A Rewrite Logic Approach to Semantic Definition, Design and Analysis of Object-Oriented Languages

Feng Chen  Mark Hills  Grigore Roșu
University of Illinois Urbana-Champaign
{fengchen,mhills,grosu}@cs.uiuc.edu

Abstract
This paper introduces a framework for rapid prototyping of object oriented programming languages and corresponding analysis tools. It is based on formal definitions of language features in rewrite logic, a simple and intuitive logic for concurrency with powerful tool support. A domain-specific front-end consisting of a notation and a technique, called K, allows for compact, modular, expressive and easy to understand and change definitions of language features. The framework is illustrated by first defining KOOL, an experimental concurrent object-oriented language with exceptions, and then by discussing the definition of JAVA. Generic rewrite logic tools, such as efficient rewrite engines and model checkers, can be used on language definitions and yield interpreters and corresponding formal program analyzers at no additional cost.

Categories and Subject Descriptors  D.3.1 [Programming Languages]: Formal definitions, design, theory.

Keywords  Semantics, rewriting, object-oriented languages.

1. Introduction
The elegance, expressiveness, and scalability of object-oriented (OO) programming have created a broad interest in the theoretical and practical aspects of the OO paradigm among language designers and semanticists. Many OO programming languages have been devised, including some that integrate the OO paradigm with other language paradigms (for instance, Scala [2] is a hybrid OO and functional language). Many features, such as concurrency, exceptions, and generics, have been added over time, and are now seen as standard in many OO languages. Program analysis and verification techniques have also been extended or adapted to OO languages, some of them defining implicitly or explicitly the complete semantics of the corresponding OO languages.

Even given the effort so far, the quest for improved OO languages and corresponding analysis tools and techniques is not finished. Appropriate frameworks for design and analysis of programming languages can significantly facilitate our efforts to define, understand and experiment with novel paradigms, combinations of paradigms, and new language features. But what makes a language definitional framework appropriate? We believe that an ideal such framework should satisfy some core requirements – it should be (1) generic, that is, not tied to any particular OO programming language or paradigm. For example, a framework enforcing static typing of programs in the defined language may be inconvenient for defining dynamically typed OO languages, while a framework enforcing object communication via explicit send and receive messages may require artificial encodings of OO languages that opt for a different communication approach.

(2) semantics-based, based on formal definitions of languages rather than on ad-hoc implementations of interpreters and/or analysis tools. Semantics is crucial to such an ideal definitional framework because, without a formal semantics of a language, the problems of program analysis and interpreter or compiler correctness are meaningless. One could use ad-hoc implementations to find errors in programs or compilers, but never to prove their correctness. (3) able to naturally support concurrency. Due to the strong trend in parallelizing architectures for increased performance, future programming languages are expected to be concurrent. In particular, the OO paradigm and concurrency fit very well together. To properly define and reason about concurrent languages, the underlying definitional framework should be inherently concurrent, rather than artificially graft concurrency on an essentially sequential paradigm, for example, by defining or simulating a process/thread scheduler. (4) executable. Having the possibility to execute programs using the semantic definition of a language gives one confidence in the correctness of that definition. In our experience, executing hundreds of programs exercising various features of a language helps not only find and fix errors in that language definition, but also stimulates the desire to experiment with new features. A computational logic framework with efficient executability and a spectrum of meta-tools can serve as a basis not only to define executable formal semantics of languages, but also to develop formal analysis techniques and tools (symbolic execution, static analysis, model checkers, etc.). (5) modular, to facilitate reuse of language features. In this context, modularity of a definitional framework means the ability to add or remove language features without having to modify any definitions of other, unrelated features. For example, if one adds parametric exceptions to one’s language, then one should just include the corresponding module and change nothing else. A typical SOS definition of a language lacks modularity because one needs to “update” all the SOS rules whenever the structure of the state, or configuration, changes (like in the case of adding exceptions).

There are additional desirable, yet more subjective and thus harder to quantify, requirements of an ideal language definitional framework, including: it should be simple, easy to understand and teach; it should have good data representation capabilities; it should scale well, to apply to arbitrarily large languages.

There are subtle tensions among the requirements above, making it hard, if not impossible, to find an ideal language definitional framework. Following up recent work in rewriting logic semantics [34, 33], in this paper we argue that rewriting logic [32] can be a reasonable starting point towards the development of such a framework, at least for the definition, design and analysis of concurrent object-oriented languages. A rewrite logic theory consists of a set of uninterpreted operations constrained equationally, together with a set of rewrite rules meant to define the concurrent evolution of the defined system. The distinction between equations and rewrite rules is only semantic; they are both executed as rewrite rules \( l \rightarrow r \).

---

* Supported by NSF grants CCR-0234524, CCF-0448501, CNS-0509321.
by rewrite engines, following the simple and uniform **match-and-apply** iterative principle of term rewriting: find a subterm matching \( l \), say with a substitution \( \theta \), then replace it by \( \theta(r) \).

Rewriting logic admits an **initial model semantics**, where equations form equivalence classes on terms and rewrite rules define transitions between such equivalence classes. Operationally, rewrite rules can be applied concurrently, thus making rewriting logic a very simple, generic and universal framework for concurrency; indeed, many other theoretical frameworks for concurrency, including \( \pi \)-calculus, process algebra, actors, etc., have been seamlessly defined in rewriting logic [31]. In our context of programming languages, a programming language definition is a rewrite logic theory in which equations define the non-concurrent features of the language, while rewrite rules define the concurrent features. A program together with its initial state are given as an uninterpreted term, whose denotation in the designated model is its corresponding transition system. Depending on the type of analysis one is interested in, one can, using existing tool support, generate anywhere from one path in that transition system (e.g., when "executing" the program) to all paths (e.g., for model checking).

One must, nevertheless, treat the simplicity and generality of rewriting logic with caution; "general" and "universal" need not necessarily mean "better" or "easier to use", for the same reason that machine code is not better or easier to use than higher level programming languages that translate it. In our context of defining concurrent object-oriented programming languages in rewriting logic, the right questions to ask are whether rewriting logic provides a natural framework for this task or not, and whether we get any benefit from using it. In this paper we attempt to empirically answer yes to these questions. In spite of its simplicity and generality, rewriting logic does not give us any immediate recipe for how to define languages as rewrite logic theories. Appropriate **definitional techniques and methodologies** are necessary in order to make rewriting logic an effective computational framework for programming language definition.

In [42] we introduced K, a programming language domain-specific front-end to rewriting logic, that allows for compact, modular, expressive and easy to understand and change definitions of language features. K has been used in [42] to define FUN, a concurrent functional language with exceptions and call/cc, and in [26] to define SILF, a simple C-like imperative language with functions. We here discuss the definition of two OO languages, KOOL and JAVA. KOOL, a K-defined object-oriented language, is dynamically typed, has parametric exceptions, typecase statements, and admits creation, termination and synchronization of arbitrarily many execution threads; unlike in the \( K \) definition of JAVA, in KOOL we have chosen to allow a statement **spawn**(E) for spawning a thread to calculate any expression; therefore, concurrency in KOOL is regarded as a top-level core feature of the language, versus hooking into the OO inheritance and message send semantics. Also, to show the flexibility of our definitional framework, following the ideas presented in [21], we show the addition of **inner calls** to our \( K \) definition of JAVA. The important point of this is not to highlight specific features of languages, but instead to highlight that a language design framework should be flexible enough to support a rapid, yet formal, investigation of language features, lowering the boundary between new ideas and executable systems for trying these ideas.

As an example of a \( K \) definition, Figure 1 depicts **all** (three) \( K \) rules for parametric exceptions in KOOL; the second line contains two rules, separated by a vertical bar. Rules in \( K \) are **contextual**: each rule consists of a context with multiple "holes" in which it applies, each hole being marked by a horizontal line underlying the corresponding subterm; the terms below the lines replace in parallel the subterms above the lines. Thus, each \( K \) rule can be regarded as a rewrite rule of the form \( C[t_1, \ldots, t_n] \rightarrow C[t'_1, \ldots, t'_n] \), where \( C \) is the corresponding context, \( t_1, \ldots, t_n \) are the terms above the horizontal lines, and \( t'_1, \ldots, t'_n \) are the terms below the horizontal lines. Variables are capital letters or, when not necessary, underscores. Rules can apply at any position in a term (not only at its top) and can apply concurrently. Equations allow us to use matching module associativity, commutativity and identity of some operations; for example, the state of KOOL is organized as a "soup" multiset of state attributes, each having nested subsoups; e.g., \( k \) wraps the remaining part of the computation, \( estack \) the exception stack, and \( env, obj \) and \( class \) the current environment, object and class.

The \( K \) definitional technique is based on a first-order representation of continuations [40], in our case lists (built with an associative operation) of tasks separated by \( \rightarrow \). The "\( \rightarrow \)" is the identity of list and set constructors, and the \( \rightarrow \) ending the soup term wrapped by \( estack \) says that we are not interested in the rest of the exception stack (the "\( \rightarrow \)" can be read as "and continues to the right"). The first rule says that if a \( try \) \( catch \) \( S \) \( S' \) \( end \) statement is the next task of some thread, then change the state as follows: (1) schedule \( S \) for execution followed by a marker to pop the exception stack; and (2) stack the current control, environment, object, class, as well as the continuation binding \( X \) then \( S' \) then switch environment back to \( Env \) then \( K \); this continuation will be executed in case an exception is thrown during \( S \) (with the third rule in Figure 1). The second rule simply pops the stack and discards the marker in case \( S \) evaluates normally. The third rule "goes back in time" at the \( try/catch \) time in case an exception is thrown, pops the stack and passes the value thrown to the previously created continuation.

The \( K \) notation and technique will be discussed more fully in Section 2. We here only want to emphasize why we believe that once learned, \( K \) is simple, natural, and leads to compact and easy to understand and read language definitions. \( K \) could be explained and presented orthogonally to rewriting logic, as a standalone language definitional framework, but we prefer to regard it as a language-specific front-end to rewriting logic; however, as discussed in [34, 5], SOS [39], MSOS [37] and reduction semantics [14] can also be seamlessly translated into rewriting logic.

\( K \) can be all resolved **statically**, so a definition of a language in \( K \) is a rewrite logic specification. Since rewriting logic is efficiently executable, and since some rewrite engines are very fast, efficient interpreters are obtained for free from such formal language definitions. Moreover, formal analysis tools for rewrite logic specifications, such as those of Maude [10] (e.g., model checkers), translate into corresponding tools for languages defined using the presented technique. Also, since rewriting logic is a computational logical framework with both initial model semantics and formal

### Figure 1. Exception handling K-rules in KOOL.
\[
\frac{k(\text{lookup}(L)) \text{ mem}(\{L, V\})}{V}
\]
(3)

\[
\frac{k(V \sim \text{assign}(L)) \text{ mem}(\{L, V\})}{V}
\]
(4)

\[
\text{parent}(C, \langle \text{cls}(\text{name}(C)) \text{name}(C') \rangle) \quad C'
\]
(5)

\[
\text{flds}(C, \langle \text{cls}(\text{name}(C)) \text{ flds}(X1) \rangle) \quad X1
\]
(6)

\[
\text{getInheritsSet}(\text{Object}, \text{CSet} \rightarrow) \quad \text{CSet} \text{ Object}
\]
(7)

\[
\text{getInheritsSet}(\langle C, \langle \text{CSet} \text{ } C \rangle \text{ Set} \rangle) \quad \text{CSet} \text{ CSet}
\]
(8)

\[
\text{getMthd}(X, C, \langle \text{cls}(\text{name}(C)) \text{ mthd}(\text{name}(X) \text{ MthdItms}) \rangle) \quad (8)
\]
(9)

\[
\text{getMthd}(X, C, \langle \text{parent}(C, \text{CSet}) \rangle) \quad \text{CSet}
\]
(10)

\text{parent} : \text{Name} \times \text{ClassSet} \rightarrow \text{Name} \quad \text{[Memo]}
\text{flds} : \text{Name} \times \text{ClassSet} \rightarrow \text{NameList} \quad \text{[Memo]}
\text{getInheritsSet} : \text{Name} \times \text{ClassSet} \times \text{ClassSet} \rightarrow \text{ContinuationItem} \quad \text{[Memo]}
\text{getMthd} : \text{Name} \times \text{Name} \times \text{ClassSet} \rightarrow \text{ContinuationItem} \quad \text{[Memo]}

**Figure 2.** K definitions of common operators

Since rewriting logic and its formal analysis capabilities have only relatively recently been considered as a semantic and operational framework for programming languages, we include a discussion of rewriting logic, and its important sublogic, equational logic, in Appendix A. We also recall the K notation in Section 2.

Section 3 discusses the K definition of the simple OO language KOOL, including some design aspects and a concurrent extension to the language. Section 4 then discusses the definition of an existing OO language in K, JAVA, together with uses of such a formal definition: formal analysis (e.g., model checking) at no additional definitional expense and flexible experimentation (e.g., adding inner calls). Finally, Section 6 concludes the paper, discusses some relevant related work, and proposes further developments.

2. **The K Framework**

The programming language definitional framework K [42] builds on two important observations: (1) in most programming language definitions, the structure of the state does not change significantly; and (2) many programming languages have non-trivial control statements, such as exceptions and thread switching, so a continuation-based [40] style making the execution control of a program explicit is desirable. Regarding (1), in many of our language definitions using plain rewriting logic, e.g., [34, 33] (and also [41]), terms to rewrite tended to become quite large and therefore hard to read. For example, Appendix B shows a K rule and its corresponding rewrite rule as it appears in Maude. Moreover, since most rewrites take place only under certain contexts, the left-hand-side and the right-hand-side terms of many rewriting rules were almost identical. Additionally, since only small parts of such contexts had to change, to state the part that does not change one had to invent many new variables for this sole purpose.

Regarding (2), it soon became obvious to us that continuation-based definitions can be not only orders of magnitude faster when executed, but also easier to read and understand when continuations are given a first order representation, as plain lists of continuation items, or tasks, in our algebraic framework. We next show some simple examples of K definitions, mentioning here only that all K rules can be resolved statically into ordinary equations or rewrite rules, and referring the interested reader to [42] for the algorithmic and technical details.

Any K definition should start by defining the state structural operators, that is, a subsignature used to construct those parts of the state of a particular language whose corresponding (paths to their) positions in the state do not change. These operations can be depicted graphically for clarity. For example, those of KOOL are shown in Figure 5, where nodes are sorts and edges are (uninterpreted) operations. One should use different names for the structural operations. We interchangeably may call the structured operations the state infrastructure. The sorts appearing in the state infrastructure are associated either with list or multiset operations; in other words, the constructor operations for terms having these sorts are defined to be either associative or commutative. Appendix A shows the details of how one can use equations to do this in rewriting logic; for now, one can simply think of these sorts as either lists (including stacks) or multi-sets. Unless specified otherwise, we use the infix comma “,” operation as a constructor for lists and the concatenation “,” operation for sets. An important exception for the comma convention is the use of the operation “,” as an associative constructor for continuations.

Therefore, the path to any structural subterm, called also state component or state attribute, of the state can be described as a sequence of named structural operations. For example, the exception stack of a thread can be identified as \(t; \text{control; estack}\). Moreover, since structural operations have different names, one can unambiguously identify a component by just mentioning the name of its corresponding attribute. Therefore, in K we avoid mentioning all the path to state components. This is not only convenient, but has a crucial role in the modularity of language definitions, because the structure of the state can change from language to language. As mentioned in the introduction, rules in K are contextual, where their application context is given by underlining subterms and writing other terms underneath the lines; the terms below the lines replace in parallel their corresponding underlined subterms. To avoid mentioning unnecessary elements in lists or sets, K provides the angle brackets “(” and “)” to signify that there may be other elements “to the left” and/or “to the right” of the enclosed subterm.

Figure 2 shows some K definitions that are common to most OO languages, the first two being common to all the programming languages that we defined so far in K. The first contextual rule, Rule (3), defines how the value \(V\) corresponding to a location \(L\) is retrieved from memory, when the lookup operation is the next task on the continuation. Note the “)” angle bracket to the right of the continuation, saying that the rest of the continuation does not matter here, and the “(” and “)” brackets used to “extract” the pair \((L, V)\) from the store, saying that it does not matter what other pairs are in the store; in our language definitions the store is defined as a multiset, so one can just as well use only one angle bracket. Once the value is found, the lookup operation is replaced by the expected value, which is hereby passed to the rest of the continuation.

Rule (4) has a two-hole context, one identifying the value-to-location-assign task on top of the continuation and the other iden-
ifying the pair corresponding to the location in the store; once matched, the assign task is eliminated ("=" is the identity of list and set operators, i.e., nil, empty, etc.) and the current value at that location is replaced by the assigned value (underscores are variables that are needed in the rule only for structural reasons; their corresponding subterms are not important). Some K-rules are boxed, meaning that they correspond to rewrite rules in the associated rewrite logic theory, so their application may split the state space in the associated transition system (see A); the unboxed K-rules correspond to equations. This distinction between rewrite rules and equations is not as important in the dynamic semantics, but as discussed in A is crucial for the formal analysis of concurrent programs. The remaining K-rules in Figure 2 define several operators typical in OO programming language definitions, such as ones for locating the parent class, the fields or a particular method of a class, or the set of names of classes inherited by a class; the syntax of these operators is defined at the bottom of Figure 2.

Rules (5) and (6) are self explanatory; each class' information is "wrapped" with the constructor cls, and all classes are kept as a set in the structure referred to with the state attribute cset. Let us discuss the definition of getInheritsSet which calculates the set of classes inherited by a given class. In Rule (7) the root of the inheritance tree has been reached, so Object is added to the set and the set replaces getInheritsSet on the continuation. In Rule (8), class name C is added to the set and C is replaced by C's parent. The definition of getMthd is straightforward. Notice that all four operation declarations in Figure 2 are memoized, so the results will be saved if they are needed again. This has the effect of calculating these operations in a "call-by-need" style, providing a performance boost in the execution without requiring explicit optimization steps to be taken, such as flattening class definitions to bring in all visible methods (essentially building a vtable). Many rewrite engines support memoization.

An natural question regarding our definitions of languages in rewriting logic in general and in K in particular, is why would one want to use an inherently intractable procedure, namely that of matching modulo associativity and commutativity (AC-matching), as a core operation of a language semantics. Our experience so far, backed by empirical results, is, however, that this complexity is not a factor of practical concern. That is because the intractability of AC-matching follows from quite artificial encodings of known hard problems as AC-matching problems, while our use of AC matching in programming language definitions is quite restricted if one follows the guidelines of good K definitions: even though K can be in the worst case as complex as unrestricted rewriting, in practice it uses an efficiently executable fragment of it. An automated procedure to compile K language definitions into efficient interpreters has been outlined in [26] and applied manually on SILF with promising results: the obtained interpreter was between one and two orders of magnitude slower than C and, one some sample programs, faster than JAVA. We here do not discuss implementation aspects of K to obtain fast language interpreters; instead, we focus on its use in the context of designing and analyzing concurrent object-oriented languages.

All the notational conventions in K can be resolved statically; therefore, from a theoretical perspective, a definition of a language in K is nothing but a syntactically sugared rewrite logic theory.

3. KOOL: A Simple Object-Oriented Language

We here define KOOL, a simple object-oriented language similar in spirit to the SMALLTALK language [20, 7]. KOOL has several core features, familiar from other object-oriented languages: types are dynamic; 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 is not defined with concurrency features in this section, but is extended to support concurrency in Section 3.3.

KOOL includes support for standard imperative features, such as assignment, conditionals, and loops with break and continue, as well as features found in many OO languages such as exceptions and run-time type inspection of objects via a typecase construct. 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. 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(). The syntax of KOOL is shown in Figure 4. 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. Finally, semicolons are used as statement terminators, not separators, and are only needed where the end of a statement is ambiguous (at the end of an assignment, for instance, but not at the end of a conditional, which has a keyword to designate its end).

To get a feel for the language, a sample program is presented in Figure 3. This program provides a simple example of inheritance and calls to super-methods. Note that here + is string concatenation and << is the output operator for console (which represents standard output). The << method sends its parameter the toString message to get the proper string to output.

There is an initial implementation of KOOL available at our website [1]. Programs are parsed using SDF [44] and then executed using Maude [8, 9]. The core of the language is finished (including all semantics discussed here), and we are currently adding addi-

![Figure 3. Inheritance and Built-ins in KOOL](image-url)
tional functionality to the prelude (which includes classes such as `Object`, `Integer`, `String`, and `Console`) as well as using KOOL as a basis for further research in semantics and OO languages.

### 3.1 State Infrastructure

One of the key design decisions for a language making use of K is the structure of the state. The K rules make use of this structure to determine the contexts within which the rules are applied, including matching over sets of terms and gathering like elements together into a single subterm that can be manipulated as a whole. It is important then to ensure that all needed information is available and organized into appropriate groups and that the structure is extensible, allowing changes to the semantics that require additions to the infrastructure without breaking existing rules in the semantics.

The KOOL state is broken into several distinct pieces, and uses a single explicit layer of nesting to group like components together. A visual depiction of the state is shown in Figure 5. The parts of the state that are grey-filled are not part of the sequential KOOL state, but are instead added to support concurrency, and can be safely ignored for now (currently, Control is directly under State). The concurrent extension to the language is discussed in Section 3.3.

During program execution, we keep track of names that are in scope and their current memory locations. This is stored in env. These memory locations then map to values in mem, with the next free memory location in nextLoc. We assume garbage collection in KOOL, but do not define it here. Input and output are stored in the input and output state components, respectively.

Those state components related directly to execution control are stored in control. This includes several stacks that are used to quickly recover the program to a state saved at a prior point in time: the method stack (mstack), exception stack (estack), and loop stack (iStack). While not strictly necessary, they save the effort of having to selectively unwind the control context to get back to the proper context for handling a method return or exception catch, for instance. Also included is the current continuation, or k, which provides an explicit representation of the current stream of execution and also gives its name to our definitional approach. Finally, we have several components needed just for the object-oriented features of the language. These include the current object (obj) and current class (class), which model the object-related portion of the execution context, and the class set (cset), which contains information on all classes that have been defined.

### 3.2 Dynamic Semantics

The dynamic semantics for KOOL are herein defined using K. As with any non-trivial language, there are actually a fair number of K-rules (about 300, including primitive operations) needed to give the semantics of the language. To allow for a fuller explication of the language features in our framework, we have selected several areas that are most illustrative of our technique and are most interesting from an object-oriented perspective. Full details on the language can be found in the companion technical report [25].

The semantics for each area of functionality are separated into individual figures. Of the operators that are used, most are left undefined, since the definition can be derived easily from the context in which the operator is used. For instance, an operator `op(X)` takes a name as a parameter and, if it is on the continuation, is a continuation item. Thus, it has signature `op : Name -> ContinuationItem`.

There are two exceptions to this. First, operators are defined for all syntactic constructs in the language in the figure in which they are used. This helps make the leap from the syntax of the language to the semantics. Second, operators are defined if they have attributes, since there would be no other way to know that they have the attributes they have been given. The K attributes in the rules below include `?` and `Memo`.

When possible, in the rules that follow we make heavy use of matching across contexts. This generally keeps the rules shorter and allows us to focus only on the important elements of the rule and context, without needing to navigate explicitly across intervening parts of the state. Also, rules are over a slightly more abstract version of the syntax, the main differences being that all message sends are transformed into dot notation with explicit (even if empty) parameter lists and terminating semicolons are dropped. All language syntax is presented in a sans serif font, while semantics are presented in italics.

#### 3.2.1 Program Evaluation

To evaluate a program in KOOL, the program must be inserted into an initial state on which the rewrite process can be started. The state will then proceed through a number of transitions until it reaches a final state (assuming it terminates), which could represent either an error execution, such as one in which an exception is thrown but not caught, causing the program to crash, or a successful execution, yielding some final output and no further execution steps. This is modeled using an `eval` function, shown in Figure 6.

![Figure 4. KOOL Syntax](image)

![Figure 6. Program Evaluation](image)
Note that the function takes the program and the program input, and then provides default values for all other state components. The semantics will process all class definitions in the program within the cset and execute the program expression. Since there are no features yet in the language that can introduce nondeterminism, a given program will always yield the same final state, with the final result in output, if it terminates.

### 3.2.2 Object Creation

Since all values in KOOL are objects, object creation is one of the core sets of rules in the semantics. At a high level, several distinct steps need to be performed:

- Each class that makes up the object's type – the current class and all superclasses up to and including Object – can contain declarations, and since any of these declarations could be used, depending on the method invoked and the current scope, a "layer" for each class that makes up the object needs to be allocated, containing instance variables for all fields;
- The layers need to be combined into a single object such that lookups occur correctly; specifically, lookups should start at the correct layer, based on the static scoping rules for the language;
- The newly created object, with the various layers and information about the its dynamic class, then needs to be returned.

The rules for object creation are shown in Figure 7, along with an example. Rule (12) handles the new expression. new is provided a class name (C) and a possibly empty list of arguments (El) to be provided to the class constructor. The desired result is that a new object of class C will be created and the class constructor for C, which must also be named C, will be invoked with the arguments El. To accomplish this, the createObj continuation item is placed on top of the continuation with the class name and the argument list. After this the invokeAndReturnObj continuation item is added. The purpose of this is to invoke the specified method and then return the target object as the result. How this is handled can be seen in Rule (13), where invokeAndReturnObj is just replaced with an invoke of the same method, a discard to remove the value returned by the method, and finally the target object, effectively replacing the return value of the method with the target object. So, this will take the new object, send it the constructor message with the provided arguments, and return the object, which is what we need. More details about handling message sends are provided in Section 3.2.3.

The rules that actually create the object start with Rule (14). Since the creation rules will use the current environment to store what is in scope for each layer in the object, we first want to save the environment so we can recover it when we are finished and also clear it, so names from the current environment do not "leak" into the object. This is done by putting the environment Env on the continuation and setting the env state attribute to . Also, the continueObj continuation item is changed to a mkObj continuation item, which contains two elements: the current layer that is being built and the object that has been constructed so far. The object, also represented as a set, is initialized with the dynamic class, which matches the class name in the new statement, and a default environment for Object, which is empty. We also set the current layer being build to the dynamic class, since we need to start with this layer and work up the inheritance tree towards Object.

Rule (15) shows the base case of the recursive creation, which is when we reach class Object. Here, we just take the current object and return this as the result of mkObj. Rules (16) and (17) show how the environment layers are configured for classes other than Object. In Rule (16), for class C, we want to allocate space for all fields in the class and store them in the environment layer assigned to this class in the object being created. To allocate space for the fields, the bind continuation item is used. This item is defined to take a list of names, add the names to the environment, and allocate storage for each name. Since there are no values on top of the bind, each name will be assigned the initial value nil in the store. fdfs is used to retrieve the fields of class C, as defined in class set CSet. The layer continuation item then says that a new layer should be formed from the resulting environment.

The process of forming this layer is shown in Rule (17), where the current environment is added into the object definition as [C,env], or the environment associated with class C. The environment is then cleared out, and the process is continued with the parent class of C. Eventually this will reach Rule (15), return the object, call the constructor, and then yield a new, initialized object.

Rules (5) and (6) show the process of getting the parent class and the fields for a given class and class set, respectively. In Rule (5), the class name is used to match against the parent class name in the set representing the class, while in Rule (6) the class name...
Figure 7. Object creation

instead matches against the list of names representing the fields of the class.

An example object creation can be seen in Figure 7. The class, ColorPoint, contains two fields, c and p. It extends class Point, which contains three fields, x, y, and p. This class extends Object by default, which has no fields. As can be seen in the Figure, the continuation item createObj(ColorPoint) will lead to the continuation item mkObj with the initial class and an initial version of the object. Each step will then either bind fields from the class or add those fields as a new layer in the object environment. Note that there are two copies of field p, one at location L3 and one at location L5. The copy chosen will depend on the method being executed – a method from class Point will use the copy of p at L5, while a method from class ColorPoint will use the copy of p at L3. Once the creation reaches Object, the new object has been created and is returned. The next step, sending the ColorPoint message with the constructor arguments, is not shown.

3.2.3 Message Sends

Message sends are by default dynamic in KOOL. Because of this, lookups for the correct method to invoke should always start with the dynamic class of the object, working back up the inheritance tree towards the Object class. There are two exceptions to this rule. First, with super calls, the correct instance of the method to call should be found by starting the search in the parent class of the current class in the execution context. Second, with constructor calls, the lookup order is the same, but the method name will change, since constructor method matches the class name in which they are defined. The first exception is part of the semantics for super, not shown here, while the second is part of the core send semantics. The rules for message sends are shown in Figure 8.

The first rule, Rule (18), is used to start processing the message send. The message target, E, and the message parameters, E₁, are evaluated, with the name of the message, X, saved in the invoke continuation item. In Rule (19), given the result of the evaluation of E and E₁, the current stream of execution from the continuation (K), the control state (Cᵣ), and the current environment (Env), object (O'), and class (C') are pushed onto the method stack (with the environment on top of the remaining continuation, so it will be recovered when this continuation is run), ensuring that the current execution context can be quickly restored when the method exits. The continuation is changed to put the value list (Vl) that resulted from evaluating the method parameters on top of the getMthd continuation item, which is on top of a different invoke continuation item that takes no parameters. This indicates that we want to find the method to invoke, based on the method name, class name, and class set, and then invoke it with actual arguments Vl. The environment is cleared to ensure names in the current environment aren’t introduced into the environment of the executing method, the current object is replaced with the object the message target evaluated to, and the current class is replaced with the dynamic class of the target object.

Rule (20) shows the result of finding the method. A pair of the class in which the method was found and the method itself are on top of the invoke continuation item. This will be replaced with a bind of the method parameters and declarations (X₁, X₁'), followed by the method body (K'). The values in Vl will then be bound to the names in X₁, with the declarations X₁' bound to nil, giving us the proper starting state for executing the method body (by default declarations are assigned a value of nil until they are assigned into).

Rule (21) shows the result of reaching the end of a method. All methods are automatically ended with a “return nil” statement when they are preprocessed, so all methods will end with a return. When return is encountered, the return continuation item and the rest of the continuation following return are discarded, replaced by the continuation on the method stack. The rest of the control state, the current object, and the current class are also reset to the values from the method stack. The value on top of the continuation is left untouched, however, since this will be returned as the result of the method.

3.2.4 Exceptions

KOOL includes a basic exception mechanism similar to that in many other OO languages, such as JAVA or C++. Code can be executed in a try block, which has an associated catch block. When an exception occurs, control is transferred to the catch block which is encountered first as the execution stack is unwound. The
exception is bound to a variable associated with the `catch`, with different types of exceptions used for different exception conditions (nil reference, message not supported, etc.). Along with system-generated exceptions, custom exceptions can be created, and both can be thrown using a `throw` statement. The semantics for Exceptions can be seen in Figure 1. One important point is that exceptions are not just added by the programmer—they are used in the language semantics as well. For instance, not shown in the rules for message sends, several possible exceptions can be raised, including an exception generated when a nil variable is used as a message target and an exception thrown when a target object does not support a message. An example where an exception is thrown by the semantics rules can be seen in Figure 10, where an exception is thrown on a lock release when the lock was not already held.

Rule (1) shows the semantics for a `try-catch` statement. The current control context (Ctrl), environment (Env), object (O), and class (C), along with an exception continuation, are all put onto the exception stack. The exception continuation is made up of a binding to the name X from the catch clause, the statement S' associated with the catch clause, the current environment Env (so we recover the current environment and remove the binding of the caught exception to X), and the current continuation, K. Finally, the try-catch block is replaced with the statement (S) from the try clause and the `popStack` continuation item. So, for a try-catch block, we will execute the statement in the try clause. If this finishes, we will pop the exception stack and continue running. If an exception is thrown, we will instead want to execute the catch clause, binding the exception to the name in the clause, running the body of the catch, and then continuing with the remainder of the computation after the end of the try-catch statement.

The left-hand side of Rule (2) handles the no exceptions case, where the pop marker is found during normal execution. In this case, the top of the exception stack is popped, but no other changes occur. When an exception is thrown, the right-hand side of Rule (2) is used. In this case, the current context information is replaced with the information that was saved on the exception stack, and the exception stack is popped. The value V that represents the exception is left on top, which will cause it to be bound correctly and made available to the catch statement (in Rule (1) the top of the stored continuation was a `bind`, so the value will be bound to the name from the catch clause). Since the computation after the end of the try-catch is part of the exception continuation, the computation will continue correctly after the end of the exception handler.

### 3.2.5 Runtime Type Inspection

KQOL allows the dynamic type of an expression to be checked at runtime using a `typecase` construct. This construct contains a sequence of cases, each with a class name and a statement. If the class name in the case matches either the dynamic class type of the expression or a superclass of the dynamic class type, the statement is executed. Cases are evaluated from top to bottom, with an optional `else` case that always matches. The rules for runtime type inspection are shown in Figure 9.

Since the parsing step can convert the `else` case to a case matching `Object`, we assume in the semantics that there is no longer a designated `else` case. When a `typecase` is encountered, Rule (22) shows that this is replaced with an evaluation of the expression E on top of the `getInheritsSet` continuation item, followed by the `Cases` that will be checked. When the expression E is evaluated to an object value, Rule (23) shows the start of building the set of class names that will be used in the check against the cases. The `getInheritsSet` continuation item is changed to another item with the same name but three parameters, a class name, a set of class names and a set of classes, with the first two parameters set to the dynamic class of the expression result, C. With the set of classes for the expression calculated, the remaining three rules, Rules (24), (25) and (26), apply sequentially to process the cases. In the first, a matching case is found, so the class name set and the cases list are both discarded, replaced by the case statement S. In the second, the case does not match, but there are cases left in the list, so the current case is removed so the next can be tried. In the third, there is no match, and there are no cases left in the list, so both the cases list and the class name set are discarded, allowing control to fall through to whatever was after the case statement. This provides for the intended semantics—the statement of the first matching case (if any) will execute, and control will pick up with the next statement after the end of the `typecase`.

### 3.2.6 Primitives

Since all operations are modeled as message sends, there isn’t a native way in the language to, for instance, add two numbers, or output a string. Yet, at some point, 5 + 3 actually has to yield 8. This is done using primitives, a concept familiar from other object-oriented languages, such as SMALLTALK. Each class which is used to represent a primitive value, such as `Integer`, contains a field that
3.3 Extending KOOL with Concurrency

The dynamic semantics from Section 3.2 does not support any concurrent operations – as defined, KOOL is a sequential language, with a single thread of execution. In this section we illustrate the process of extending a language defined with K by adding concurrency support to KOOL. To support concurrency, a new statement, spawn, will be added to create new threads; threads will be able to acquire and release locks on specific objects using acquire and release statements; and accesses to shared memory locations should compete – if two threads both assign a value to a shared variable, the resulting value should be nondeterministic, based on the actual execution order of the threads.

With multiple threads, and thus multiple concurrent streams of execution, some of the state components will need to be duplicated. This includes the current object, the current class, the entire control, and the environment. This allows each thread to have enough local information to execute without interfering with the execution of other threads (for instance, threads should not share the current class, since a message send in one thread would potentially interfere with a message send in the other if they did). However, some information, such as the set of classes and the store, will be global to all threads. The grayed sections of Figure 5 represent the changes in the state to enable concurrency. The additional syntax and new rules for the dynamic semantics for concurrency in KOOL are shown in Figure 10. It is important to note that, even though we are making a fairly significant change to the language, we have only had to manually change one rule to work over the new state configuration, Rule (27). This is the concurrent version of Rule 11. Rule 27 makes use of the newThrd continuation item to create a new execution thread and set up the starting state appropriately.

The spawn statement creates a new thread based on a provided expression. The expression is evaluated in the new thread, meaning any exceptions thrown by the expression when it is evaluated will be handled in the new, not the spawn’ing, thread. Rule (28) shows the semantics of spawn. Here, the expression E in the spawn statement is given to the newThrd item, along with the current environment(Ev), the current object (O), and the current class(C). spawn returns no value, so it is just removed from the continuation. Rule (29) shows how the new thread is actually created. The passed expression, environment, object, and class are plugged into the proper state components nested within the new thread. This will start the new thread for expression E running in the proper environment. When the thread finishes, it needs to be removed, with any locks it holds being removed from the global busy lock set. This is illustrated in Rule (30).

Along with the ability to create new threads, we also need to be able to acquire and release locks. This is done using the acquire and release statements. The semantics for acquire is shown in Rules (31) and (32). In Rule (31), a lock is acquired on an object V that the current thread already has a lock on. This just increments the lock count on this object from N to the successor of N. In Rule (32), a lock is acquired on a value V that no thread, including the current thread, has a lock on. This adds the value V and a lock count of 1 to the thread’s holds set, while also adding V to the current global lock set LS. This rule is boxed since multiple lock attempts in situations where no thread already holds a lock on a given object can compete.

The semantics for release is shown in Rules (33), (34), and (35). In rule (33), a lock on value V with lock count 1 is released. This removes the lock from both the local holds set and the global busy set. Rule (34) shows what happens when a lock on value V with lock count greater than 1 is released – here, the count simply goes from \(s(N)\) (the successor of \(N\)) to \(N\). Finally, if there is an attempt to release a lock that the thread does not hold, an exception should be thrown. This is shown in Rule (35), where an attempt to release a lock on V not held by the thread results in a LockNotHeldEx exception being thrown.

A sample concurrent program, the thread game, is shown in Figure 11, with a Java version discussed in more detail in Section 5.2.1. It has been proven that the variable x can take the value of any natural number greater than 0 [36].
4. Java in K

We next discuss how to define an existing OO language, Java, in K. A preliminary effort to specify Java in rewriting logic can be found in [12], where Maude was adopted as the specification formalism. The previous effort partially motivates the design of the K notation, which specifically aims at improving the effectiveness of defining programming languages. Compared with our previous work, the current K-based approach provides a more concise and more modular way to specify Java language features. For example, many modules can be shared between the Java specification and the KOOL specification, although the languages are different in many aspects, with specifications developed by different authors and with different state infrastructures. One can also easily extend the specification to introduce new advanced features into the language, as discussed in Section 3.3 and Section 4.3.

Our formal specification is based on the Java language specification 2.0 ([28]), which is effective up to Java 1.4. We currently cover most major features of Java 1.4, including object creation, dynamic type checking, multiple threads and synchronization, etc., and will support the full Java language in the near future. The K specification is translated into Maude (currently by hand, although we are working on an automatic translation) to obtain the executable specification along with static analysis tools for the defined language. The latest version of our specification can be found at [1]. To be interpreted with the executable Maude specification, the complete source code of the program should be provided, including all the Java classes from the Java distribution that are referenced in the program. To support those library classes that do not come with a Java implementation, e.g., native classes, there are three options. The first is to provide an equivalent Java implementation. The second is to specify their functionality directly in the specification. For example, we define the behavior of println in our specification. The third, which we do not yet support, is to invoke the external library via some mechanism provided by the underlying rewrite engine, e.g., the TCP/IP socket support in Maude.

4.1 State Infrastructure

As mentioned above, the state infrastructure is the first thing to decide for a programming language specification. Figure 12 shows the state infrastructure adopted in our Java specification. Obviously, it shares most major components with the one used in the concurrent KOOL specification (Figure 10). But since Java has built-in support for concurrency, we design the state infrastructure to allow multiple threads from the beginning, resulting in a two-level structure which distinguishes the global state from the local state of the thread. The global state contains the components shared by different threads, e.g., the store which maps locations into values, including local and static fields of classes. The thread state contains information which is needed to process the thread and distinguish it from other threads, including the continuation to execute in the thread, three stacks for efficient control flow changes, the environment that maps variables into locations, the object in which the thread is executed, the unique id of the thread, and the locks held by the thread. The global state may contain several instances of thread states corresponding to different threads.

Another important design decision in OO language specifications is the state of an object. For Java, we represent the object with three attributes, as shown in Figure 13, including the type of the object in the current context, the actual type of the object, and the fields of the object. Since Java supports dynamic types, an object may be regarded as being of different types in different contexts, which is encoded in the first component of the object state. The current type should be either a superclass of or the actual type of the object. The value of the object is constituted by the values of its fields. In Java, the fields of an object are determined by several classes, i.e., the current class and all superclasses. Every class may contribute some fields to the object, and the visibility of these fields may be changed when the object is casted to different types or when different methods are invoked. Therefore, instead of encoding the fields in one integrated internal environment of the object, we represent them using a set of (type, environment) pairs that preserves the mapping between the fields and corresponding source classes, like in KOOL. This set, called the object environment, is also used in the global state to store the static fields for every class.

4.2 Dynamic Semantics

Presently, the Java specification contains about one thousand equations and rules. It is impossible to go over all of them in this paper, so we only discuss a few non-trivial features that are representative of the Java definition. It is worth noting that KOOL and Java support many similar features and the corresponding modules can be shared between both languages with at most minor modifications, even though the state infrastructures are different. For example, the object creation process in both languages is almost the same except that the Java object representation contains two types, and when the object is created, the two types are both set to the created type. For many other formalisms, changes to the critical semantic structures, like the state infrastructure, may affect many rules and thus reduce the reusability and modularity of the specification. In this sense, K provides a possible solution to building a language with desired features by composing existing language modules.

4.2.1 Method Invocation

Figure 14 shows the rules to invoke methods in Java 1. Similarly to KOOL, Java allows subclasses to override methods of superclasses and uses dynamic method dispatch, which determines the (non-static) method to invoke according to the actual type of the object.

1 For here on, only rules for normal behaviors are given.
object instead of its current type. In other words, if class A overrides a method M in its superclass A', then whenever one invokes M on an object of A, the implementation of A will be called, even if the object is casted to A'. This is captured by Rule 37 which uses the actual type T to search for the method. Compared to the invo-
cation rules for KOOL (Figure 8), the Java rules are more complex. This is for two reasons. First, the search for the appropriate method to invoke is more complicated. Java allows overloaded methods, which requires additional checks of type compatibility between de-
clared parameters and actual arguments\(^2\). We postpone the detailed rules about this search process to Section 4.3 when we extend Java with inner calls.

Second, there are more cases to specify when invoking methods in Java, namely, static methods, synchronized methods, and some system methods. Static methods will be executed in the static context of the class instead of an object. Therefore, in Rule 41 an object with empty object environment (meaning with no object fields) is created for the method invocation, because the static method can only access static fields of the class which are stored in the static component of the global state. For synchronized methods (Rule 38), before the method is executed, the thread has to lock the object; and after the method is executed, the thread releases the lock. More discussion about synchronization and locks can be found in Section 4.2.3. Some system methods have to be specified to properly interpret the program, for example, the I/O methods and the thread lifecycle methods (e.g., Thread.start(), Object.wait() and so on). These methods are recognized in the method search process and will be turned into special commands, which are modularly specified like other features of Java. For example, Thread.start() is turned into the new Thread command, which has a definition similar to that in Figure 10.

4.2.2 Exceptions

Exceptions were not specified in our previous effort based on Maude. Here we give an efficient way to define the exception mechanism in Java, as shown in Figure 15. The specification is more complicated than that for KOOL (Figure 1), because Java uses designated classes for exceptions and provides a more expressive syntax, allowing multiple catch clauses and an optional final clause. Therefore the specification for the try...catch statement is divided into two rules. First, we construct a continuation for the try statement (Rule 42) and then we use a command, buildEStack(), to build the exception stack for every catch clause (Rule 43), similarly to Rule 1. Since every try...catch statement may push multiple exceptions into stack, we use restoreEStack() instead of a pop operation to restore stack when leaving the try block.

The rules for finally are more involved, and are not shown here. According to the Java specification, the finally clause needs to be executed whenever the control flow leaves the try statement (normally or exceptionally). Therefore, in addition to executing the finally block after the try block, we also propagate it through the stack, fstack, and lstack to capture the correct semantics. Also, there is a subtle situation in the Java semantics caused by the lock mechanism, namely that when the control flow jumps out of some synchronized blocks and/or functions, the corresponding locks have to be released. So we use the releaseLock() command in Rule 45 to compare the control states before and after the catch, and then release locks according to the difference. Detailed rules for this command are out of the scope of this paper.

4.2.3 Concurrency

One advantage of rewriting logic is that it naturally supports concurrency. With a properly formalized language specification based on rewriting logic, one may be able to model check the concurrent properties of a program without extra effort. For example, Section 5 shows some results of model checking concurrent Java programs using the translated Maude specification. More discussion about model checking Java program can be found in Section 5. Here we focus on how to properly specify the concurrent semantics of Java using K.

The concurrency in Java is composed of four features, i.e., thread spawning, synchronization with locks, the wait/notify mechanism, and shared memory accesses. Some of these have already been discussed: the rules to handle threads and locks in Figure 10 can be easily reused for Java and the rule for synchronized methods is given in Section 4.2.1. We next focus on the wait/notify mechanism and shared variable accesses. Figure 16 gives the specifications for synchronized blocks (Rule 47) and the wait/notify mechanism. According to the Java language specification, when a thread calls wait() on an object, it should already hold a lock on the object, and will release its lock and wait for notification. The thread will then try to re-acquire the lock on the object (setting the held count the same as the count before releasing) when it is awakened (Rule 48). When notify() is called, a waiting thread is randomly chosen to awake (Rule 49). A rewrite rule is used for this behavior because the choice is made nondeterministically. When

\(^2\)Note that one may use a static transformer to rename all the method declarations and corresponding invocations in the Java program to eliminate the overloaded methods (by processes such as name mangling). By doing so, the method search process can be simplified to check only the name of the method, as KOOL does.
notifyAll() is called, all waiting threads are awakened (Rule 51). Note that since the rules are tried in order, Rule 50 and Rule 52 are executed only when Rule 49 and Rule 51, respectively, cannot be applied, handling the situation where there is no waiting thread.

An important characteristic of concurrent programs is that threads need to compete for shared memory locations. A simple way to specify this feature is to make the threads compete for any variable that they try to access, just like the rules used in KOOL (Rule 4 and Rule 3). However, it is inefficient because most variables are not shared in practice. Therefore, it may cause many unnecessary applications of rewrite rules (versus equations) during the interpretation of the concurrent program, which increases the complexity of model checking exponentially. To improve the efficiency of model checking, our current specification keeps track of the ownership of every location in the store, as shown in Figure 17. The element of the store is now a triple, containing a location, the corresponding value, and the id of the owning thread. All newly created values are regarded as having no ownership until they are stored into some location; then the ownership of the stored value is set to the owner of the location (Rule 54). The command setOwner() is used to go through all the locations referred by the value, i.e., through fields of objects or elements of arrays, and set the owner of those locations. So when a value is set to a shared location whose owner id is -1, those locations that can be reached from that value will be marked as shared too (Rule 55). And only accesses to shared locations need to compete (Rule 55 and Rule 57). Initially, all locations referenced by static fields are marked as shared, and so are the visible (public and, in some cases, protected/default) fields of a thread object if the object is stored in some variable of another thread.

### 4.3 Extending Java with Inner Calls

In this section, we show the strength of our approach in helping the design of new language features by extending Java with inner calls. This extension is inspired by the work in [21]. In Java, subclasses can arbitrarily override methods from superclasses. A superclass’s method can be reused via super calls. In other words, a subclass implementor has full control of the functionality of the subclass. In some other languages, e.g., Beta, the method of the superclass cannot be overridden arbitrarily. The subclass’s method is invoked only through inner calls made in a superclass method. This way, the implementor of the superclass is able to control the work that the subclass may perform, helping to enforce class behavior. [21] argues that both forms are useful in practice and supports both in MzScheme. We next show that one can extend the K Java specification to support inner calls in a straight-forward manner. This allows the new feature to be tried without requiring a modification to an existing Java compiler.

We adopt the convention in [21]. First, we introduce a new method modifier, beta, to the language, as well as a new expression, inner . method (). For a method invocation on an object, the first (closest to Object) compatible method implementation with the beta modifier in the class hierarchy of the object is invoked if it exists; otherwise, the last implementation of the method is called just as the normal method dispatch in Java. When an inner call is executed, the closest beta method implementation between
the current class and the actual class of the object is invoked. Re-
calling the rules for the method invocation in Figure 14, one can see that 
in order to support the inner call semantics, the method search 
process needs to be modified to locate the correct method imple-
mentation. Figure 18 gives the original search process for the Java 
semantics.

In this process, first we use getMethodList to build a method 
list bottom-up according to the method name, and then search 
through this list to locate a compatible method implementation. 
Now we first change findMthd and getMethodList to have one 
more argument, namely, the class where we stop searching and 
modified the original rules correspondingly, as shown in Figure 19. 
Rule 63 is to translate the old-fashion findMthd command into 
the new version, and Rule 64 then uses the new getMethodList to build 
the method list. The new version of getMethodList performs just 
like the old one (Rule 67 and Rule 66), except that it stops when 
the designated top class is encountered (Rule 66) since inner calls 
do not go beyond the current class of the object. To accommodate 
the method dispatch for beta methods, we only need to add a new 
rule, Rule 65, for building the method list. This rule simply adds 
any beta method to the front instead of the end of the method 
list. This way, the following process will search the beta methods 
top-down first and then other methods bottom-up. Rule 69 and 
Rule 71 then handle the inner call expression by building the 
method list up to the current type of the object using the new 
getMethodList command. So the method search starts from the 
class for the current type down to the first compatible beta method 
or the farthest compatible method, which precisely defines the 
behavior of the inner call.

One restriction on combining super calls and inner calls in a 
Java-like language is mentioned in [21], namely that super calls 
are disallowed in beta methods. Because of the limited space, we do 
not include the rules for super calls, which simply search from the 
immediate superclass upwards until a compatible method is found. 
To be compatible with the restriction, the only modification required 
is to check the modifiers of the found method. If it contains the beta 
modifier, an exception is thrown. Note that [21] does not mention 
what they do with the disallowed super call. But it is easy to change 
the behavior in our specification to be compatible. The other subtle 
situation is that when an inner call is made in the last subclass, 
which means that the current type of the object is its actual type, 
then nothing happens. This is handled by Rule 70. In summary, to 
support the beta-like inner call in Java, we removed 2 rules in the 
original specification, added 9 new rules, and modified 1 rule. All 
the work is done by the author in an hour after understanding the 
semantics of the inner call in [21]. With this modified specification, 
the user now has an interpreter for a "new" Java language that 
supports inner calls.

5. Experimental Results
To evaluate our Java specification, we translated the K rules into 
Maude in order to create an executable rewriting interpreter for 
Java. Using the underlying fair rewriting engine, the translated 
 specification can be used as an interpreter to simulate fair compu-
tations of Java programs. We can also formally analyze the concur-
rent properties of multithreaded Java programs, such as data races 
and deadlocks, based on the search tool and the model checker pro-
vided by Maude. We next show some preliminary experimental re-
 sults of analyzing Java programs based on our language definition. 
Note that although some existing tools have been compared with 
our approach, the evaluation is still in an early stage and is not 
comprehensive.

5.1 Simulation
Our Maude specification provides executable semantics for Java, 
which can be used to execute Java programs in source code formats. 
This simulator can also be used to execute programs with symbolic 
inputs. Maude’s rewrite command provides fair rewriting with 
respect to objects, so no thread ever starves, without imposing a 
specific scheduling algorithm.

5.2 Breadth-First Search Analysis
Using the simulator (Section 5.1), one can explore only one possi-
ble trace (modeled as a sequence of rewrites) of the Java program 
being executed. Maude’s search command allows exhaustive ex-
ploration of all possible traces of a Java program. The breadth-first 
nature of the search command gives us a semi-decision procedure 
to find errors even in infinite state spaces, being limited only by the 
available memory. Below, we discuss two examples.

5.2.1 The Thread Game
The Thread Game is a simple multithreaded program which shows 
the possible data races between two threads accessing a common 
variable (see Figure 20). Each thread reads the value of the static 
variable c twice and writes the sum of the two values back to c. 
Note that these two readings may or may not coincide. As discussed 
in 3.3, it has been shown that c can potentially hold any natural 
number greater than 0 during an infinite execution [36]. 
We can use Maude’s search command to address this question 
for any given N. The search command, using breadth-first search, 
can find one or all existing solutions (sequences) that lead to c 
holding the value N. Table 1 presents, for several values of N, the 
amount of time (in seconds) to find a solution.

<table>
<thead>
<tr>
<th>N</th>
<th>50</th>
<th>100</th>
<th>200</th>
<th>400</th>
<th>500</th>
<th>1000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Java</td>
<td>2.7</td>
<td>6.6</td>
<td>17</td>
<td>54.7</td>
<td>2m</td>
<td>5.1m</td>
</tr>
</tbody>
</table>

Table 1. Thread Game Times.

5.2.2 Remote Agent
The Remote Agent (RA) is an AI-based spacecraft controller de-
veloped at NASA Ames Research Center, part of the software compo-
nents of NASA’s Deep Space 1 shuttle. However, Deep Space 1’s 
software deadlocked away from the Earth and consequently had to 
be manually interrupted and restarted from ground. The blocking 
was due to a missing critical section in the RA that led to a data-
race between two concurrent threads, which then caused a deadlock 
[22, 23].

The RA (Figure 21) consists of three components: a Planner that 
generates plans from mission goals; an Executive that executes the 
plans; and a Recovery system that monitors the RA’s status. The 
planner is responsible for generating plans, the executive is re-
sponsible for executing them, and the recovery system is re-
sponsible for handling failures. The RA is a complex system that 
requires sophisticated scheduling algorithms to ensure that all 
threads are scheduled fairly. The RA uses Maude’s search command 
to explore all possible schedules and find errors even in infinite 
state spaces, being limited only by the available memory. Below, we 
discuss two examples.

1 By not committing to a specific thread scheduling mechanism, we have the 
advantage of detecting the violations that may happen in some scheduling 
schemes, but not in others.
Executive contains features of a multithreaded operating system, and the Planner and Executive exchange messages in an interactive manner. Hence, this system is highly vulnerable to multithreading errors. Events and tasks are two major components. In order to catch the events that occur while tasks are executing, each event has an associated event counter that is increased whenever the event is signaled. A task then only calls \texttt{wait\_for\_event} in case this counter has not changed, hence, there have been no events since it was last restarted from a call of \texttt{wait\_for\_event}.

The error in this code results from the unprotected access to the variable \texttt{count} of the class \texttt{Event}. When the value of \texttt{event1.count} is read to check the condition, it can change before the related action is taken, and this can lead to a possible deadlock. This example has been extensively studied in [22, 23]. Using the search capability of our system, we can find the deadlock in 0.1 seconds, while the tool in [38] finds it in more than 2 seconds in its most optimized version.

5.3 Model Checking

Maude’s model checker is explicit state and supports Linear Temporal Logic. This general purpose rewriting logic model checker can be directly used on the Maude specification of the concurrent Java semantics. This way, we obtain a model checking procedure for Java programs for free. The user has to specify in Maude the Java semantics. This way, we obtain a model checking procedure that can be directly used on the Maude specification of the concurrent Java semantics.

5.3.1 Dining Philosophers

See Figure 22 for the version of the dining philosophers problem that we have used in our experiments. The property that we have model checked is whether all the philosopher can eventually dine.

The model checker generates counterexamples, in this case a sequence of states that lead to a possible deadlock. The sequence shows a situation in which each philosopher has acquired one fork and is waiting for the other fork. Currently, we can detect the deadlock for up to 9 philosophers.

4 Other model checkers, such as JPF, can do larger numbers. We do not apply partial order reduction techniques, which would shrink the state space.

\footnote{All the performance results given in this paper are in seconds on a 2.4GHz PC.}
class Planner extends Thread {
    public synchronized void wait_for_event() {
        try { wait(); } catch (InterruptedException e) { } }
    public synchronized void signal_event() {
        event1.count = (event1.count + 1) % 3; notifyAll(); }
}

class Planner1 extends Thread {
    Event event1, event2; int count = 0;
    public Planner1(Event e1, Event e2) {
        this.event1 = e1; this.event2 = e2; }
    public void run() {
        int count = 0;
        while(true){
            if (count == event1.count) event1.wait_for_event();
            event1.count = event1.count; event2.signal_event();
        }
    }
}

class Planner2 extends Thread {
    Event event1, event2; int count = 0;
    public Planner2(Event e1, Event e2) {
        this.event1 = e1; this.event2 = e2; }
    public void run() {
        int count = 0;
        while(true){
            event1.signal_event();
            if (count == event2.count) event2.wait_for_event();
            event2.count = event2.count; count = (count + 1) % 3; notifyAll();
        }
    }
}

class Philosopher extends Thread {
    int id; Fork F1, F2;
    public Philosopher(int i, Fork f1, Fork f2)
    { this.id = i; this.F1 = f1; this.F2 = f2; this.id = i; return; }
    void Dine() { System.out.print(id); return; }
    public void run() {
        synchronized (F1) synchronized (F2) { Dine(); }
    }
}

class Fork {
    public int num; }

class DiningPhilosophers2 {
    public static void main(String[] args) {
        Fork F1 = new Fork(); Fork F2 = new Fork();
        Fork F3 = new Fork(); Fork F4 = new Fork();
        new Philosopher(1, F1, F2).start();
        new Philosopher(2, F2, F3).start();
        new Philosopher(3, F3, F4).start();
        new Philosopher(4, F4, F1).start();
    }
}

class Executive extends Thread {
    Event event1, event2; int count = 0;
    public Executive(Event e1, Event e2)
    { this.event1 = e1; this.event2 = e2; }
    public synchronized void signal_event() {
        event1.count = event1.count; event2.signal_event();
    }
    public synchronized void wait_for_event() {
        int count = 0;
        if (count == event1.count) event1.wait_for_event();
        event1.count = event1.count; event2.count = event2.count;
    }
}

class CheckPoints {
    public static int sreturn = 0;
    public static int stop = 0;
    static public void main (String argv[]) {
        Connector c1, c2, c3;
        c1 = new Connector(); c2 = new Connector();
        c3 = new Connector();
        new Stage(1, c1, c2).start();
        new Stage(2, c2, c3).start();
        new Listener(c3).start();
        for (int i=1; i<4; i++) c1.add(i);
        c1.stop(); CheckPoints.stop = 1;
    }
}

class Connector {
    public int queue = -1;
    public synchronized int take() {
        int value;
        try { wait(); } catch (InterruptedException ex){ };
    }
    return value; queue = queue - 1;
    return value;
    public synchronized void add(int o) { queue = o; notifyAll(); }
    public synchronized void stop() { queue = 0; notifyAll(); }
}

class Stage extends Thread {
    int id; Connector c1, c2;
    public Stage(int i, Connector a1, Connector a2)
    { id = i; this.c1 = a1; this.c2 = a2; }
    public void run() {
        int tmp = -1;
        while (tmp != 0)
            if ((tmp = c1.take()) != 0)
                c2.add(tmp+1);
            if (id == 1) CheckPoints.sreturn = 1;
    }
}

class Listener extends Thread {
    Connector c;
    public Listener(Connector con) { this.c = con; }
    public void run() {
        int tmp = -1;
        while ((tmp = c.take()) != 0)
            System.out.print(tmp);
    }
}

class Main {
    public static void main (String argv[]) {
        Connector c1, c2, c3;
        c1 = new Connector(); c2 = new Connector();
        c3 = new Connector();
        new Stage(1, c1, c2).start();
        new Stage(2, c2, c3).start();
        new Listener(c3).start();
        for (int i=1; i<4; i++) c1.add(i);
        c1.stop(); CheckPoints.stop = 1;
    }
}

Figure 21. Remote agent

Figure 22. Deadlocked Dining Philosophers

5.3.2 2-stage Pipeline

Code in 23 implements a pipeline computation, where each pipeline stage executes as a separate thread. Stages interact through connector objects that provide methods for adding and taking data. The property we have model checked for this program is related to the proper shutdown of a pipelined computation, namely, “the eventual shutdown of a pipeline stage in response to a call to stop on the pipeline’s input connector”. The LTL formula for the property is □(c1stop → ◊(stop2return)). Model checking the property returns true in 20 minutes. This compares favorably with the model checker in [38] which, when not using partial order reduction takes more than 100 minutes.

6. Conclusion, Related Work and Future Work

In this paper we showed how the K rewrite logic framework can be used for rapid prototyping, design and experimentation with complex object-oriented programming languages and corresponding analysis tools. Despite their rigorous mathematical underlying foundations, K language definitions have quite an operational flavour. K was illustrated by first defining KOOL, an experimental concurrent object-oriented language with exceptions, and then by...
discussing the definition of JAVA and an extension of it with inner calls. Generic rewrite logic tools, such as efficient rewrite engines and model checkers, can be used on K language definitions and yield interpreters and corresponding formal program analyzers at no additional definitional cost.

We believe that the K-based approach to define OO languages discussed in this paper gives a good balance among often opposite factors such as: mathematical rigor (it is denotational and its initial model semantics is open to inductive reasoning), executability (it is operational by term rewriting), formal analysis, ease of understanding and teaching, tool support, scalability. Rewriting logic semantics of programming languages has been taught at the University of Illinois for several years by now [41], to both graduate and undergraduate students. While term rewriting concepts and techniques tend to be grasped without difficulty, we learned from interaction with students of some limitations of K. For example, it takes students already familiar with SOS some time to “think backwards”, that is, to adapt to the top-down style of “deriving” executions of programs in K, rather than bottom-up like in SOS. Also, since currently there is no parser or translator for K, its manual translation to rewriting logic theories may be tedious (by having to declare all the variables, sorts, operations, and to mention the contexts of rules twice, both in the left and the right hand sides of rules) and consequently error prone. Also, since the underlying semantics of rewriting logic is almost transparent, many students fail to perceive it and thus end up thinking of K as a purely operational technique to “implement” interpreters for languages, regarding its formal semantics and analysis capabilities as “something else”, rather than as projections of the same general semantic principle. Also, again because of lack of direct tool support for K, some students find its compactness (see, e.g., Figure 9) to be a plus, others a minus.

There is much related work on defining programming languages in various computational logical frameworks. We cannot mention all these here, but we refer the interested to the K report [42] for a comprehensive discussion of the various techniques, comparisons, their advantages and limitations. We here only list a few of them which are, in our view, closer in purpose to our approach, in the sense that they aim at more than just implementing interpreters for various programming languages, but rather give languages formal definitions that can be used not only for execution, but also for the explicit goal of formal analysis. By being both a (functional) programming language and a theorem prover, ACL2 [29] is a formalism that allows both definitions and formal analysis of programming languages. The operational semantics of a substantial subset of the Java Virtual Machine (JVM) has been defined in ACL2 [29]. The power of ACL2 comes from its underlying inductive theorem prover, but this greater power requires greater expertise and effort, which makes it more difficult to support more abstract languages. Since ACL2 is inherently sequential, to support concurrency it needs to “implement” or “simulate” a thread scheduler.

Another interesting approach is based on Abstract State Machines (ASMs) [27, 43], which can be regarded at some extent as a “simplified programming language” whose programs consist of one loop that may contain a large number of potentially nested conditional assignments. ASMs can encode any computation and have a rigorous semantics, so any programming language can be defined as an ASM and thus implicitly be given a semantics. ASMs are executable, so, like K, languages defined as ASMs are executable. The main difference between ASMs and rewriting logics is that the former regard a specification as an (abstract) automaton, while the latter as an algebraic specification; the former is operational in nature (no need to explain how a program “runs”) and can be associated models/semanitics, while the latter is denotational/semantical in nature, in the sense that equations and rules define an initial model, which can also be executed and formally analyzed. ASMs were originally sequential, but there are also recent extensions with concurrency; rewriting logic is inherently concurrent, to such an extent that it is artificial to give it a sequential semantics (but one can enforce sequentiality by appropriate specification; see Appendix A). Verification tasks in ASMs are performed manually, and we are not aware of any attempts to develop generic ASM formal analysis tools, e.g., model checkers, that can be seamlessly instantiated into corresponding analysis tools for the languages defined as ASMs. Therefore, we regard our approach as complementary to ASMs, in the sense that we provide new analysis capabilities.

Among the approaches based on term rewriting and related techniques, the first extensive study on defining a programming language equationally, with an initial algebra semantics, seems to be [15]; there, OBJ [19] was used to execute the language specifications via term rewriting. Interesting work in not only defining languages by term rewriting but also in compiling those has been investigated under the ASF+SDF project [44]. Stratego [45] is a program transformation framework based also on term rewriting. Besides the development and use of the K framework, what makes our work different from other language definitional works based on rewriting is precisely the use of a first-order representation of continuations and of AC matching, which turn out to have a crucial effect on the compactness and simplicity of definitions. There is some similarity between our approach and monads [30, 35]. The monad approach gains modularity at the denotational level by using monad transformers to lift program constructs from one level of specification to a richer one. In our case, modularity is achieved by the use of AC matching and context transformers based on the structure of the state, which allow selecting from the state “soup” only those attributes of interest. The complete enumeration of the state attributes is done only once, when defining the eval command. We intend to implement a parser for K and a translator into rewriting logic in the near future. However, as explained in [42], this task is much harder than it may seem and involves researching several important and interesting problems, such as: sort inference, because, for elegance and especially for modularity reasons, we’d like to avoid declaring variables whose sorts can be inferred from contexts - this is a non-trivial problem in the context of subsorting and overloading operation names; tuple operation inference, because, for the same reasons, we’d like to avoid declaring operations needed only for tupling, such as those placing information in stacks. Also, once a parser is implemented, the next step is to mechanize the compilation technique outlined in [26].

References

[9] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer,


### A. Rewriting Logic

Rewriting logic, an extension of equational logics with rewrite rules, proved to be a universal framework for concurrency [31].

#### A.1 Equational Logic and Term Rewriting

Equational logic is perhaps the simplest logic having the full expressivity of computability [3]. One can think of it as a logic of “term replacement”: terms can be replaced by equal terms in any context. An *equational specification* is a pair \((\Sigma, E)\), where \(\Sigma\) is a set of “uninterpreted” operations, also called its “syntax”, and \(E\) is a set of equations of the form \(\forall X \ t = t'\) constraining the syntax, where \(X\) is some set of variables and \(t, t'\) are well-formed terms over variables in \(X\) and operations in \(\Sigma\). Equational logics can be many-sorted [16] (operations in \(\Sigma\) have arguments of specific sorts), or even order-sorted [17], i.e., sorts come with a partial order on them; we use order-sorted specifications in this paper. Also, equations can be *conditional*, where the condition is a (typically finite) set of pairs \(u = u'\) over the same variables \(X\). We
Term rewriting is a related approach in which equations are oriented left-to-right, written \((\forall X) \; t \rightarrow r\), and can be applied to a term \(t\) at any position where \(t\) matches as follows: find some subterm \(t'\) of \(t\), that is, \(t = c[t']\) for some context \(c\), which is an instance of the left-hand-side term (lhs) of some rewrite rule \((\forall X) \; t \rightarrow r\), that is, \(t = \theta(l)\) for some variable assignment \(\theta\), and replace \(t'\) by \(\theta(r)\) in \(t\). This way, the term \(t\) can be continuously transformed, or rewritten. A pair \((\Sigma, R)\), where \(R\) is a set of such oriented rewrite rules, is called a rewrite system. The corresponding term rewriting relation is written \(\rightarrow_R\) and its inverse is written \(\leftarrow_R\). If no rule in \(R\) can rewrite a \(\Sigma\)-term, than that term is said to be in normal form w.r.t. \(R\).

Term rewriting can be used as an operational mechanism to perform equational deduction. Specifically, \(E \vdash_{\Sigma} (\forall X) \; t \rightarrow t'\) if and only if \(t \leftarrow_R E \cdot t'\), where \(E\) is the set of rewrite rules obtained by orienting all the equations in \(E\) from left to right. Even though in theory term rewriting is as powerful as equational deduction, in practice term rewriting is used as a heuristic for equational deduction. A very common case is to attempt the task \(E \vdash_{\Sigma} (\forall X) \; t \rightarrow t'\) by showing \(t \rightarrow_R \cdot t'\), i.e., by reducing both \(t\) and \(t'\) to some common term using \((\Sigma, R)\). In some cases, for example when \((\Sigma, R)\) is confluent and terminates, this becomes a decision procedure for equational deduction.

There are many software systems that either specifically implement term rewriting efficiently, known also as rewrite engines, or that support term rewriting as part of a more complex functionality. Without attempting to be exhaustive, we here only mention (alphabetically) some engines that we are more familiar with, noting that many functional languages and theorem provers provide support for term rewriting as well: ASP/SDF [44], CafeOBJ [11], Elan [4], Maude [10], OJB [19], Stratego [45]. Some of these can achieve remarkable speeds on today’s machines, in the order of millions or tens of millions of rewrite steps per second. Many engines store terms as directed acyclic graphs (DAGs), so applications of rewrite rules consist in many cases of just permuting pointers, which can be indeed implemented quite efficiently. In our language definitions, we often place or remove environments or continuations onto other structures; even though these may look like heavy operations, in fact these do nothing but save or remove pointers to subterms when these definitions are executed.

Because of the forward chaining executability of term rewriting and also because of these efficient rewrite engines, equational specifications are often called executable. As programming languages tend to be increasingly more abstract due to the higher speeds of processors, and as specification languages tend to be provided with faster execution engines, the gap between executable specifications and implementations is becoming visibly narrower. For example, we encourage the curious reader to specify the factorial operation equationally as follows (\(s\) is the Peano “successor”)

\[
0! = 1 \quad \forall n \geq 1 \quad (n + 1)! = s(n) \cdot n!
\]

in a fast rewrite engine like Maude, versus implementing it in programming languages like ML or Scheme. In our experiments, the factorial of 50,000, a number of 213,237 digits, was calculated in 18.620 seconds by the executable equational engine Maude and in 19.280 and 16.770 seconds by the programming languages ML and Scheme, respectively. In case one thinks that the efficiency of built-in libraries should not be a factor in measuring the efficiency of a system, one can define permutations equationally instead:

- Sorts and Subsorts

  \(Nat < Perm < Perms\)

  \[
  \begin{align*}
  \text{Operations} & \quad \text{perm} : Nat \rightarrow Perm \\
  & \quad \text{map-cons} : Nat \rightarrow Perms \\
  & \quad \text{insert} : Perms \rightarrow Perms
  \end{align*}
  \]

  \[
  \begin{align*}
  \text{Equations} & \quad \text{perm}(1) = 1 \\
  & \quad \text{perm}(s(n)) = \text{insert}(\text{perm}(n), \text{perm}(n)) \\
  & \quad \text{insert}(N, (P:Perm ; Ps:Perms)) = \text{insert}(N, P) \cdot \text{insert}(N, Ps) \\
  & \quad \text{insert}(N, (M,P)) = (N,M,P) \cdot \text{map-cons}(M, \text{insert}(N,P)) \\
  & \quad \text{insert}(N, (M)) = (N,M) \cdot (M,N) \\
  & \quad \text{map-cons}(M, (P ; Ps)) = \text{map-cons}(M, P) \cdot \text{map-cons}(M, Ps) \\
  & \quad \text{map-cons}(M, (P)) = (MP)
  \end{align*}
  \]

The above is an order-sorted equational specification; for readability, we used the \texttt{mfix} notation for operation declarations (underscores are argument placeholders) which is supported by many rewrite engines. Also, we declared the list constructor operations with the attribute \texttt{assoc}. Semantically this is equivalent to giving the equation of associativity, but rewrite engines typically use this information to enable specialized algorithms for rewriting and matching modulo associativity; we will discuss matching modulo attributes like associativity, commutativity and identity in more depth shortly. Like in the previous example, we assumed some built-in natural numbers coming with a successor operation.

In our experiments with permutations, the executable equational specifications outperformed the implementations. Maude took 61 seconds to “calculate” permutations of 10, while ML and Scheme took 83 and 92 seconds, respectively. None of these systems were able to calculate permutations of 11. These experiments have been performed on a 2.5GHz Linux machine with 3.5G of memory, and we used the Maude 2.0, PolyML and PLT Scheme (specifically mzscheme), all providing libraries for large numbers. These simplistic experiments should by no means be considered conclusive; our measurements favoring executable specifications may be due to fortunate uses of data-structures in the Maude implementation, or even to our lack of usage of Scheme and ML at their maximum efficiency. While more extensive comparisons and analyses would be interesting and instructive, this is not our goal here; nor to unreasonably claim that executable specifications will ever outperform implementations. All we are trying to say is that the pragmatic, semantics-reluctant language designer, can safely regard the subsumption of language features as implementations, in spite of their conciseness and mathematical flavor.

### A.2 Rewriting Logic

While equational logic (and its execution via term rewriting) provides as powerful computational properties as one can get in a sequential setting, it is not appropriate for specifying or reasoning about concurrent systems. The initial algebra model of an equational specification collapses all the computationally equivalent terms, but it does not say anything about evolution of terms under
concurrent transitions. There are two broad algebraic semantic approaches to concurrency [46]. One builds upon the Platonist belief that models are deterministic, but, by making use of underspecification, one never knows precisely in which model one is. While underspecification is a very powerful approach in semantics, in our programming language definitional framework it suffers from a crucial impediment: it is not executable enough to allow one to execute actual concurrent programs; nevertheless, we make intensive use of underspecification by not specifying implementation details of programming languages, such as how environments, stores, lists or stacks are implemented. A more widely accepted direction of thought in concurrency is to allow nondeterminism in models, typically by means of nondeterministic transitions. Specifications become executable, because all what one needs to do is to randomly pick some transition when more are possible. This is similar to what thread/process schedulers do in concurrent computer systems.

To properly define and analyze programming languages formally, we need a framework which provides natural support for concurrency. In other words, we need a framework where we can state what concurrent language features are meant to be done without artificial encodings due to artifacts of the underlying definitional framework. For example, we consider “unnatural” to define, or simulate, a particular “thread or process scheduler”, just because the underlying definitional framework is inherently sequential. Since both underspecification and nondeterministic transitions seem important for capturing the meaning of programming language features, we would like an underlying framework that supports both.

Rewriting logic [32] is a logic for concurrency, which should not be confused with term rewriting. A rewrite specification, or theory, is a triple \((\Sigma, E, R)\), where \((\Sigma, E)\) is an equational specification and \(R\) is a set of rewrite rules. Rewriting logic therefore extends equational logic with rewrite rules, allowing one to derive both equations and rewrites, or transitions. Deduction remains the same for equations, but the symmetry rule is dropped for rewrite rules. Models of rewrite theories are \((\Sigma, E)\)-algebras enriched with transitions satisfying all the rewrite rules in \(R\). In our context, the equational part of a rewrite theory is allowed to “underspecify” features as far as the specification remains executable. Interestingly, rewrite theories also have initial models, consisting of term models factored by the equational derivability relation and enriched with appropriate transitions between the equational equivalence classes. They also follow the slogan “no junk, no confusion”, but extend it also w.r.t. reachability of terms via transitions. Rewriting logic is a framework for true concurrency. The reader interested in details is referred to [32]. We here only discuss this by means of examples.

Suppose that \((\Sigma, E, R)\) is the following rewrite theory:

\[
\Sigma: \text{sort State} \\
\theta, 0, 1 : \rightarrow \text{State} \\
_\ast : \text{State} \times \text{State} \rightarrow \text{State} \\
E: (\forall S : \text{State}) 0 S = S \\
(\forall S_1, S_2 : \text{State}) S_1 S_2 = S_2 S_1 \\
(\forall S_1, S_2, S_3 : \text{State}) (S_1 S_2) S_3 = S_1 (S_2 S_3) \\
R: \text{rule } r_1 : 0 \Rightarrow 1 \\
\text{rule } r_2 : 1 \Rightarrow 0
\]

The two equations state the associativity and commutativity of the binary \(_\ast\) “operator, thus making it a multi-set operator, and the two rules flip the two constants 0 and 1. If one starts with an initial term as a multi-set containing 0 and 1 constants, then the rules \(r_1\) and \(r_2\) can apply concurrently. For example, if the initial term is the multi-set \(0 1 0 1 0\) then three instances of \(r_1\) and two of \(r_2\) can apply in parallel and transform the multi-set into \(1 0 1 0 1\). Note, however, that there is no requirement on the number of rewrites applied concurrently; for example, one can apply only the instances of \(r_1\), or only one instance of \(r_1\) and one of \(r_2\), etc. Consequently, on a multi-set of 0 and 1 constants, the rewrite theory above manifests all possible concurrent behaviors, not only those following an interleaving semantics. And indeed, if one “executes” this specification on a machine with an arbitrarily large number of processors, then one can observe any of these concurrent behaviors. Parallel execution of rewrite logic specifications is an important aspect of our language definitions (see [42]).

One can regard a rewrite logic specification as a compact means to encode transition systems, namely one that has the capability to generate for any given term a transition system manifesting all its “concurrent” behaviors. The states of that transition system are the terms to which the original term can evolve by iterative applications of rewrite rules; the equations are used to keep the states in canonical forms (in this case as AC multi-sets, but in general can be any terms which are not reducible by applying equations from left to right - modulo particular axioms, such as AC) and the rules are used to generate transitions (in this case the rules are not parametric, but in general they can have variables).

Given a rewrite logic specification and an initial term, the corresponding transition system may or may not need to be generated explicitly. For example, if one is interested in one execution of the specification on that term, than one only needs to generate one path in the transition system. If one is interested in testing whether a particular term can be reached (reachability analysis), then one can only generate the transition system by need, for example following a breadth-first strategy. However, if one is interested in checking some complex property against all possible executions (model-checking) then one may need to generate the entire transition system. Interestingly, one can regard a concurrent programming language also as a means to encode transition systems, namely one taking a program and, depending upon the intended purpose, generate one path, part of, or the entire transition system comprising all behaviors of that program. Thus, that programming languages can be given a rewriting logic semantics should come at no surprise. What may seem surprising in the sequel is the simplicity of such language definitions when one uses the K framework.

All rewrite engines, by their nature, generate one (finite or infinite) path in the transition system of a term when requested to reduce that term. Therefore, we can use any of these rewrite engines to execute K specifications, converting them into interpreters for the languages defined in K. The Maude system also supports breadth-first exploration of the state space of the transition system of term, as well as linear temporal logic (LTL) model checking. Using these features one can, for example, show that in the example above it is possible to start with the state of zeros 0 0 0 0 0 and reach a state of just ones; also, using the model checker one can show that it is not the case that whenever one reaches the state of zeros then one will eventually reach a state of ones. Indeed, there are infinite executions in which one can reach the state of zeros and then never reach the state of ones. The report [42] shows how these formal analyses can be performed in Maude. As one may expect, this capability of rewrite logic systems to explore the state space of the transition system associated to a term will allow us to obtain corresponding analysis tools for the programming languages that we will define as rewrite logic theories in K.

If rewrite rules can apply concurrently and in as many instances as the term to rewrite permits, then how can one attain synchronous applications of rules? How can one simulate situations in which one wants that each application of a rule is an atomic action and thus happen only one at a time? As usual, this can be achieved by introducing “synchronization objects” and making sure that each rule intended to synchronize grabs the synchronization object. In the example above, we can, e.g., introduce a new constant \(S : \rightarrow \text{State} \) and replace the two rewrite rules by

\[
\text{rule } r_1^S : 0, S \Rightarrow 1, S \\
\text{rule } r_2^S : 1, S \Rightarrow 0, S
\]
and make sure that the multi-set to rewrite contains precisely one constant $\$. Rewrite rules can apply in parallel on a term only if that term can be matched a multi-context with one hole per rule application, the subterm corresponding to each hole further matching the left-hand-side (lhs) of the corresponding rule. In particular, that means that rule instances whose lhs’s overlap cannot be applied in parallel. Since the rules above overlap on the “synchronization object” $\$, they can never be applied concurrently. Note that the equations are being applied “silently” in the background, to permute the constants in a way that rules can apply. Indeed, the role of equations is to generate equivalence classes on which rewrite rules can apply and thus transit to other equivalence classes, etc. Rewrite engines provide heuristics to choose a “good representative” of each equivalence class, typically by canonicalizing it applying the equations as rewrite rules (from left to right), potentially modulo associativity and/or commutativity.

Therefore, the underlying execution engine has the possibility to nondeterministically pick some rule application and thus disable the applications of other rules that happen to overlap it. In practice, rewrite logic theories contain both synchronous and asynchronous rules. In particular, our language definitions will contain asynchronous rules for thread local computations and synchronous rules for thread interactions; for example, reading/writing shared variables is achieved with rules that synchronize on the store (mapping locations to values). Thus, if one executes a concurrent program on a multi-processor rewrite engine in its programming language rewrite logic definition, one can obtain any of the possible (intended or unintended) concurrent behaviors of that program.

Together with concurrency and synchronization, the problem of deadlock is almost unavoidable. Let us next show how deadlocking can appear in a rewrite logic context by means of a classical example, the dining philosophers. We can encode all the philosophers and the forks as elements in a set called “state”, define equations to keep the state in a canonical form and to capture actions that need not split the state space, such as releasing forks, and define rules to capture those actions that split the state space, in our case the operation of acquiring a fork. Let us next define one possible rewrite logic theory specifying the dining philosophers problem and show by reachability analysis that it can, indeed, lead to a deadlock. We start by defining a “state” sort $\text{State}$ as a multi-set as above. When specified formally in rewrite logic systems, one may want to replace the three standard ACI equations above by operation attributes. We next add the two important constructors for states, philosophers and forks. We identify philosophers and forks by natural numbers, assuming built-in numbers and integers. Note that a philosopher can keep some part of the state, namely a set of philosophers; we will make sure, by corresponding equations and rules, that philosophers will hold at most two forks:

operation $p : \text{Nat} \times \text{State} \rightarrow \text{State}$
operation $f : \text{Nat} \rightarrow \text{State}$

The following (uninteresting) operations and equations declare an initial state, in this case of 10 philosophers:

operation $n : \rightarrow \text{Nat}$
equation $n = 9$
operation $\text{init} : \rightarrow \text{State}$
operation $\text{init} : \text{Nat} \rightarrow \text{State}$
equation $\text{init} = \text{init}(n)$
equation $\text{init}(1) = 0$
operation $(\forall N : \text{Nat}) \text{init}(N) = p(N, \emptyset) f(N) \text{init}(N - 1)$

We are now ready to give the three rules defining the philosophers’ actions, namely grabbing one of their neighbor forks:

rule $(\forall N : \text{Nat}, Fs : \text{State}) p(N, Fs) f(N) \Rightarrow p(N, Fs f(N))$
rule $(\forall N : \text{Nat}, Fs : \text{State}) p(s(N), Fs) f(N) \Rightarrow p(s(N), Fs f(N))$
rule $(\forall N : \text{Nat}, Fs : \text{State}) p(0, Fs) f(N) \Rightarrow p(0, Fs f(n))$ if $N = n$

Assuming the actions of eating and of releasing both forks are local actions without involving any “competition” among the philosophers (these would happen anyway, regardless of the external environment), we can capture both these actions with just one equation:

$$\text{equation } (\forall N, X : \text{Nat}) p(N, f(X)) f(Y) = p(N, \emptyset) f(X) f(Y).$$

One can now use the rewrite logic theory above to generate a transition system comprising all the behaviors that can result from its initial state, the term $\text{init}$. That transition system will have equational equivalence classes as states and instances of the three rules above as transitions. It is easy to see that that transition system encodes indeed all the behaviors of the dining philosophers’ problem, and also that it has a finite number of states. Using generic formal analysis tools for rewrite logic specifications, such as reachability analysis, one can show that there are precisely two scenarios in which the concurrent system above deadlocks. All one needs to show is that one can reach a state in which no rule can be applied anymore. The report [42] shows how such an analysis can be performed in Maude. For 10 philosophers, for example, Maude takes 2.7 seconds on a 2.5GHz/3.5GB to explore the entire state space of 15,127 states and find the two deadlock solutions. For 14 philosophers it takes Maude about 400 seconds to explore all the 710,647 states (Maude crashed when tried on 15 philosophers).

### B. Translation to Maude

To translate a rule from K into MAUDE, the context for the rule is transformed as necessary to make it a valid MAUDE term, and sorts of variables are inferred, with necessary variable declarations then added. As an example of this process, we show the transformation for Rule 32, from the Concurrent KOOL semantics given in Section 3.3. The rule, in $K$, is the following:

$$\text{if notin(LS,lk(V)) \cdot \text{holds}(\cdot \text{busy}(LS) \Leftrightarrow V \notin LS \text{ (V, T)} \text{ LS} \cdot \text{V}}$$

The equivalent rule in MAUDE is:

```maude
var K : Continuation . var CS : Control .
var LTS : LockTupleSet .
var V : Value . var LS : LockSet . var TS : Thread .
crl t(control(k(val(V) \rightarrow acquire \rightarrow K) CS)
holds(LTS) TS) busy(LS) \Rightarrow
\text{t(control}(k(k(CS)
holds(LTS [lk(V),1])TS) busy(LS 1k(V))
if notin(LS,1k(V)))) .
```

In **MAUDE** values (as well as statements, expressions, etc) are wrapped before being put on the continuation to assist parsing. Since the $K$ rule is boxed, this is a rewrite rule, and since the $K$ rule has a condition, the **MAUDE** rule does as well. This is indicated by $\text{crl}$, for conditional rewrite rule. The context of the $K$ rule has been transformed to include the remainder of the referenced sets, including unmentioned parts of the control, the thread, the lock set, and the lock tuple set used by $\text{holds}$.