Kirchner, P.-E. Moreau, and C. Ringeissen. An overview of ELAN. *ENTCS*, 15, 1998.

FMOODS'07, LNCS 4468, pp 107-121, 2007

2. F. Chalub and C. Braga. A Modular Rewriting Semantics for CML. In *Proceedings of the 8th. Brazilian Symposium on Programming Languages*, May 2004.
3. 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.
4. 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.
5. 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.
6. 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.
7. 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.
8. 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.
9. A. Goldberg and D. Robson. *Smalltalk-80: the language and its implementation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983.
10. 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.
11. M. Hills and G. Rosu. KOOL Language Homepage. http://fsl.cs.uiuc.edu/KOOL.
12. G. J. Holzmann. The Model Checker SPIN. *IEEE Trans. Softw. Eng.*, 23(5):279–295, 1997.
13. N. Martí-Oliet and J. Meseguer. Rewriting logic: roadmap and bibliography. *Theoretical Computer Science*, 285:121–154, 2002.
14. J. Meseguer. Conditional Rewriting Logic as a Unified Model of Concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.
15. 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.
16. J. Meseguer, M. Palomino, and N. Martí-Oliet. Equational Abstractions. in Proc. CADE-19, Springer LNCS, Vol. 2741, 2–16, 2003.
17. 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.
18. J. Meseguer and G. Roşu. The rewriting logic semantics project. *Theoretical Computer Science*, to appear, 2006.
19. J. S. Moore. http://www.cs.utexas.edu/users/moore/publications/thread-game.html.
20. D. E. Rodríguez. On Modelling Sensor Networks in Maude. In *Proceedings of WRLA'06*. Elsevier, 2006. To appear.
21. 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.
22. E. Visser. Program Transf. with Stratego/XT: Rules, Strategies, Tools, and Systems. In *Domain-Specific Program Generation*, pages 216–238, 2003.