Partial Order Reduction for Rewriting Semantics of Programming Languages

Azadeh Farzan and José Meseguer
Department of Computer Science,
University of Illinois at Urbana-Champaign.
{afarzan,meseguer}@cs.uiuc.edu

Abstract. Partial order reduction (POR) capabilities are typically added by extending a model checking algorithm supporting analysis of programs in a given programming language. In this paper we propose a *generic* method to generate a model checker with POR capabilities for any programming language of interest. The method is based on giving a formal executable specification of the semantics of a programming language L as a rewrite theory $\mathcal{R}_L$, and then exploiting the efficient execution, search, and LTL model checking capabilities of the Maude rewriting logic language to generate a model checker for L essentially for free. The key idea is to achieve the desired POR reduction by means of a *theory transformation* that transforms the theory $\mathcal{R}_L$ into a semantically equivalent theory which is then used to explore the POR-reduced state space. This can be done for a language L with relatively little effort and has the advantage of not requiring any changes in the underlying model checker. Our experiments with the JVM and with a Promela-like language indicate that significant state space reductions and time speedups can be gained for the tools generated this way.

1 Introduction

Developing formal analysis tools for a programming language is a labor-intensive process which often requires man-years of effort. Furthermore, such tools, being by design language-specific, may not be easy to reuse for other languages, and may even require substantial re-engineering for new versions of the same language. Little mathematical confidence can be placed on such tools when they are based on the low-level coding of the given language's implementation; and it is then unfeasible, by lack of a formal semantics, to combine model checking and theorem proving in a rigorous manner. Therefore, the possibility of developing generic, language-independent software analysis tools seems attractive. This work proposes a generic method that can be used to obtain a linear time temporal logic (LTL) model checker with *partial order reduction* (POR) capabilities for any concurrent programming language L of interest with minimal requirements on L, and with relatively little effort: in our experience with substantial languages like Java and the JVM in a matter of a few weeks (as opposed to man-years) for the entire effort, including developing the formal semantics of L.

Our method is *semantics-based*, in the sense that it takes a formal semantic definition of the concurrent programming language L as input and then yields a POR-enabled LTL model checker for L as the result. This allows reasoning about the correctness of the tool thus derived, and supports a seamless integration between model checking and theorem proving for L, because both tasks and their combinations are based on the formal semantics of L. Therefore, this work advances a broader research program in collaboration with several colleagues at UIUC [17, 8, 7], in which we are investigating *generic methodologies* to develop software analysis tools for a wide range of concurrent programming languages based on the *formal semantic definition* of the language of interest in rewriting logic [16].

Adding POR capabilities to a model checker for a language L typically requires nontrivial extensions and modifications to the model checker. In contrast, the generic method proposed in this paper is based on a *theory transformation* $\mathcal{R}_L \mapsto \mathcal{R}_{L+POR}$ in which the original rewrite theory $\mathcal{R}_L$ specifying the semantics of L is transformed into a semantically equivalent rewrite theory $\mathcal{R}_{L+POR}$ that accomplishes the desired partial order reduction when used for model checking a given program. This theory transformation approach means that *no changes to the underlying model checker are needed* to achieve the desired partial order reduction, which is one of the reasons why developing a POR-enabled LTL model checker for a language L using our method requires such little effort.

Besides its genericity, language-idependence, and short development time, our method has two additional advantages:

Flexible Partial Order Heuristic Algorithm. The heuristic algorithm can be specified using a few equations. Although our basic version of the heuristic can in theory work for any programming language, additional modifications, based on specific knowledge of the given programming language or the types of programs to be verified, could make the POR reduction considerably more efficient. The tool builder can easily modify the heuristic algorithm, which compares favorably with having to change the source code of a model checker.

Flexible Dependence Relation. Although a basic dependence relation can generally hold for a certain programming language, additional knowledge of the types of programs that one needs to verify can result in removing some dependencies; for example, Java supports shared memory in general, so we have to assume that memory read/write pairs are generally interdependent; but if the programs being verified do not use the shared memory at all, we can remove this dependency for such programs. Having the dependence relation as an explicit input to the partial order reduction module not only contributes to the generality of the method, but also gives the tool builder the advantage of modifying it, based on the type of input programs.

The general point illustrated by the above flexible customization features is that our theory transformation assumes a simple interface of functionality in the language L and allows a first automatic transformation of the theory $\mathcal{R}_L$; but this does not preclude a second *language-specific customization* phase, which can be easily accomplished adding or customizing a few equations, in which the detailed knowlege of L's semantics can be used to improve the reduction; for example, by improving the heuristic algorithm and/or defining a more precise dependence relation using static analysis techniques as mentioned above.

Besides developing its theoretical foundations and establishing its correctness, the practical usefulness of a generic method like the one we propose should be evaluated experimentally. Therefore, we have developed a prototype tool in Maude that, given an original semantics of a language L specified as a rewrite theory $\mathcal{R}_L$, performs the theory transformation $\mathcal{R}_L \mapsto \mathcal{R}_{L+POR}$ and can be used to model check LTL properties of programs in L using Maude's generic LTL model checker. We have applied this prototype to the rewriting semantics of the Java bytecode and of a simple Promela-like language and have evaluated the performance of our POR methods for both languages using several benchmarks. The goal of this prototype and experimentation is a *proof-of-concept* one. Therefore, we have not incorporated a number of well-known optimizations that a mature tool should support. Nevertheless, our experiemts indicate that, even without such optimizations, substantial gains in time and space can be obtained using our POR method.

Related Work. There are two well-known approaches to attack the state-explosion problem while model checking. The first approach consists of *partial order methods* introduced by Peled in [18]. The generic method proposed in this paper fits within this approach. Several different variations [11, 12, 21, 1, 9, 3, 15] of the POR approach have been introduced since.

A first class of POR methods —including the stubborn sets method of [21], the persistent sets method of [13], and the ample sets method of [19]— are based on modifying the search algorithm and applying the reduction dynamically. [9] takes the matter even further, and dynamically tracks the interactions between threads based on initially exploring an arbitrary interleaving of them. Details of the reduction heuristic are orthogonal to our method; although we propose two different heuristics in this paper, many other heuristics can be implemented with little effort. A second class of POR methods such as the one in [15] use a static approach in which all partial order reduction information is computed statically, and then an already reduced model is generated to be model checked.

In the dynamic methods, one has to alter the existing model checker to include the reduction, while static methods suffer from the fact that only a limited amount of information is available at compile time. We believe that our method addresses both problems: it can work with an existing model checker, so it has the advantages of the static methods, but it applies the reduction dynamically and therefore can benefit from the runtime information.

It seems fair to say that current POR-enabled model checkers are mostly language-specific, or, by using for example a static approach such as [15], achieve only a limited "genericity by translation into a common intermediate language". Verisoft [12] provides language-independence by mapping program processes to UNIX processes and monitoring the system calls of these UNIX processes. This approach involves wrapping of the system code which is a nontrivial task for some programming languages such as Java. Therefore, Verisoft is used for a limited family of languages in practice and also cannot benefit from any potential optimizations that may use information provided by static analysis of the program. By contrast, our approach semantic-based approach provides the language-independence without these problems. Moreover, Verisoft is restricted to model checking the concurrent systems with an acyclic state space and consequently merely checking safety properties, while we can accept arbitrary state spaces and model check any LTL_{-X} property.

Besides the POR methods, a second state space reduction approach, which could be called *transaction-based*, consists of more recent techniques that consider various kinds of *exclusive access predicates* for shared variables specifying some synchronization disciplines [20, 10, 5]. These predicates can be used to reduce the search space during the state space explorations. The POR techniques (including the method proposed in this paper) are complementary to these other methods. We discuss how our method exploits some ideas from [20] in Section 2.1. We strongly believe that the reductions in [10] can be achieved using a very similar method to that presented in this paper (see Section 5 for more details).

The rest of the paper is organized as follows: Section 2 contains the background knowledge needed in Section 3, where we discuss the generic method in details; Section 4 presents the experimental results including the instantiation of the method for the Java bytecode and for a Promela-like language, as well as presenting some performance figures; and Section 5 includes the conclusions and future directions.

2 Preliminaries

2.1 Rewriting Logic Language Specification

The rewriting logic semantics of a programming language [17] combines and extends both equational/denotational semantics based on *semantic equations*, and structural operational semantics (SOS) based on *semantic rules*. Given a programming language L, its rewriting logic semantics is defined as a rewrite theory $\mathcal{R}_L = (\Sigma_L, E_L, R_L)$, with Σ_L a signature specifying both the syntax of L and of operations on auxiliary semantic entities like the store, environment, and so on, with (Σ_L, E_L) an equational theory specifying the semantics of the sequential features of L, and with R_L a collection of (possibly conditional) rewrite rules specifying the semantics of L's concurrent features. Under the assumption that $\mathcal{R}_L$ is coherent [22], equations in E_L (corresponding to execution of sequential features) are applied until reaching a canonical form, and then rules in R_L (corresponding to execution of concurrent features) are applied. This key distinction between equations and rules immediately gives the advantage of reductions similar to those in [20]. The *invisible states* in [20] are closely related to the reduction steps done by equations in E_L. Only when no more equations from E_L apply to the state, does a rewrite with a rule in R_L take place. This makes any sequence of *sequential* instructions in a thread to be executed as an *atomic* block, without any interleavings. Note that this kind of state reduction is available at the level of the original semantics $\mathcal{R}_L$; what is now further needed, which is the topic of this paper, is to achieve an *additional* POR state space reduction by reducing the state explosion due to the execution of *concurrent* features.

Specifying formally the semantics of a concurrent programming language L in the Maude rewriting logic language, not only yields a language interpreter for free, but also, thanks to the generic analysis tools for rewriting logic specifications that are provided as part of the Maude system [4], additional analysis tools are also automatically provided for L, including a semi-decision procedure to find failures of safety properties, and an LTL model checker. There is already a substantial experience on the practical use of such language definitions and the associated analysis tools for real languages such as Java, the JVM, and a substantial subset of OCaml [17, 8, 7].

2.2 Background on Partial Order Reduction

A *finite transition system* is a tuple (S, S_0, T, AP, L), where S is a finite set of states, $S_0 \subseteq S$ is the set of initial states, T is a finite set of transitions such that $\alpha \in T$ is a partial function $\alpha : S \to S$, AP is a finite set of propositions and $L : S \to 2^{AP}$ is the labeling function. A transition α is *enabled* in a state S if $\alpha(s)$ is defined. Denote by $\text{enabled}(s)$ the set of transitions enabled in s. The main goal of partial order reductions is to find a subset of enabled transitions $\text{ample}(s) \subseteq \text{enabled}(s)$ that is used to construct a reduced state space that is behaviorally equivalent. Partial order reduction is based on several observations about the nature of concurrent computations. The first observation is that concurrent transitions are often commutative, which is expressed in terms of an *independence relation*, $I \subseteq T \times T$, that is, a symmetric and antireflexive relation which satisfies the following condition: for each $(\alpha, \beta) \in I$, and for each state s, if $\alpha, \beta \in \text{enabled}(s)$ then (1) $\alpha \in \text{enabled}(\beta(s))$ and $\beta \in \text{enabled}(\alpha(s))$, and (2) $\alpha(\beta(s)) = \beta(\alpha(s))$. Note that $D = (T \times T)\backslash I$ is the *dependence* relation. The second observation is that in many cases only a few transitions can change the value of the propositions, which suggests the concept of *visibility*; a transition $\alpha \in T$ is *invisible* if for each $s \in S$, if $s' = \alpha(s)$ then we have $L(s) = L(s')$.

There are several existing heuristics to compute $\text{ample}(s)$. [2] gives a set of four conditions that, if satisfied by $\text{ample}(s)$, guarantee a correct reduction of the given state transition system. In Section 3.3, we present a special case of the conditions in [2] which are used in this paper.

3 Partial Order Reduction for Language Definitions

3.1 Some Assumptions

In order to devise a general partial order reduction module for semantic definitions of concurrent programming languages, we have to make some basic assumptions about these semantic definitions. These assumptions are quite reasonable and do not limit in practice the class of semantic definitions that we can deal with. They simply specify a standard interface between the semantic definition module and the partial order reduction module. We can enumerate these assumptions as follows: (1) In each program there are entities equivalent to threads which can be uniquely identified by a *thread identifier*. The computation is performed as the combination of local computations inside individual threads, and communication between these threads through any possible discipline such as shared memory, synchronous or asynchronous message passing, and so on. (2) In any computation step (transition) a single thread is always involved. In other words, threads are the entities that carry out the computations in the system. (3) Each thread has *at most* one transition enabled at any moment.

3.2 The Theory Transformation

The rewrite theory $\mathcal{R}_L = (\Sigma_L, E_L, R_L)$ specifying the semantics of a concurrent programming language L is transformed in *two steps* into the semantically equivalent theory $\mathcal{R}_{L+POR} = (\Sigma_{L+POR}, E_{L+POR}, R_{L+POR})$ that is equipped with partial order reduction capabilities.

The Marked-State Theory. The objective of the first step of this transformation is to change the original theory $\mathcal{R}$ in order to facilitate the addition of the partial order module. In the transformed theory $\widehat{\mathcal{R}}_L = (\widehat{\Sigma}_L, \widehat{E}_L, \widehat{R}_L)$: (1) the rewrite rules of R_L are changed syntactically to only allow one-step rewrites, and (2) the structure of the states of $\mathcal{R}$ is enriched to allow a specific thread to be marked as *enabled*. Rewrite rules are then modified to only allow the threads that are marked *enabled* to make a transition. This way, when the POR heuristic decides on an ample set, the corresponding threads can be marked as *enabled*, and this causes only the ample transitions to be explored next. Here we give a detailed construction of $\widehat{\mathcal{R}}_L$ and show that $\mathcal{R}_L$ and $\widehat{\mathcal{R}}_L$ are one-step bisimilar.

We assume that $\mathcal{R}_L$ is coherent [22] and that all rules in $\mathcal{R}_L$ are of the form $l(u(t)) \longrightarrow r(u'(t))$ where terms l and r are of sort *State*, and where the subterms $u(t)$ and $u'(t)$ are thread expressions of sort *Thread*, and t is variable ranging over thread identifiers of sort *Tid*. Note that based on assumptions we made (see Section 3.1), there is going to be exactly one such thread expression $u(t)$ on either side of a rule. We also assume that the equations in E_L are *thread-preserving*, that is, in any any two state expressions equated by E_L both must have the same number of thread expressions and there is a bijective correspondence between such thread expressions preserving their thread identifiers.

We define $\widehat{\Sigma}_L$ by adding fresh new sorts: *MState* and *MThread*. A new constructor *enabled* : *Thread Bool* $\longrightarrow$ *MThread* is introduced for the sort *MThread* to instrument threads with this additional flag that allow us to mark them as *enabled* or not for the next execution step. The use of the sort *Thread* in all state constructors is everywhere replaced by the sort *MThread*. We also add two unary operators $\{\_\}, [\_]$: *State* $\longrightarrow$ *MState*. The equations in $\widehat{E}_L$ are systematically derived from those in E_L by replacing in each equation in E_L each occurrence of a thread expression $u(t)$ by the expression $\mathit{enabled}(u(t), b_t)$, where b_t is a fresh new variable of sort *Bool* depending on t. For every rewrite rule $l(u(t)) \to r(u'(t))\ \mathit{if}\ C$ in R_L, the corresponding rewrite rule in $\widehat{R}_L$ is then of the form $\{Ct(l(\mathit{enabled}(u(t), \mathit{true}))\} \to [Ct(r(\mathit{enabled}(u'(t), \mathit{true})))]\ \mathit{if}\ \widehat{C}$, where $Ct(.)$ is the context expression for the application of the rule in case r does not rewrite the entire state but only a state fragment[1], and where $\widehat{C}$ is the conjunction of equations obtained from C by changing each equation in C containing thread expressions as done in the definition of $\widehat{E}_L$, and leaving all other equations untouched. Note that the use of the operators $\{\_\}, [\_]$ in the rules in $\widehat{R}_L$ means that in $\widehat{\mathcal{R}}_L$ *only one-step rewrites are possible*, since the operator $[\_]$ in the right-hand side blocks the application of any further rules.

As an example of the above transformation, consider the following rewrite rule specifying the semantics of the `monitorenter` instruction of Java bytecode:

[1] If the rule r rewrites the global state of the computation, the context $Ct(.)$ is empty, i.e. $Ct(l(\boldsymbol{u})) = l(\boldsymbol{u})$. We do however allow language specifications in which a rule r can be local to some fragment of the state. In this second case, it is important to make explicit a pattern $Ct(.)$ for the context in which the rule is applied.

```
rl < T: JavaThread | callStack:([PC, monitorenter, Pgm, ..., (REF(K) # OperandStack), ...]
   CallStack), ... > < O : JavaObject | Addr: K, ..., Lock: Lock(OIL, NoThread, 0) >
=> < T: JavaThread | callStack: ([PC + 2, Pgm(PC + 2), Pgm, ..., OperandStack, ...]
   CallStack), ... > < O: JavaObject | Addr: K, ..., Lock: Lock(OIL, T, 1) > .
```

the transformed rewrite rule has the following form:

```
rl { enabled( < T:JavaThread | callStack:([PC, monitorenter, Pgm, ..., (REF(K) # OperandStack),
 ...] CallStack), ... >, true) < O:JavaObject | Addr:K, ..., Lock:Lock(OIL, NoThread, 0) > Ct }
=> [ enabled(< T: JavaThread | callStack: ([PC + 2, Pgm(PC + 2), Pgm, ..., OperandStack, ...]
   CallStack), ... >, true) < O: JavaObject | Addr: K, ..., Lock: Lock(OIL, T, 1) > ] .
```

The key point about the transformation $\mathcal{R}_L \mapsto \widehat{\mathcal{R}}_L$ is then:

Proposition 1. *The surjective projection π mapping terms of sort* MState *to terms of sort* State *defined by: (1) erasing the operators $\{\_\}, [\_]$, and (2) erasing the* enabled *operators, the corresponding flags and the context expression defines a* one-step bisimulation *between the corresponding rewrite theories.*

That is, if we have a one-step rewrite $u \to v$ with $\widehat{\mathcal{R}}_L$, then we have also a corresponding one-step rewrite $\pi(u) \to \pi(v)$ with $\mathcal{R}_L$; and conversely, if we have a one-step rewrite $u' \to v'$ with $\mathcal{R}_L$, then we can find $u \in \pi^{-1}(u')$ $v \in \pi^{-1}(v')$ such that we have a one-step rewrite $u \to v$ with $\widehat{\mathcal{R}}_L$ (see Appendix A for proof).

The Partial Order Reduction Theory. In the second step, the theory $\widehat{\mathcal{R}}_L = (\widehat{\Sigma}_L, \widehat{E}_L, \widehat{R}_L)$ is transformed into $\mathcal{R}_{L+POR} = (\Sigma_{L+POR}, E_{L+POR}, R_{L+POR})$ which adds to $\widehat{\mathcal{R}}_L$ the partial order reduction module. Components of the transformed theory are defined based on the components of $\widehat{\mathcal{R}}_L$ as follows:

- $\Sigma_{L+POR} = \widehat{\Sigma}_L \cup \Sigma_{POR} \cup \Sigma_{AUX}$, that is, the signature $\widehat{\Sigma}_L$ is extended with the signature Σ_{POR} of operators used in implementing the partial order heuristic algorithm, plus the signature of auxiliary operators Σ_{AUX} that are used for implementation purposes.
- $E_{L+POR} = \widehat{E}_L \cup E_{POR} \cup E_{AUX}$, that is, the set of equations $\widehat{E}_L$ are extended with the equations E_{POR} which specify the partial order heuristic algorithm, plus the equations E_{AUX} which define the auxiliary operators.
- $R_{L+POR} = \widehat{R}_L \cup \{r_{step}\}$. In the case of the rewrite rules, only one new rewrite rule is added. We label this rule as *step*. It is the only rule applicable to the new state, and therefore the only rule which will determine the transitions of the system at a given state.

The New State. There is a new fresh sort *PState*, as part of Σ_{POR}, representing the new state of the system. A new sort *StateInfoSet* also belongs to Σ_{POR}, capturing all the information necessary for the reduction algorithm (see Section 3.3). A new constructor operator $\{\_|\_\}$: *MState StateInfoSet* $\longrightarrow$ *PState* is introduced for the new state. Therefore, a state in $\mathcal{R}_{POR}$ is a pair $\{s|I\}$, where s is a state in $\widehat{\mathcal{R}}_L$, and I is a term containing information necessary for the reduction algorithm.

The New Rule (step). A single new conditional rule r_{step} in R_{L+POR} simulates one step rewrites of the original system:

$$\text{step} : \{s|I\} \to [s'|I] \;\; \text{if } s \to s' \wedge s \neq s'$$

where s and s' are variables of sort *MState*, and the operators $\{\_|\_\}$ and $[\_|\_]$ are state constructors for the sort *PState* and are *frozen* operators [4], that is, no rewriting is allowed below these operators. I is a variable of sort *StateInfoSet*. By using this single rewrite rule, only one rewrite at a time can happen, which changes the given state to one of its successor states. Since the resulting state is in [_|_] format, no rewrite rule is applicable to it anymore, until it is changed to the {_|_} format. This is the point at which the partial order heuristic algorithm is applied, using an equation that completes the effect of the above rule:

$$[s \mid I] = \{\mathit{state}(\mathit{MarkAmples}(s, I)) \mid \mathit{stateInfo}(\mathit{MarkAmples}(s, I))\}. \qquad (*)$$

The partial order reduction is applied at state s, using the information in I, by means of a single operation *MarkAmples*. This operation takes a pair of elements of sorts *MState* and *StateInfoSet* as an input, and returns a pair of the same sort. The *MarkAmples* operation computes the ample set for the current state and returns the state with the ample transitions marked as specified by the POR algorithm. It also returns an updated version of *StateInfoSet* (see the POR algorithm part of Section 3.3). In the next section, we discuss in detail how the *MarkAmples* operations is specified.

3.3 The Partial Order Reduction Module

This module performs two main tasks: (1) extracting the set of enabled transitions at a given state, and (2) finding an ample subset of these transitions.

First, we have to define a *transition* in this context. Having the rewriting semantics (Σ_L, E_L, R_L) of a concurrent programming language L, one can view the initial state of the system (a program and its inputs) as a Σ_L-term t being rewritten by the equations E_L and the rewrite rules R_L of the specification.

In a state transition system, a given state s has a set of immediate successor states $\{s_1, s_2, \ldots, s_k\}$, and each pair (s, s_i) is an enabled transition from state s. In the rewriting semantics, state s is a term, and the set of enabled transitions leading to successor states can be represented as a set of pairs (r_i, p_j), where $r_i \in R_L$ and p_j is a position in term s. In other words, if a certain rule $r_i : l(\boldsymbol{u}) \to r(\boldsymbol{v})$ is enabled at a position p_j in term s, then we have a transition from s to its successor $s[l(\boldsymbol{u}) \backslash r(\boldsymbol{v})]$.

In general a position p can be any position in the term *tree*. However, in our special case of semantics of concurrent programming languages together with the general assumptions discussed in Section 3.1, *a thread identifier will uniquely specify a position*, since we have assumed that a single thread is involved in each rewrite, and that each thread has at most one transition enabled at a time. Therefore, a pair (t_i, r_j) consisting of a thread identifier t_i together with an applicable rule r_j uniquely characterizes a transition. This gives us a considerable practical advantage; because when the algorithm decides on an *ample* subset of the transitions, it suffices to mark the corresponding threads as enabled (see Section 3.1), which makes it unnecessary for all the unmarked threads (transitions) to be explored. Note that in the transformed theory, although the only rule applied to the state of the system is the rule *step*, in fact an application of *step* always simulates some rewrite rule r_i from the original system, and it is that rule that we consider in the above pair.

Extracting Enabled Transitions

As discussed above, a transition is a pair (t_i, r_i) of a thread identifier and a rewrite rule. We can add a third component I_k to this tuple, which includes all the information about context (i.e., names of variables, functions, locks, ...). This information can later help resolving some dependencies between the transitions, which may result in fewer dependencies and possibly in a better reduction.

At a given state s, we have to find all pairs $(t_i, r_j : l(\boldsymbol{u}) \rightarrow r(\boldsymbol{v}))$ where the rewrite rule r_i is enabled for the term s at the position associated with the thread t_i. In other words, we have to go over all the rewrite rules $r_i \in R_L$ and find all the positions at which r_i can be applied to the term s. To do this, we generate a new set of equations, based on the rewrite rules in R_L, with exactly one equation per rule in the following manner. Let us assume that a rewrite rule $r \in \widehat{R}_L$ is of the following general form:

$$r : \{l(u(t))\} => [r(u'(t))] \text{ if } C$$

where $u(t)$ and $u'(t)$ are subterms of sort *Thread*, t is a variable of sort *Tid*, and C is the rule's condition. The corresponding equation for r is then:

$$\langle T_e, l(u(t))\rangle = \langle T_e \cup \{< t, r, I >\}, l(u(t))\rangle \text{ if } C \wedge T_e \cup \{< t, r, I >\} \neq T_e$$

where T_e is a set that accumulates enabled transitions. Note that rewrite rules in $\widehat{R}_L$ are already modified to capture the context in which the corresponding original rule of R_L would have been applied. Starting from the pair $< \emptyset, t_s >$, by applying all equations of the above form, we will converge to the pair $< T_e, t_s >$, where T_e is the set of all enabled transitions.

Since the context information I depends on the specific programming language L and on the way the semantics of L is defined, the I component has to be left as a null constant when these equations are generated automatically based on the rules. However, a tool builder familiar with the language semantics can customize these equations to include whatever context information may be useful later. In our experience with several rewriting semantics for different programming languages, there are relatively few rewrite rules in the semantic definitions (that is, E_L is much bigger than R_L), so this process is rather quick and easy.

Computing the Ample Set

Dependence Relation. The Definition of a *dependence* relation between the transitions is required for computing the ample sets. The dependence relation is represented by the operator *Dependence: Transition Transition* $\longrightarrow$ *Bool*. Clearly, the dependence relation is different for different programming languages. Some common dependence properties can be shared by many programming languages, such as: "all the transitions in a single thread are interdependent", which is expressed by the following equation:

$$\text{Dependence}(< t, r, I >, < t, r', I' >) = \text{true}$$

where t is a variable ranging over thread identifiers, r and r' are variables ranging over rule names, and I and I' are variables ranging over context information.

In order to have the best possible reduction, the language specifier/tool builder should supply the definition of the dependence relation for the given language as a set of additional equations. The dependence relation can often be defined through a few equations, even for complicated languages. See Section 4 for the definition of the dependence relation for the Java bytecode. Note that, in general, since the dependence relation is defined by a set of equations (that can potentially be conditional) we can naturally support the case of *conditional dependence* as in [5, 14].

The Heuristic Algorithm. Since the core of the heuristic algorithm can be specified using a few equations, we have specified two different heuristics. Many additional optimizations for these heuristics and also other heuristics can likewise be specified with little effort (see Section 5), but they are beyond the scope of this work. Figure 1 shows both algorithms. Functions C_1', C_2, and C_3 check the three conditions discussed in the next Section, returning *true* or *false*.

$T_{e,s}$: enabled transitions in state s.
$\mu_{c_{D,S}}$: transitive closure of the dependence relation.
S: set of transitions.
P: set of predicates of the LTL formula to be model checked.

```
1 Take a transition $t$ from $T_{e,s}$.
2 If $C_1'(t)$ and $C_2(t,P)$ and $C_3(t)$.
3 then
        mark thread of $t$ as ample.
        quit.
4 else
        go to step 1.
5 Mark all threads as ample.
```

```
1 Take a transition $t$ from $T_{e,s}$.
2 Let $S = \mu_{c_{D,T_{e,s}}}(t)$.
3 If $C_1'(S)$ and $C_2(S,P)$ and $C_3(S)$.
4 then
        mark thread of $t$ as ample.
        quit.
5 else
        go to step 1.
6 Mark all threads as ample.
```

Fig. 1. Two Partial Order Reduction Heuristics.

These procedures are called at each state (see Section 3.2) to compute the ample set at that state. The algorithm on the left is a simpler version, which only considers ample sets of cardinality one (one transition). The algorithm on the right extends the former to consider sets of any cardinality, which can result in a better reduction. If we have n threads, and at some point no single thread can be a candidate for ample, we may be able to find a subset of threads that can satisfy the conditions as a whole. To do so, we use the transitive closure of the dependence relation D defined on the set $\mathcal{T}$ of transitions as follows:

$$\begin{array}{ll} D : \mathcal{T}^2 \to \{\text{true}, \text{false}\} & S, T \subseteq \mathcal{T}, t \in \mathcal{T} \\ c_{D,S} : \mathcal{P}(\mathcal{T}) \to \mathcal{P}(\mathcal{T}) & c_{D,S}(T) = T \cup \{t' \in S | \exists t \in T, D(t,t') = \text{true}\} \\ \mu_{c_{D,S}} : \mathcal{T} \to \mathcal{P}(\mathcal{T}) & \mu_{c_{D,S}}(t) = \bigcup_{n=1}^{\infty} c_{D,S}^n(\{t\}) \end{array}$$

where $c_{D,S}(T)$ computes all the transitions of S which are immediately dependent on transitions in T. Since S is a finite set of transitions, $c_{D,S}$ is monotonic; if we reapply $c_{D,S}$ repeatedly, we eventually reach a set T (a fixpoint) where $c_{D,S}(T) = T$. The function $\mu_{c_{D,S}}$ represents this fixpoint. The set $\mu_{c_{D,T_e}}(t)$ is a good candidate for an ample set, since we know that at least no transition outside the set $\mu_{c_{D,T_e}}(t)$ is dependent on anything inside it. A good method to find the best ample set is to sort the sets $\mu_{c_{D,T_e}}(t)$, for all $t \in T_e$ based on their cardinality, and then start checking the conditions, beginning with the smallest one. This way, if we verify all the conditions for a candidate set, we are sure that it is the smallest possible ample set, and we are done.

Checking The Conditions. The most involved part of the partial order reduction algorithm is checking the conditions in [2]. Conditions $C2$ and $C3$ are exactly the same as in [2]. Condition $C'1$ is a stronger version (see Appendix B) of condition $C1$ from [2] (since the original $C1$ from the POR theory is not locally verifiable) and very similar to the variation of it in the heuristic proposed in [2]. Since the algorithm always works on nonempty sets, we are left to check three out of the four conditions. Here, we describe how the conditions are checked for a candidate set of transitions (ample set). The special case of a single transition as a candidate (as in [2]) follows from this easily.

$\mathcal{T}_e$ represents the set of all enabled transitions in the current state. Note that, as argued before, the notions of transition and of enabled thread are equivalent in our framework, so we often switch between the two.

C′1: if transition set $T \subset \mathcal{T}_e$ is a an ample set, then no thread in $\mathcal{T}_e - T$ should have a transition in future that is dependent on t. To compute future transitions of a thread $t_i \in \mathcal{T}_e - T$, a conservative flow-insensitive context-insensitive static analysis of the code is performed. This kind of static analysis can be done locally, and is different for different programming languages. Therefore, the language specifier/tool builder needs to provide it. In the definition of the algorithm we assume that there is an operation *ThreadTransitions* which takes the thread identifier and the current state of the system and returns all the future transitions of the thread in the form of a set of tuples (transition format) through a purely static analysis of the code of the input program which usually offers an overestimation of the actual set. Having the future transitions of all the threads in $\mathcal{T}_e - T$, condition $C'1$ can then be easily checked by using the *dependence* relation. To see that $C'1$ implies $C1$ in [2], see Appendix B.

C2: ample transitions should be invisible if the state is not fully expanded. This condition is the simplest of the three to verify. The set of propositions used in the desired property is given as an input. The check just has to go over this set, element by element, and check whether each proposition has the same truth value in state s and in its successor state with respect to all transitions in the ample candidate set.

C3: Cycle-closeness Condition. This condition ensures that no transition is enabled over a cycle in the state transition graph and is never taken in the ample set. This condition can be easily checked when the partial order reduction algorithm is embedded in a model checker, since the stack of states being explored is available. In our case, we use exactly the same method, but we simulate part of that stack as part of the state. The second component of the new system state, *StateInfoSet* takes care of this. Whenever in a state s there is a transition t outside the ample set, the pair (t, s) will be stored in the *StateInfoSet* component. As soon as a transition is taken in some future step, the pair is removed from the *StateInfoSet*. If a pair (t, s) is still there when we revisit s, we know that we are closing a cycle, so we must take the transition.

3.4 Correctness of the Theory Transformation

The correctness of our theory transformation can be now stated as the following theorem, , whose proof is sketched in Appendix B:

Theorem 1. *Assuming that a set AP of atomic state predicates has already been added to $\mathcal{R}_L$ by means of a set of equational definitions, the Kripke structures associated to the rewrite theories $\mathcal{R}_L$ (with* `State` *as its sort of states) and to $\mathcal{R}_{L+POR}$ (with* `PorState` *as its sort of states) are stuttering bisimilar.*

4 Applications of the Method and Experimental Results

We have implemented the theory transformation for our generic POR reduction method in a Maude [4] prototype and have used it to build POR units for Java bytecode and for a Promela-like language. In this section we illustrate how the method was used to build the POR unit for Java bytecode, which has been added to JavaFAN [8], a tool to formally analyze Java programs based on a rewriting semantics of both Java source code and bytecode. We also present some performance figures for both the JVM and the Promela-like language to show that the generic partial order module can result in drastic reductions in the state space of programs in the above languages.

4.1 The JVM POR Unit

By briefly discussing this example, we illustrate how the language-dependent parts are defined in Maude for the Java bytecode semantics to give a better understanding of these parts, and also to show that they can be specified by the tool builder with relatively little effort and in a program-independent way.

Extracting Transitions. There are 16 equations, corresponding to the 16 rewrite rules in the semantics of the Java bytecode, which extract all the enabled transitions from a given state. Here is an example of one of these equations:

```
ceq << S, < T: JavaThread | callStack:([PC, monitorenter, .., (REF(K) # OperandStack),
    ...] CallStack), .. > < O:JavaObject|Addr:K, .., Lock:Lock(OIL, NoThread, 0) > Ct >>
= << S {'MONITORENTER, T, noInfo}, < T: JavaThread | callStack: ([PC, monitorenter, ...,
     OperandStack, ...] CallStack), Status: scheduled, ... > < O: JavaObject | Addr: K,
     ..., Lock: Lock(OIL, NoThread, 0) > Ct >> if S {'MONITORENTER, T, noInfo} =/= S .
```

where `S` is the enabled transitions set. The equation says that if in the current state (containing a thread `T` , an object `O`, and a context `Ct` which captures the rest of the JVM state that is a multiset), `T` is ready to execute a `monitorenter` (lock) instruction, and `O` is not locked by any other thread, it means that the tuple `{'MONITORENTER, T, noInfo}` is an enabled transition, and it is added to the set `S` if it is not already in it.

Dependence Relation. The dependence relation for Java bytecode is defined based on the following facts: (1) two accesses to the same location are dependent if at least one of them is a write. This is defined through a few equations to cover the access to the instance fields as well as static fields; (2) two lock operations accessing the same lock are dependent. This is defined through a few equations to cover synchronized method calls, the `monitorenter` instruction, as well as the `notifyAll` built-in method of Java.
As an example of equations defining the dependence relation we have:

```
    eq Dependence({T, 'PutField, I}, {T', 'GetField, I'}) = true .
    eq Dependence({T, 'InvokeStatic, C}, {T', 'InvokeStatic, C) = true .
```

which specify that a read and a write to an instance field (first line) are always dependent, and (second line) two synchronized static method calls are dependent if they are locking the same class, `C`.

Thread Transitions. As mentioned at the end of Section 3.3, to check condition $C'1$, the operation *ThreadTransitions*, which conservatively computes the set of future transitions of a thread, has to be specified by the user. In the case of Java bytecode the idea is to start from the current point in t_i and add all the future instructions (transition steps) of the current method executing, and upon a method call, add in all the instructions (transitions) of the code of that method as well (avoiding repetition). This is conservative, in the sense that in the cases where more than one method can be the potential resolution of a call site, all of them are considered, and also in transitions such as reading/writing a field of an object where the object cannot be resolved until the point of execution, conservatively all possible objects will be considered.

4.2 Experiments

Table 1 presents the result of partial order reduction performance for the Promela-like language compared with results of the partial order reduction unit of SPIN from [3]. Since these are reports from different machines and different models, a one-to-one comparison of numbers is not meaningful, but the ratios of time/space

Program	Reduction	Time (ours)	States (ours)	Time ([3])	State ([3])
sieve	No	41s	61842	1.68	10878
	Yes	0.3s	174	0.08	157
ratios	—	136	355	21	69

Table 1. Time and Space Reduction Comparisons.

reduction can be compared. Table 2 shows the results of time/space reduction for a deadlock-free version of dining philosophers with different number of philosophers in the Promela-like language. Entries left empty indicate that we could not model check the example on our platform, a PC running Linux with a 2.4GHz processor and 4GB of memory.

Program	Reduction	Time	States
DP(5)	No	25.1s	56,212
	Yes	7.3s	3,033
DP(6)	No	146.2.0s	623,644
	Yes	30.0s	22,822
DP(7)	No	—	—
	Yes	5m	168,565
DP(8)	No	—	—
	Yes	66m	1,412,908

Table 2. Dining Philosophers.

Table 3 illustrates a dining philosophers program (5 philosophers) model checked in JavaFAN, where two versions of the dependency relation are compared. In the "basic" version, the dependency relation is the general version (presented in Section 4) that holds for all Java programs. The "NotShared" version lifts the dependencies of read/write memory accesses, since we know that the dining philosophers code does not use any shared memory and works merely based on locks. As shown in the table, a simple change like this (which means commenting out a few equations in the definition of the dependency relation) can result in a considerably better performance.

Test	Basic(t)	Basic(n)	NotShared(t)	NotShared(n)
Dining Philosophers	7m	6991	41s	2690

Table 3. Changing Dependency Relation.

Table 4 shows the state reduction obtained when the partial order reduction module is used. The JavaFAN tool reduces the number of states substantially by itself, since it uses the rewrite rules to model only the concurrent parts of Java (see [8] for details). But, the partial order reduction can still add a substantial reduction to that. PL is a two stage pipeline, DP is a deadlock-free version of the dining philosophers, RA is NASA's remote agent benchmark, and SE is a distributed sieve of Eratosthenes. All programs in these experiments, as well as the semantic definitions of the JVM and the Promela-like language and their POR-transformations by our method are available in [6].

Test	States (w POR)	States(wo POR)
PL	6612	18074
DP(5)	6991	16248
RA	24	33
SE	186	247

Table 4. Partial Order Reduction Results.

5 Conclusions

We have presented a generic method to build a model checker with POR capabilities for any programming language of interest, based on a theory transformation of the rewriting logic formal semantics of the given language. The instantiation of our method to a given programming language L of choice can be done semi-automatically and with relatively little effort by a tool builder familiar with the semantic definitions. Furthermore, since all POR computations are performed in the transformed theory itself, the method does not require any modifications to the underlying LTL model checker. Language-specific optimizations can also be added, because the heuristic algorithm and the dependence relation are explicit parameters of the theory transformation. Our experience evaluating this method in practice for the JVM and a Promela-like language indicates that significant state space reductions and time speedups can be gained.

The current prototype implementation of our method does not support various well-known optimization strategies, but many of these can be incorporated into our framework in a straightforward way. These strategies are often based on assumptions about the structure of the programming language under consideration. Therefore, they belong to the second, language-specific customization phase of our theory transformation, although in some cases they can be applied to entire families of languages. For example, a reduction strategy proposed in [5] for concurrent object oriented software is detecting heap objects that are *thread-local* to sharpen the dependence relation. All the static/dynamic analysis in [5] that leads to detecting the thread locality is possible in our framework, since we have both the static and dynamic information available. A more extensive experimentation with a broader set of language instantiations and incorporating the above optimizations should be performed in the future. Furthermore, the mechanical verification of the correctness of our theory transformation along the lines of the proof sketched in the Appendix should be investigated.

Another interesting direction for future work is extending our generic method beyond POR to also support what we have called "transaction-based reductions" in Section 1. Such reductions are complementary to those obtained by POR methods. We conjecture that a similar *theory transformation* would allow us to achieve transaction-based reductions in a generic way. The equation (*) in Section 3.2 works as a *nondeterministic scheduler* which in the present method schedules all the threads belonging to the *ample* set for the next step. In a transaction-based method the role currently played by the *MarkAmples* operation could instead schedule a single thread t, provided t is inside a *transaction*, and the component I could then be used for the instrumentation predicates.

References

1. R. Alur, R. K. Brayton, T. A. Henzinger, S. Qadeer, and S. K. Rajamani. Partial order reduction in sumbolic state exploration. In *CAV*, pages 340 – 351, 1997.
2. E. M. Clark, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 2000.
3. E.M. Clarke, O. Grumberg, M. Minea, and D. Peled. State space reduction using partial order techniques. In *Journal of STTT*, volume 2, pages 279 – 287, 1999.
4. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *Maude 2.0 Manual*, 2003. `http://maude.cs.uiuc.edu/manual`.
5. M. Dwyer, J. Hatcliff, and V. Prasad. Exploiting object escape and locking information in partial order reductions for concurrent object-oriented programs. *Formal Methods in System Design Journal*, 25:199–240, 2004.
6. A. Farzan. Specifications and examples. `http://maude.cs.uiuc.edu/POR/`.
7. A. Farzan, F. Chen, J. Meseguer, and G. Rosu. Formal analysis of Java programs in JavaFAN. In *Proceedings of CAV*, volume 3114, pages 501 – 505, 2004.
8. A. Farzan, J. Meseguer, and G. Rosu. Formal JVM code analysis in JavaFAN. In *AMAST*, volume 3116, pages 132 – 147, 2004.
9. C. Flanagan and P. Godefroid. Dynamic partial order reduction for model checking software. In *Proceedings of POPL*, 2005.
10. C. Flanagan and S. Qadeer. Transactions for software model checking. In *Workshop on Software Model Checking*, volume 338–349, 2003.
11. P. Godefroid. Partial-order methods for the verification of concurrent systems - an approach to the state-space explosion problem. In *Lecture Notes in Computer Science*, volume 1032. Springer-Verlag, 1996.
12. P. Godefroid. Model checking for programming languages using verisoft. In *POPL*, volume 174–186, 1997.
13. P. Godefroid and P. Wolper. A partial approach to model checking. In *Proceedings of Logic in Computer Science*, pages 406 – 415, 1991.
14. S. Katz and D. Peled. Defining conditional independence using collapses. *Theoretical Computer Science*, 101:337–359, 1992.
15. R. Kurshan, V. Levin, M. Minea, D. Peled, and H. Yenigun. Static partial order reduction. In *TACAS*, volume 1384, pages 345 – 357, 1998.
16. J. Meseguer. Conditional Rewriting Logic as a Unified Model of Concurrency. *Theoretical Computer Science*, pages 73–155, 1992.
17. J. Meseguer and G. Rosu. Rewriting logic semantics: from language specifications to formal analysis tools. In *Automated Reasoning*, volume 3097, pages 1–44, 2004.
18. D. Peled. All from one, one for all: on model checking using representatives. In *CAV'93*, LNCS, pages 409–423, 1993.
19. D. Peled. Combining partial order reduction with on-the-fly model checking. In *Proceedings of computer aided verification*, volume 818, pages 377 – 390, 1994.
20. S. D. Stoller. Model-checking multi-threaded distributed java programs. In *SPIN Workshop*, pages 224–244, 2000.
21. A. Valmari. A stubborn attack on state explosion. In *Proceedings of the 2nd workshop on computer aided verification*, volume 531, pages 156 – 163, 1990.
22. P. Viry. Equational rules for rewriting logic. *Theoretical Computer Science*, 285:487–517, 1992.

Appendix

A Proof of Proposition 1

Proof: (Sketch). Since we have assumed that $\mathcal{R}_L$ is coherent [22], which is the usual requirement for executability of a rewrite theory, the equations E_L are Church-Rosser and terminating, and we can assume that the rules in R_L are only applied to terms in E_L-canonical form. The first key obsevation is that the map π defines also a one-step bisimulation between the one-step equational rewriting relation $\longrightarrow^1_{E_L}$ and $\longrightarrow^1_{\widehat{E}_L}$. This can be shown by structural induction on the structure of rewrite proofs, using the fact that the equations in $\widehat{E}_L$ preserve all flags. It is then easy to show that $\widehat{E}_L$ is also terminating, and that a term $\widehat{u}$ is in $\widehat{E}_L$-canonical form iff $\pi(\widehat{u})$ is in E_L-canonical form. Showing that $\widehat{E}_L$ is Church-Rosser then follows using the just-established one-step bisimulation at the equational rewriting level, the fact that E_L is Church-Rosser, and by observing that the equations in $\widehat{E}_L$ are thread-preserving (because those in E_L are) and flag-preserving by construction. In a similar way we can then show that the rules in $\widehat{R}_L$ are coherent with respect to the equations $\widehat{E}_L$.

To show the one-step bisimulation at the level of the rewrite rules $\widehat{R}_L$ and R_L, we can now assume terms in $\widehat{E}_L$-canonical form (resp. E_L-canonical form). The fact that a one-step rewrite $\widehat{u} \longrightarrow_{\widehat{R}_L} \widehat{u'}$ induces a one-step rewrite $\pi(\widehat{u}) \longrightarrow_{R_L} \pi(\widehat{u'})$ follows easily from the definition of $\widehat{R}_L$ and of the function π. Conversely, if we have a one-step rewrite of state terms $u \longrightarrow_{R_L} u'$, say with a rule $r \in R_L$, we can always choose a term $\widehat{u} = \{v\} \in \pi^{-1}(u)$, where v is the term obtained from u by marking the flags in all threads of u as *true*. Then, by the definition of $\widehat{R}_L$, the rule $\widehat{r}$ applies to *widehatu* (note that if r and $\widehat{r}$ are conditional, then, by the bisimulation at the equational level, the r condition holds for u iff the $\widehat{r}$ condition holds for $\{v\}$) and we have a one-step rewrite $\widehat{u} \longrightarrow_{\widehat{R}_L} \widehat{u'}$ with $\pi(\widehat{u'}) = u'$. q.e.d.

B Proof of Theorem 1

Proof: (Sketch). The stuttering bisimulation we are after is a relation between terms of sort *porState* in $\mathcal{R}_{L+POR}$ and terms of sort *State* in $\mathcal{R}_L$. The bisimulation relation is defined by a surjective function π' which: (i) erases the $\{\_ \mid \_\}$ and $[\_ \mid \_]$ operators and discards the *StateInfo* components; and (ii) applies the π function defined in Section 3.2. We now have to show that both π' and its inverse relation π'^{-1} are stuttering simulations. But in fact, π' defines an ordinary simulation (therefore a trivial case of a stuttering simulation) from $\mathcal{R}_{L+POR}$ to $\mathcal{R}_L$, since any one-step application of the *step* rule requires a one-step rewrite with $\widehat{\mathcal{R}}_L$ of the correspoding *Mstate* components, that is, of the first erasing (i) above; and by Proposition 1 (see Section 3.2) $\widehat{\mathcal{R}}_L$ is one-step bisimilar to $\mathcal{R}_L$ with π, which is the second erasing (ii) above.

To prove that π'^{-1} is a stuttering simulation, we rely on Thoerem 12 of [2], which states that for every path in the original system R_L, there is a stuttering equivalent path in the system $\mathcal{R}_{L+POR}$ reduced with respect ample sets which satisfy conditions $C1$, $C2$, and $C3$ (Condition $C0$ is implicit in our case).

Note that the conditions $C2$ and $C3$ used in this paper (for both heuristics) are exactly the same as the corresponding conditions in [2]. Condition $C'1$ is a stronger version of condition $C1$ from [2], meaning that $C'1 \implies C1$. [2] defines $C1$ as follows:

C1: along every path in the full state graph that starts at s, the following condition holds: a transition that is dependent on a transition in ample(s) cannot be executed without a transition in ample(s) occurring first.

Our condition $C'1$ strengthens this condition in the sense that it says that a transition that is dependent on a transition in ample(s) cannot be executed along any path starting at s following a transition outside ample(s) at all. Those starting at s and following one of the ample(s) transitions clearly satisfy $C1$.