RTA'07, LNCS 4533, pp 246-256, 2007

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

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. This paper presents KOOL, a concurrent, dynamic, object-oriented language defined in rewriting logic. KOOL has been designed as an experimental language, with a focus on making the language easy to extend. This is done by taking advantage of the flexibility provided by rewriting logic, which allows for the rapid prototyping of new language features. An example of this process is illustrated by sketching the addition of synchronized methods. KOOL also provides support for program analysis through language extensions and the underlying capabilities of rewriting logic. This support is illustrated with several examples.

Key words: object-oriented languages, programming language semantics, term rewriting, rewriting logic, formal analysis

1 Introduction

Language design is both an art and a science. Along with formal tools and notations to address the science, it is important to have good support for the "art": tools that allow the rapid prototyping of language features, allowing new features to be quickly developed, tested, and refined (or discarded). Rewriting logic, briefly introduced in Section 2, provides a good environment for addressing both the art and the science: formal tools for specifying language semantics and analyzing programs, plus a flexible environment for adding new features that automatically provides language interpreters. As an example of the capabilities of this model of language design, we have created KOOL, a concurrent, dynamic, object-oriented language built using rewriting logic. The KOOL language and environment are described in 3; since KOOL is focused on language experimentation, Section 4 illustrates this by sketching the addition of `synchronized` methods, a concurrency-related feature borrowed from the Java language [7]. KOOL also inherits support for analysis from rewriting logic, which has been enhanced with language constructs and runtime support. This is explored in Section 5. Section 6 concludes, summarizing and discussing some related work.

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

RTA'07, LNCS 4533, pp 246-256, 2007

A sister paper [10] presents more detailed information about formal analysis of KOOL programs, including a detailed look at how design decisions in rewriting logic definitions of object oriented languages impact analysis performance.

2 Rewriting Logic

Rewriting logic [12–15] 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 using it to "execute" a term using standard term rewriting techniques.

3 KOOL

KOOL is a concurrent, dynamic, object-oriented language, loosely inspired by, but not identical to, the Smalltalk language [6]. 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; 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, etc). Most message sends are specified in a Java-like syntax; those representing

RTA'07, LNCS 4533, pp 246-256, 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	$E ::=$	$X \mid I \mid F \mid B \mid Ch \mid Str \mid (E)$ | new X | new X ($\{E,\}^+$) | self | $E\ X_{op}\ E'$ | $E.X(())^?$ | $E.X(\{E,\}^+)$ | super() | super.$X(())^?$ | super.$X(\{E,\}^+)$ | super($\{E,\}^+$) | primInvoke($\{E,\}^+$)
Statement	$S ::=$	E <- E'; | begin $D^*\ S$ end | if E then S (else $S')^?$ fi | try S catch $X\ S$ end | throw E ; | while E do S od | for X <- E to E' do S od | break; | continue; | 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

binary operations can also be used infix (`a + b` desugars to `a.+(b)`), with these infix usages all having 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`.

To get a feel for the language, two sample class definitions are presented in Figure 2. These definitions provide a simple example of inheritance and calls to `super`-methods using a familiar `Point`/`ColorPoint` example. Class `Point` represents a point in 2D space, with x and y coordinates, and implicitly inherits from (extends) class `Object`. Class `ColorPoint` explicitly extends class `Point`, adding a new variable `c` to represent the color of the point. The constructor for `ColorPoint` calls its parent constructor, passing the x and y coordinates, while the version of `toString` defined in `ColorPoint` also uses `super`, here to invoke the parent version of the `toString` method. `+` is defined in the `String` class as string concatenation.

```
class Point is
  var x,y;

  method Point(inx, iny) is
    x <- inx; y <- iny;
  end

  method toString is
    return ("x = " + x.toString() + " and y = "
                   + y.toString());
  end
end

class ColorPoint extends Point is
  var c;

  method ColorPoint(inx, iny, inc) is
    super(inx,iny); c <- inc;
  end

  method toString is
    return (super.toString() + " and c = "
                             + c.toString());
  end
end
```

Fig. 2. Inheritance and Built-ins in KOOL

RTA'07, LNCS 4533, pp 246-256, 2007

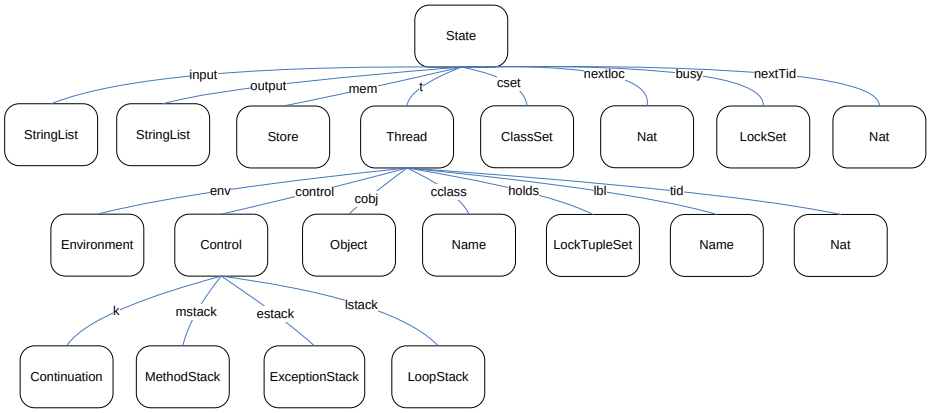

Fig. 3. KOOL State Infrastructure

3.2 KOOL Semantics

The semantics of KOOL is defined using Maude [2, 3], a high-performance language and engine for rewriting logic. The current program is represented as a "soup" (multiset) of nested terms representing the current computation, memory, locks held, etc. A visual representation of this term, the state infrastructure, is shown in Figure 3. Here, each box represents a piece of information stored in the program state, with the text in each box indicating the information's *sort*, such as `Name` or `LockSet`. Edges between boxes represent the names used to reference the information, such as `cclass` (the current class context) or `busy` (a set of all locks held by any thread in the program), and are defined as operations from the boxed to the containing sort (i.e., `cclass` is an operation from `Name` to `Thread`, so it can be treated as thread information). Information stored in the state can be nested: `State` contains at least one `Thread`, accessed with `t`, which contains `Control` information, accessed with `control`. The most important piece of information is the `Continuation`, located in `Control` and named `k`, which is a first-order representation of the current computation, made up of a list of instructions separated by `->`. The continuation can be seen as a stack, with the current instruction at the left and the remainder (continuation) of the computation to the right. This *continuation-based* methodology is described in more detail in papers about the rewriting logic semantics project [14, 15].

Figure 4 shows examples of Maude equations and rules included in the KOOL semantics. The first three equations (shown with `eq`) process a conditional. The first indicates the value of the guard expression `E` must be computed before a branch state-

```
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

RTA'07, LNCS 4533, pp 246-256, 2007

ment (`S` or `S'`) is evaluated; to do this, `E` is put before the branches on the continuation, with the branches saved for later use by putting them into an `if` continuation item. The second and third 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` and `V` are actually in `Mem`), the result of the computation is the value `V`. This must be a rule, since memory reads and writes among different threads could lead to non-equivalent behaviors. `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. The website also contains current information about the language, which is constantly evolving. A companion technical report [9] explains the syntax and semantics of KOOL in more detail.

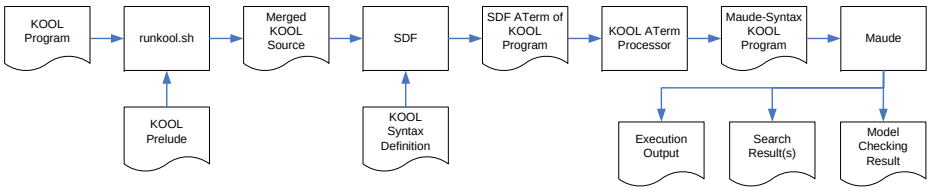

Fig. 5. KOOL Program Evaluation/Analysis

KOOL programs are generally run using the `runkool` script, since running a KOOL program involves several steps and tools, and since programs can be run in different modes (execution, search, and model checking, each with various options). First, the KOOL prelude, a shared set of classes for use in user programs, is added to the input program. This program is then parsed using the SDF parser [19], which takes the program text and a syntax definition file as input. The parser produces a file in ATerm format, an SDF format used to represent the abstract syntax tree. A pretty printer then converts this into Maude, using prefix versions of the operators to prevent parsing difficulties. Finally, Maude is invoked by `runkool` with the language semantics and the Maude-format program, generating the result based on the execution mode. A graphical view of this process is presented in Figure 5.

RTA'07, LNCS 4533, pp 246-256, 2007

4 Extending KOOL

Because of its use in our language research and in teaching, the KOOL system has been designed to be extensible. To illustrate the extension process at a high level, an example extension, `synchronized` methods, is presented here. Further details, including a full implementation, are available at the KOOL website [11].

The KOOL language has a fairly simple model of concurrency based on threads, which each contain their own continuation and execution context (shown in Figure 3). Each program starts with one thread. The `spawn` statement can then be used to create more threads, which execute independently of one another. All mutual exclusion is handled with object-level locks, acquired using `acquire` and released with `release`. Objects can only be locked by one thread at a time, although that thread may lock the same object more than once. If this happens, the thread needs to release an equivalent number of locks on the object before it can be `acquire`'d by another thread. `synchronized` methods, similar to those in Java, would provide a higher-level abstraction over these locking primitives. In Java, methods tagged with the `synchronized` keyword implicitly lock the object that is the target of the method call, allowing only one thread to be active in all `synchronized` methods on a given object at a time. We will assume the same semantics for KOOL.

The syntax changes to add `synchronized` methods are minor: the keyword needs to be added to the method syntax, which then also needs to be reflected in the Maude-level syntax for KOOL. In SDF, two context-free definitions for Method nonterminals need to be added, one for `synchronized` methods with parameters, the other for those without. Similar changes need to occur in Maude, where one additional syntax operation needs to be added. Finally, the pretty printer that translates from SDF ATerms into Maude needs to be modified to handle the two new SDF definitions; this change is fairly mechanical, and methods to eliminate the need for this are being investigated.

The changes to the semantics are obviously more involved. In KOOL, as in Java, `synchronized` methods should work as follows:

- a call to a synchronized method should implicitly acquire a lock on the message target before the method body is executed;
- a return from a synchronized method should release this lock;
- additional calls to synchronized methods on the same target should be allowed in the same thread;
- exceptional returns from methods should release any locks implicitly acquired when the method was called.

A quick survey of these requirements shows that adding `synchronized` methods will change more than the message send semantics – the semantics for exceptions will need to change as well, to account for the last requirement.

To handle the first requirement, a new lock can be acquired on the `self` object at the start of any `synchronized` method simply by adding a lock acquisition to the start of the method body, which can be done when the method definition is processed. Lock release cannot be handled similarly, though, since there may

RTA'07, LNCS 4533, pp 246-256, 2007

be multiple exits from a method, including `return` statements and exceptional returns. This means that locks acquired on method entry will need to be tracked so they can be properly released on exit. This can be accomplished by recording the lock information in the method and exception stacks (`mstack` and `estack` in Figure 3) when the lock is acquired, since these stacks are accessed in the method return and exception handling semantics. With this in place, the second and fourth requirements can be handled by using this recorded information to release the locks on method return or when an exception is thrown. Finally, the third point is naturally satisfied by the existing concurrency semantics, which allow multiple locks on the same object (here, `self`) by the same thread. Overall, adding synchronized methods to the KOOL semantics requires:

- 2 modified operators (to add locks to the two stack definitions),
- 4 modified equations (two for method return, two for exception handling),
- 4 new operators (to record locks in the stacks, to release all recorded locks),
- 6 new equations (to record locks in the stacks, to release all recorded locks).

With these changes, KOOL includes 243 operators, 334 equations, and 15 rules.

5 Analyzing KOOL Programs

KOOL program analysis is based around the underlying functionality provided by Maude and rewriting logic, including the ability to perform model checking based on LTL formulae and program states and the ability to perform a breadth-first search of the program state space. These capabilities have been extended with additions to the KOOL language, including assertion checking capabilities that interact with the model checker and program labels that can be used in LTL formulae to check progress (or the lack thereof) between program points.

5.1 Breadth-First Search

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 [16], it is possible to output any natural number ≥ 1. A KOOL version of the thread game is shown in Figure 6.

```
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. 6. Thread Game

To check if a specific value can be output, one could run the program repeatedly, or try model checking. However, with the program's infinite state space and nondeterministic behavior, this may never yield the desired result. Maude's search capability can be used, though, either to enumerate

RTA'07, LNCS 4533, pp 246-256, 2007

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 output values.

Search can also be useful when adding new language features. For instance, an example of `synchronized` methods in KOOL is shown in Figure 7. Here, class `WriteNum` contains two `synchronized` methods. When the `write` method is called, the starting value of the number stored in member variable `num` is written to the console, some simple arithmetic operations are performed on it, and then the final value is written. The `set` method assigns a new value to `num`. Since both methods are marked `synchronized`, it should be the case that, for any given object, once one thread is executing either method, another thread that tries to execute either will wait. To test this, the `Driver` class creates a new object of class `WriteNum`, spawns one call to `write`, creating a new thread, modifies the value stored in the object using `set`, and then creates a second thread, also calling `write`. Using search to determine possible program outputs reveals that there are only two possible solutions, with the call to `set` either occurring before the first spawned thread runs (with output `"Start:","20","End:","22","Start:","22","End:","24"`), or after it completes (printing `"Start:","10","End:","12","Start:","20","End:","22"`). By contrast, with the `synchronized` keywords removed, there are 470 solutions, corresponding to all possible orderings of output based on various interleavings of the main thread with the two spawned threads.

```
class WriteNum is
  var num;

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

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

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

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

(new Driver).run
```

Fig. 7. Synchronized Methods

5.2 Model Checking

Programs in KOOL can take advantage of Maude's model-checking capabilities. The Maude model checker uses LTL formulae, which are written against the state infrastructure. Since the state can be very complex, KOOL allows *labels*, which can be referenced in LTL formulae, to be included in the program source. For example, Figure 8 contains a KOOL fragment of the Dining Philosophers problem. Each `Philosopher`

```
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
```

Fig. 8. Philosophers and Forks

RTA'07, LNCS 4533, pp 246-256, 2007

will try to lock left and right `Fork`s before eating, releasing them when finished. We can check for deadlock freedom by verifying that reaching label `hungry:` implies always eventually reaching label `eating:` (always eventually acquiring both locks). Results for model checking a deadlocking and a fixed version of the problem with 5, 6, and 8 philosophers are shown in Figure 9. More details about model checking and search in KOOL, including further discussion of labels, additional examples of search, additional performance measurements, and an investigation of the impact of language design on analysis performance can be found in a related paper [10].

Philosophers	# of States	Find Counterexample/s	Prove Deadlock Freedom/s
5	634	1.989	23.014
6	2943	8.444	130.174
8	63505	278.276	4975.583

3.40 GHz P4, 2 GB RAM, OpenSuSE 10.2, kernel 2.6.16.27-0.6-smp, Maude 2.2.

Fig. 9. Dining Philosophers Verification Time

6 Conclusions and Related Work

In this paper we have presented the KOOL language as a concrete application of rewriting logic to language design and program analysis, illustrating the process of language extension and highlighting the provided analysis capabilities. KOOL has been used as a basis for both teaching [17] and research in language semantics, design, analysis, and verification. The evolving nature of KOOL seems to make it especially appealing for classroom use, where student projects have included adding reflection and additional concurrency features.

There is a large volume of work related to defining programming languages using executable techniques. Probably the most related is the JavaFAN project, which defined formal analysis tools for both the Java language [4] and JVM bytecode [5]. In contrast to our work, which has focused on the design of languages and feature prototyping, JavaFAN has focused on formal methods applications, with the goal of being a competitive Java formal analysis tool. The published work on Java [4], for instance, is a 4 page tools paper that provides a short summary of the rewriting logic definition of Java, focusing instead on a description of the tool, leaving the language definition unpublished. The languages are also quite different, leading to different challenges in implementation (such as KOOL's use of Smalltalk-like primitives, here fragments of Maude, to implement low-level operations).

Functional languages defined in Maude include CML [1] and Eden [8]. Work on the latter, a parallel variant of Haskell, has also looked to Maude as a language experimentation environment, using the module system to vary the degree of parallelism and the scheduling algorithm. Still in the realm of term rewriting, the ASF+SDF project [19] has focused on language prototyping and definition. More related work can be found in papers on the rewriting logic semantics of programming languages [14, 15] and the K language definition technique [18].

RTA'07, LNCS 4533, pp 246-256, 2007

Acknowledgments. We thank the anonymous reviewers for their helpful comments, which have improved the quality of this paper.

References

1. F. Chalub and C. Braga. A Modular Rewriting Semantics for CML. In *Proceedings of the 8th. Brazilian Symposium on Programming Languages*, May 2004.
2. 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.
3. 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 of *LNCS*, pages 76–87. Springer, 2003.
4. 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.
5. 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. Springer, 2004.
6. A. Goldberg and D. Robson. *Smalltalk-80: the language and its implementation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983.
7. J. Gosling, B. Joy, and G. Steele. *The Java Language Definition*. Addison-Wesley, 1996.
8. M. Hidalgo-Herrero, A. Verdejo, and Y. Ortega-Mallén. Using Maude and its strategies for defining a framework for analyzing Eden semantics. In *Proceedings of WRS'06*. Elsevier, 2006. To appear.
9. 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.
10. M. Hills and G. Roşu. On Formal Analysis of OO Languages using Rewriting Logic: Designing for Performance. In *Proceedings of FMOODS'07*, LNCS. Springer, 2007. To appear.
11. M. Hills and G. Rosu. KOOL Language Homepage. http://fsl.cs.uiuc.edu/KOOL.
12. N. Martí-Oliet and J. Meseguer. Rewriting logic: roadmap and bibliography. *Theoretical Computer Science*, 285:121–154, 2002.
13. J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.
14. J. Meseguer and G. Roşu. Rewriting Logic Semantics: From Language Specifications to Formal Analysis Tools . In *Proceedings of IJCAR'04*, volume 3097 of *LNAI*, pages 1–44. Springer, 2004.
15. J. Meseguer and G. Roşu. The rewriting logic semantics project. *Theoretical Computer Science*, to appear, 2007.
16. J. S. Moore. http://www.cs.utexas.edu/users/moore/publications/thread-game.html.
17. G. Roşu. Lecture notes of course on Programming Language Design. Dept. of Computer Science, UIUC, 2006. http://fsl.cs.uiuc.edu/index.php/CS422.
18. G. Roşu. K: a Rewrite Logic Framework for Language Design, Semantics, Analysis and Implementation. Technical Report UIUCDCS-R-2006-2802, Department of Computer Science, University of Illinois at Urbana-Champaign, 2006.
19. 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.