Partial Order Reduction for Rewriting Semantics of Programming Languages

Azadeh Farzan  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 $R_L$, 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 $R_L$ 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 (a few man-weeks in total, including defining the language semantics, for a language like Java) 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 specific 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. There can also be serious limitations in the degree of mathematical confidence that can be placed on such tools, when they are based on the low-level coding of the given language’s implementation; and it becomes then unfeasible, by lack of a formal semantics, to combine model checking and theorem proving in a rigorous manner.

To overcome these drawbacks, we, with several colleagues at UIUC including Feng Chen and Grigore Roşu, are investigating a generic methodology to develop software analysis tools for a wide range of programming languages based on a formal semantic definition of the programming language of interest [13, 8, 7]. Specifically, we develop a formal executable specification of the semantics of a given language in rewriting logic [12], and leverage the Maude efficient execution engine, and its generic support for breadth-first search and LTL model checking [4], to obtain software analysis tools for the language of interest virtually for free, as a straightforward instantiation of the generic Maude tools. Combination
of model checking and theorem proving then becomes both rigorously based and straightforward, using Maude’s inductive theorem prover [5]. The experimental results obtained so far in applying this technique to several languages, including Java, the JVM, and a substantial fragment of OCaml [13, 8, 7], are quite encouraging: developing software tools for a given language can be done in a few weeks instead of years; the tools are easily extensible and adaptable; and their performance is competitive with, and in some cases better than, that of language-specific tools.

The issue of partial order reduction (POR) was not addressed in our previous work. The main question we ask and provide an answer to in this paper is: is it possible to develop a generic, language-independent POR methodology based on the rewriting logic semantics of the programming language of interest? In this regard, it is worth noting that the rewriting logic semantics, by its flexible distinction between equations, used to specify deterministic language features, and rewrite rules, used to specify concurrent features, already provides a drastic state-space reduction, since rewriting with rules takes place modulo the semantic equations, so that the intermediate stages of a deterministic subcomputation are identified with the final state of such a subcomputation. What is further needed is to achieve an additional POR reduction of the state space by also reducing the state explosion due to the execution of concurrent features. That is, the issue to be investigated is how to apply in a generic, language-independent way POR techniques. The basic intuition behind POR techniques is that concurrent executions are really partial orders, and that, since the order of their occurrence is irrelevant, concurrent events should be left unordered. These methods then try to limit the expansion of each partial order computation to just one of its interleavings, instead of all of them. Studying language-independent POR methods seems interesting, since existing POR methods have been mostly suggested and implemented as an algorithmic modification to an existing model checker having a fixed language to describe the programs to be model checked.

Our proposed solution is based on a theory transformation, in which the original rewrite theory specifying the semantics of a given programming language is transformed into a semantically equivalent rewrite theory that accomplishes the desired partial order reduction when used for model checking a given program. As we explain in the paper, for a given programming language, this can be done once and for all, and in a relatively simple and principled way, by the programming language specifier/tool builder, thus endowing the given language’s search and LTL model checking tools with a POR capability. A software engineer not familiar with the tool’s foundations, but having a program written in such a language that needs to be analyzed, can then use the POR-enabled analysis tools so obtained to check whether the program satisfies desired properties. The only additional information that such an engineer has to provide is the definition of the specific state predicates used in the LTL property that must be model checked. We believe that our methodology has several useful strengths including the following:
Language-Independence. Our proposed method can be applied to model check programs in a wide range of programming languages, under minimal assumptions about the style of the rewriting logic semantic definitions of the programming languages. The language specifier/tool builder is only required to specify a few operators that work as an interface between the partial order reduction module and the semantics definition. When this interface is specified (once and for all), the partial order reduction module can be used to model check any given program in that language. Our experience shows that these additional operations can be defined using a few additional equations.

Flexible Partial Order Heuristic Algorithm. The heuristic algorithm is also 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 changing the source code of a model checker.

Flexible Dependence Relation. The dependence relation is the core of a partial order reduction algorithm. 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 to verify 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.

We report also on our experimental results with a first prototype of our generic theory transformation POR method, that we have applied to the rewriting semantics of the Java source code and bytecode and we have added to the JavaFAN [8] tool. We also report on the experimental results of applying the method to rewriting semantics of a simple Promela-like language which allows us to compare our method with previous experimental work.

Related Work. There are several works on developing methods to apply reduction principles in model checking. These works include the stubborn sets method of [15], the persistent sets method of [10] and the ample sets method of [14]. These works contain similar ideas, but differ in the details of the suggested reduction. 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.

[1] proposes a partial order reduction for symbolic state exploration, which is based on modifying the breadth search algorithm. These methods all work 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.
As an alternative to the above, somewhat dynamic approaches to partial order reduction, [11] proposes a static approach, where 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 solves 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, therefore it can benefit from all the detailed runtime information extractable from the state.

The rest of the paper is organized as follows: Section 2 contains the background knowledge needed in Section 3, where we describe the general method; Section 4 shows how the methodology can be applied to an example language; Section 5 shows how the approach works in practice; and Section 6 includes the conclusions and future directions.

2 Preliminaries

2.1 Rewriting Logic Language Specification

The rewriting logic semantics of a programming language [13] 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 $(\Sigma_L, E_L, R_L)$, with $\Sigma_L$ a signature specifying both the syntax of $L$ and of operations on auxiliary semantic entities like the store, environment, and so on, with $(\Sigma_L, E_L)$ an equational theory specifying the semantics of the deterministic, sequential features of $L$, and with $R_L$ a collection of (possibly conditional) rewrite rules specifying the semantics of $L$’s concurrent features.

Specifying formally the semantics of a concurrent programming language 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 provided, 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 [13, 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 $\alpha : S \rightarrow S$, $AP$ is a finite set of propositions and $L : S \rightarrow 2^{AP}$ is the labeling function. A transition $\alpha$ 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 \( D = (T \times T) \setminus I \) is the dependence relation. The second observation is that in many cases only a few transitions can change the truth value of the propositions which suggests the concept of visibility; a transition \( \alpha \in T \) is invisible if for each \( s, s' \in S \) where \( s' = \alpha(s) \) 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 3.5, 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 algorithm 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, or processes, 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 and 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 \( R_L = (\Sigma_L, E_L, R_L) \) specifying the semantics of a concurrent programming language \( L \) is transformed in two steps into the semantically equivalent theory \( R_{L+POR} = (\Sigma_{L+POR}, E_{L+POR}, R_{L+POR}) \) that is equipped with partial order reduction.

The Marked-State Theory. The objective of the first step of this transformation is to change the original theory \( R \) in order to facilitate the addition of the partial order module. In the transformed theory \( \hat{R}_L = (\hat{\Sigma}_L, \hat{E}_L, \hat{R}_L) \):

(1) The rewrite rules of \( R \) are changed syntactically to only allow one-step rewrites (see the Appendix A for details).

(2) The structure of the states of \( R \) 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 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. Appendix B gives a detailed construction of $\hat{R}_L$ and shows that $R_L$ and $\hat{R}_L$ are one-step bisimilar.

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

- $\Sigma_{L+POR} = \hat{\Sigma}_L \cup \Sigma_{POR} \cup \Sigma_{AUX}$, that is the signature $\hat{\Sigma}_L$ is extended with the signature of operators used in implementing the partial order heuristic algorithm $\Sigma_{POR}$, plus the signature of auxiliary operators $\Sigma_{AUX}$ that are used for implementation purposes.

- $E_{L+POR} = \hat{E}_L \cup E_{POR} \cup E_{AUX}$, that is the set of equations $\hat{E}_L$ are extended with the equations $E_{POR}$ which specify the partial order heuristic algorithm, plus the equations $E_{AUX}$ which define the auxiliary operators.

- $R_{L+POR} = \hat{R}_L \cup \{r_{step}\}$. 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 type of state, and therefore the only rule which will determine the transitions of the system at a given state.

### 3.3 The New State

Let $State$ be the sort of the state $s$ of the original system $R$. After marking $s$ we obtain a bisimilar state $s'$ of sort $MState$ in the theory $\hat{R}$. The new state $s''$ of $R_{L+POR}$ is then of sort $porState$, which is a pair of elements of sorts $MState$ and $StateInfoSet$. All the information necessary for the partial order reduction heuristic algorithm will be stored in this second component of the new state to be available whenever needed (see Section 3.5). Therefore, for each state $s'' = \{s'|I\}$ in the new system, there is an equivalent state $s$ in the old system.

### 3.4 The New Rewrite Theory

Having the rewriting semantics $(\Sigma_L, E_L, R_L)$ of a concurrent programming language $L$, one can view the initial state of the system (a program and its inputs) as a $\Sigma_L$-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 $r_i : l(u) \rightarrow r(v)$ is enabled at a position $p_j$ in term $s$, then we have a transition from $s$ to its successor $s[l(u)\backslash r(v)]$.

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 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.

The essence of partial order reduction is to look at this enabled set of transitions from the state \(s\) and find an ample subset of them to explore instead. A single new conditional rule \(r_{\text{step}}\) in \(R_{L+POR}\) simulates one step rewrites of the original system:

\[
\text{step} : \{s\}I \rightarrow [s']I \quad \text{if} \quad s \rightarrow s' \land s \neq s'
\]

where \(s\) and \(s'\) are variables of sort \(\text{MState}\), and the operators \(\{\_\}\) and \([\_\]\) are state constructors for the sort \(\text{porState}\) and are frozen operators \([4]\), that is, no rewriting is allowed below these operators. \(I\) is of sort \(\text{StateInfoSet}\) which is a component that holds all the information required for the heuristic algorithm (see Section 3.5). 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:

\[
[s]I = \{\text{state}(\text{MarkAmples}(s, I))|\text{stateInfo}(\text{MarkAmples}(s, I))\}
\]

The partial order reduction is applied at state \(s\) using the information in \(I\), by means of a single operation \(\text{MarkAmples}\). This operation takes a pair of elements of sorts \(\text{MState}\) and \(\text{StateInfoSet}\) as an input, and returns a pair of the same sort. The \(\text{MarkAmples}\) operation computes the ample set for the current state and returns the state with the ample transitions marked \(^1\) as specified by the algorithm. It also returns an updated version of \(\text{StateInfoSet}\) (see the algorithm part of Section 3.5).

### 3.5 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. Below, we describe how these tasks are performed.

**Extracting Enabled Transitions**

First, we have to specify the notion of a transition. As discussed in Section 3.4, a transition can be uniquely represented as a pair \((t_i, r_j)\) where \(t_i\) is a thread identifier and \(r_j\in R_L\) is a rewrite rule (or its unique label). However, 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.

\(^1\) An interface operation \(\text{MarkAmpleThreads}\) needs to be defined, which takes a state \(\text{MState}\) in \(\mathcal{R}\) and a list of thread identifiers (result of the algorithm), and returns the state with the threads in the list marked as enabled.
At a given state $s$, we have to find all pairs $(t_i, r_j : l(u) \rightarrow r(v))$ 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 $r \in R_L$ is of the following general form:

$$r : l(u) \Rightarrow r(v) \text{ if } C$$

Where $u$ and $v$ are sets of variables, and $C$ is the rule's condition. We should have a variable $t$ of sort $Tid$ (thread identifier) in both $u$ and $v$. The corresponding equation for $r$ is then:

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

Where $Ct(.)$ is the context\textsuperscript{2} in which the rule is applicable and $T_e$ is a set that accumulates enabled transitions. 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 (see Section 4).

Since the context information $I$ depends on the specific programming language $L$ and on the way that 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 edit these equations to include whatever context information may be useful later. In our experience with several rewriting semantics for different programming languages, there are only a few rewrite rules in the semantic definitions (that is, $E_L$ is much bigger than $R_L$), so this process is a rather quick and easy one.

**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 an operator Dependence which given two transitions returns true or false, depending on whether the two transitions are dependent or independent. Clearly, the dependence relations 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 as an equation of the following form:

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

where $< t, r, I >$ is a transition with $t$ a thread identifier, $r$ a rule name, and $I$ carries the 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.

\textsuperscript{2}If the rule $r$ rewrites the global state of the computation, the context $Ct(.)$ is empty, i.e. $Ct(l(u)) = l(u)$. 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.
**The Heuristic Algorithm.** Since the core of the heuristic algorithm can be specified using a few equations, we have specified two different heuristics. Figure 1 shows both algorithms. Functions $C_1'$, $C_2$, and $C_3$ check the three conditions discussed in next Section, returning `true` or `false`.

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_{e,s}$: enabled transitions in state $s$.</td>
<td>$\mu_{c_{D,S}}$: transitive closure of the dependence relation.</td>
<td></td>
</tr>
<tr>
<td>1 Take a transition $t$ from $T_{e,s}$.</td>
<td>1 Take a transition $t$ from $T_{e,s}$.</td>
<td></td>
</tr>
<tr>
<td>2 If $C_1'(t)$ and $C_2(t,P)$ and $C_3(t)$.</td>
<td>2 Let $S = \mu_{c_{D,T_e}}(t)$.</td>
<td></td>
</tr>
<tr>
<td>3 then mark thread of $t$ as ample. quit.</td>
<td>3 If $C_1(S)$ and $C_2(S,P)$ and $C_3(S)$.</td>
<td></td>
</tr>
<tr>
<td>4 else go to step 1.</td>
<td>4 then mark thread of $t$ as ample. quit.</td>
<td></td>
</tr>
<tr>
<td>5 Mark all threads as ample.</td>
<td>5 else go to step 1.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>6 Mark all threads as ample.</td>
<td></td>
</tr>
</tbody>
</table>

Fig. 1. Two Partial Order Reduction Heuristics.

These procedures are called at each state (see Section 3.4) 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 $T$ of transitions as follows:

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

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 fixed point) where $c_{D,S}(T) = T$. The function $\mu_{c_{D,S}}$ represents this fixed point. 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 $C1'$ is a stronger version (see Appendix B) of condition $C1$ from [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.

\( T_e \) 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.

**C1**: if transition set \( T \subseteq T_e \) is an ample set, then no thread in \( T_e - T \) should have a transition in future that is dependent on \( t \). To compute future transitions of a thread \( t_i \in T_e - T \), 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 \( \text{ThreadTransitions} \) 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). This should be quite easy. In our version of the algorithm for the Java bytecode this operation was defined by means of equations. Having the future transitions of all the threads in \( T_e - T \), condition \( C'1 \) can then be easily checked by using the \textit{dependence} relation. To see that \( C'1 \) implies \( C1 \) in [2], see Appendix B.

**C2**: \textit{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 its successor state with respect to all transitions in the ample candidate set.

**C3**: \textit{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 have to simulate part of that stack as part of the state. The second component of the new system state, \( \text{StateInfoSet} \) 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 \( \text{StateInfoSet} \) component. As soon as a transition is taken in some future step, the pair is removed from the \( \text{StateInfoSet} \). 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.

### 3.6 Correctness of the Theory Transformation

The correctness of our theory transformation can be now stated in 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 \( R_L \) by means of a set of equational definitions, the Kripke structures associated to the rewrite theories \( R_L \) (with \textit{State} as its sort of states) and to \( R_{L+POR} \) (with \textit{PorState} as its sort of states) are stuttering bisimilar.

### 4 POR for the Semantics of Java Bytecode

Here, we present how our method is used to add a partial order reduction component to JavaFAN [7], a tool to formally analyze Java programs based on a
rewriting semantics of both Java source code and bytecode. We explain how the language-dependent parts are defined for 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:

\[
\text{eq} \quad \text{S} \land \text{T} : \text{JavaThread} \mid \text{callStack} : ([\text{PC}, \text{monitorenter}, ... \text{(REF(K)} \# \text{OperandStack}), ... \} \text{CallStack}), \text{Status:scheduled}, ... > \\
\text{< O : JavaObject | Addr: K, ..., Lock: Lock(OIL, NoThread, 0) > Ct >>}
\]

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

Then comes the decision about what kind of information to include in the third component of the transition tuple (besides rule name and thread identifier). Since in Java, most targets of read, write, and lock operator are determined dynamically, there is not much information to help us resolve possible dependencies. For example, two different fields of two different objects of two different classes could refer to the same object on the heap at some point in the execution, so we have to be conservative and consider it. The only place where extra information (besides the rule name which depicts the action) can be helpful is for synchronized static method calls, which a class has to be locked. So, the only information that we add in such a case as the third component is the class name, which will be ignored and left empty in all other cases. As an example of equations defining the dependence relation we have:

\[
\text{eq} \quad \text{Dependence}([\text{T}, \text{PutField, I}], [\text{T}', \text{GetField, I}]) = \text{true} .
\]

\[
\text{eq} \quad \text{Dependence}([\text{T}, \text{InvokeStatic, C}], [\text{T}', \text{InvokeStatic, C}]) = \text{true} .
\]

which specifies 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, \(\text{C}\).

After declaring all possible dependencies in this way, we add a last equation stating that any other two transitions not covered by the above equations are independent.
Thread Transitions. As mentioned at the end of Section 3.5, to check condition $C'1$, the operation $\text{ThreadTransitions}$, 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.

5 Experiments

We present some experimental results on how the partial order module works for the semantics of Java bytecode as part of the JavaFAN tool and also for the semantics of a simple Promela-like language.

<table>
<thead>
<tr>
<th>Program</th>
<th>Reduction</th>
<th>Time (ours)</th>
<th>States (ours)</th>
<th>Time ([3])</th>
<th>State ([3])</th>
</tr>
</thead>
<tbody>
<tr>
<td>sieve</td>
<td>No</td>
<td>41s</td>
<td>61,842</td>
<td>1.68</td>
<td>10,878</td>
</tr>
<tr>
<td></td>
<td>Yes</td>
<td>0.3s</td>
<td>174</td>
<td>0.08</td>
<td>157</td>
</tr>
</tbody>
</table>

Table 1. Time and Space Reduction Comparisons.

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 reasonable, but the ratios of time/space 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.

In the case of the JavaFAN tool, we still observe considerable reduction in space, but in many cases no reduction in time. We believe that the cause of this is the much more complicated state of the JVM and the overhead of the partial order heuristic applied to a big state. Also, the Java language allows

<table>
<thead>
<tr>
<th>Program</th>
<th>Reduction</th>
<th>Time</th>
<th>States</th>
</tr>
</thead>
<tbody>
<tr>
<td>DP(5)</td>
<td>No</td>
<td>25.1s</td>
<td>56,212</td>
</tr>
<tr>
<td></td>
<td>Yes</td>
<td>7.3s</td>
<td>3,033</td>
</tr>
<tr>
<td>DP(6)</td>
<td>No</td>
<td>146.2s</td>
<td>623,644</td>
</tr>
<tr>
<td></td>
<td>Yes</td>
<td>30.0s</td>
<td>22,822</td>
</tr>
<tr>
<td>DP(7)</td>
<td>No</td>
<td>—</td>
<td>168,565</td>
</tr>
<tr>
<td></td>
<td>Yes</td>
<td>5m</td>
<td>168,565</td>
</tr>
<tr>
<td>DP(8)</td>
<td>No</td>
<td>—</td>
<td>1,412,908</td>
</tr>
<tr>
<td></td>
<td>Yes</td>
<td>66m</td>
<td>1,412,908</td>
</tr>
</tbody>
</table>

Table 2. Dining Philosophers.
fewer reductions in general due to the existence of the heap. Table 3 illustrates a dining philosophers program (5 philosophers) model checked in JavaFAN, where two versions of the dependency relation are compared. 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 presented 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.

<table>
<thead>
<tr>
<th>Test</th>
<th>Basic(t)</th>
<th>Basic(n)</th>
<th>NotShared(t)</th>
<th>NotShared(n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dining Philosophers</td>
<td>7m</td>
<td>6991</td>
<td>41s</td>
<td>2690</td>
</tr>
</tbody>
</table>

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 substantial amount of 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].

Table 4. Partial Order Reduction Results.

<table>
<thead>
<tr>
<th>Test</th>
<th>States (w POR)</th>
<th>States(wo POR)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PL</td>
<td>6612</td>
<td>18074</td>
</tr>
<tr>
<td>DP(5)</td>
<td>6991</td>
<td>16248</td>
</tr>
<tr>
<td>RA</td>
<td>24</td>
<td>33</td>
</tr>
<tr>
<td>SE</td>
<td>186</td>
<td>247</td>
</tr>
</tbody>
</table>

6 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. However, a more extensive experimentation with a broader set of language instantiations should be performed in the future. Three other topics that we plan to advance in future work are: (1) achieving a higher degree
of automation of the theory transformation process by exploiting Maude’s parameterized module instantiations mechanisms [4]; (2) using more sophisticated static analysis techniques to compute in a more optimized way the future transitions of a system; and (3) mechanically verifying the correctness of our theory transformation along the lines of the proof sketched in the Appendix.

References

Appendix

A The $\bar{R}_L$ Construction

We assume that all rules in $R_L$ are of sort $State$, and that there are sort $Thread$, classifying terms corresponding to threads in the state, and $ThreadId$ of thread identifiers. We define $\Sigma_L$ by removing from the signature $\Sigma$ the sort $State$ and adding three fresh new sorts: $State'$, $MState$, and $MThread$. All the state constructor operators of sort $State$ are now replaced by operators with the same name of sort $State'$, except that the use of the sort $Thread$ in those constructors is everywhere replaced by the sort $MThread$. We also add an operator $enabled : Thread \times Bool \rightarrow MThread$, and two unary operators $(\_), [\_] : State' \rightarrow MState$. We also assume that any subterm of sort $Thread$ appearing in either the original equations $E$ or rules $R$ is a term of the form $u(t)$, where $t$ is either a variable of sort $ThreadId$ or a ground term (and $u$ may perhaps have other variables). The equations in $E_L$ are systematically derived from those in $E$ by replacing in each equation in $E$ each occurrence of a thread expression $u(t)$ by the expression $enabled(u(t), b_t)$, where $b_t$ is a fresh new variable of sort $Bool$ depending on $t$ (so that different subterms $u'(t)$ appearing in the equation (with $u'$ perhaps different from $u$) will get the same $b_t$. The new set $R_L$ of rules is defined as follows. First of all, note that our assumption that a single thread is involved in each transition (see Section 3.1) means that any rule $r : s \rightarrow s'$ if $C$ in $R$ will be such that $s$ contains a single thread expression $u(t)$ and $s'$ a single thread expression $u'(t)$. The corresponding rewrite rule $\hat{r}$ in $R_L$ is then of the form $\hat{r} : (Ct(s)) \rightarrow \{ Ct(s') \} \text{ if } \hat{C}$, 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 (see Footnote 2), and where $\hat{s}$ and $\hat{s'}$ are obtained from $s$ and $s'$ by replacing $u(t)$ by $enabled(u(t), true)$ and $u'(t)$ by $enabled(u'(t), true)$, and where $\hat{C}$ is the conjunction of equations obtained from $C$ by changing each equation in $C$ containing thread expressions as done in the definition of $E_L$, and leaving all other equations untouched. Note that the use of the operators $(\_), [\_]$ in the rules in $R_L$ means that in $\bar{R}_L$, only one-step rewrites are possible, since the operator $[\_]$ in the righthand side blocks the application of any further rules.

The key point about the transformation $\mathcal{R}_L \rightarrow \bar{R}_L$ is then that the surjective projection $\pi$ mapping terms of sort $MState$ to terms of sort $State$ defined by: (i) erasing the operators $(\_), [\_)$, and (ii) erasing the $enabled$ operators and the corresponding flags defines a one-step bisimulation between the corresponding rewrite theories. That is, if we have a one-step rewrite $u \rightarrow v$ with $\mathcal{R}_L$, then we have also a corresponding one-step rewrite $\pi(u) \rightarrow \pi(v)$ with $\mathcal{R}_L$; and conversely, if we have a one-step rewrite $u' \rightarrow v'$ 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 $u \rightarrow v$ with $\bar{R}_L$. 

B Proof of Theorem 1

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

To prove that $\pi'^{-1}$ is a stuttering simulation, we rely on Theorem 12 of [2], which states that for every path in the original system $\mathcal{RL}$, there is a stuttering equivalent path in the system $\mathcal{RL}_{+P\text{OR}}$ reduced with respect ample sets which satisfy conditions $C_1$, $C_2$, and $C_3$ (Condition $C_0$ is implicit in our case).

Note that the conditions $C_2$ and $C_3$ 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 $C_1$ from [2], meaning that $C'_1 \implies C_1$. [2] defines $C_1$ as follows:

$C_1$: 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 $C_1$. 
