State Space Reduction for Model Checking Agent Programs

Sung-Shik T.Q. Jongmans[1], Koen V. Hindriks[2], and M. Birna van Riemsdijk[2]

[1] Centrum Wiskunde & Informatica, Amsterdam, the Netherlands
[2] Delft University of Technology, Delft, the Netherlands

Abstract. State space reduction techniques have been developed to increase the efficiency of model checking in the context of imperative programming languages. Unfortunately, these techniques cannot straightforwardly be applied to agents: the nature of states in the two programming paradigms differs too much for this to be possible. To resolve this, we adapt core definitions on which existing reduction algorithms are based to agents. Moreover, the framework that we introduce is such that different reduction algorithms can be defined in terms of the same relations. This is beneficial because it enables the reuse of code and reduces computation time when different techniques are used simultaneously. Specifically, we adapt and combine two known techniques: property-based slicing and partial order reduction. We exemplify our work with the Goal agent programming language, and implement the theory that we present for Goal. Several experiments with this implementation show that performance is in line with known results from traditional model checking.

1 Introduction

Model checking techniques for the verification of programs have traditionally been developed in the context of *imperative* programming languages (IPL). Ideally, for model checking programs written in *agent* programming languages (APL), one would take the technology and tools developed for IPLs, and apply them to agent programs without too much alteration. Unfortunately, this is sometimes an INEFFICIENT solution, and sometimes even IMPOSSIBLE:

INEFFICIENT — In [11], we show that it can be beneficial to develop new model checkers tailored to the verification of an APL rather than reusing existing tools for agent verification. The reason is that APL-tailored model checkers can reuse the APL's standard interpreter for fast generation of states. Consequently, there is no need to encode the agent program to lower-level code serving as input to an existing tool, which typically blows up the state space.

IMPOSSIBLE — In this paper, we argue that *state space reduction techniques*,[3] henceforth simply *reduction techniques*, known from traditional model check-

[3] State space reduction techniques combat the *state space explosion problem* (common to both IPL and APL model checking). This is the problem that systems to be verified are typically huge in terms of their *state space*, rendering model checking such systems often beyond our reach: it takes too many resources to finish verification.

ing cannot be applied directly in an agent context. The reason is that (dependencies between) states and transitions in the transition system of an imperative program differ fundamentally from those of an agent program.

Our main contribution is the redefinition, for agents, of concepts at the heart of existing reduction algorithms with a novel framework that brings together different techniques in a unifying way: we show that both *property-based slicing* (PBS) and *partial order reduction* (POR) can be defined in terms of the same relations using our framework. This enables a shared code base and runtime synergy: computations carried out for one algorithm can be reused by the other. We use the GOAL agent language [6] as running example throughout the paper.

The remainder is organised as follows. Section 2 provides background on model checking and GOAL. In Sect. 3, we argue why existing reduction techniques cannot straightforwardly be applied to agents, and introduce our framework. In Sect. 4, we define PBS and POR algorithms in terms of this framework. Section 5 discusses our implementation. Finally, Sect. 6 discusses related work with respect to reduction techniques in agent verification, and concludes the paper.

2 Preliminaries

Model checking Model checking [4] is a technique for automatically establishing whether a program P satisfies a property φ. Usually, φ is expressed in a *temporal logic*, a formalism for describing change over time. In this paper, we consider *linear temporal logic* (LTL) [4]. An LTL formula, denoted by ϕ or φ (if φ is a property to be model checked), is built from a set of propositions $\mathcal{P}$, the boolean connectives, and the temporal operators $\bigcirc$ (next), $\mathcal{U}$ (weak until), and $\mathcal{R}$ (strong release). We denote the set of all LTL formulas by $\mathcal{L}$. An LTL formula is interpreted over an infinite sequence of states, which we call a *computation*, denoted by $\boldsymbol{\pi}$. Let $i \geq 0$ be an index of $\boldsymbol{\pi}$, and let $\models$ be LTL's entailment relation. Purely propositional (sub)formulas are interpreted with respect to the i-th state on $\boldsymbol{\pi}$, denoted by $\boldsymbol{\pi}_i$, using a *valuation function* $\mathcal{V}$. Such a function maps a state to the set of propositions in $\mathcal{P}$ that are true in it. Temporal (sub)formulas are interpreted with respect to the (infinite) postfix of $\boldsymbol{\pi}$ starting in the i-th state:

$$\begin{array}{lll}
\boldsymbol{\pi}, i \models \bigcirc\, \phi & \text{iff } \boldsymbol{\pi}, i+1 \models \phi & \\
\boldsymbol{\pi}, i \models \phi\, \mathcal{U}\, \phi' & \text{iff } \exists_{k \geq i}(\boldsymbol{\pi}, k \models \phi' \text{ and } \forall_{i \leq j < k}(\boldsymbol{\pi}, j \models \phi)) & \\
\boldsymbol{\pi}, i \models \phi\, \mathcal{R}\, \phi' & \text{iff } \boldsymbol{\pi}, i \models \neg(\neg\phi\, \mathcal{U} \neg\phi') & \text{(note that } \mathcal{R} \text{ is the dual of } \mathcal{U})
\end{array}$$

In model checking, the program P is represented by its *transition system* $\mathcal{T} = \langle M, \mu_0, \longrightarrow \rangle$ in which M is a finite *set of states*, $\mu_0 \in M$ is the *initial state*, and $\longrightarrow \subseteq M \times M$ is a *transition relation* connecting states. A *path* π through $\mathcal{T}$ is an infinite sequence of states $\pi_0 \pi_1 \cdots$ such that for all $i \geq 0$: $\pi_i, \pi_{i+1} \in M$ and $\pi_i \longrightarrow \pi_{i+1}$. A computation $\boldsymbol{\pi}$ of P is a path through its transition system that starts in μ_0, i.e. $\boldsymbol{\pi}_0 = \mu_0$. We denote the set of all computations of P by $\boldsymbol{\Pi}$. The model checking problem for P and φ, given a valuation function $\mathcal{V}$, can now be formulated more formally as follows: determine for all $\boldsymbol{\pi} \in \boldsymbol{\Pi}$ whether $\boldsymbol{\pi}, 0 \models \varphi$.

In that case, we say that P *satisfies* φ. Otherwise, if there exists a $\boldsymbol{\pi} \in \boldsymbol{\Pi}$ such that $\boldsymbol{\pi}, 0 \models \neg\varphi$, P is said to *violate* φ, and $\boldsymbol{\pi}$ is called a *counterexample*.

Various approaches to model checking exist. In this paper, we assume *NDFS explicit-state automata-theoretic LTL model checking* [4], because the implementation we discuss in Sect. 5 extends [11] in which this approach is also taken.[4] In this approach, every $\boldsymbol{\pi} \in \boldsymbol{\Pi}$ is checked for satisfaction of $\neg\varphi$ in *negation normal form* (NNF). If such a computation is found, the model checker immediately halts, and reports it as a counterexample. Otherwise, the model checker terminates after investigating all computations, reporting that $\neg\varphi$ is not satisfied by any computation, i.e. φ is satisfied by all computations. Thus, instead of determining if all computations satisfy φ, in fact one determines whether there exists a counterexample. Henceforth, we assume all LTL formulas in NNF.

An important optimisation that the sketched approach allows for is *on-the-fly exploration*: the transition system of the program under investigation is generated *during* execution of the model checking algorithm instead of *before* it. Consequently, if a counterexample is quickly found and the model checker terminates, no resources have been spent on the generation of parts of the transition system whose inspection has turned out unnecessary. Importantly, the reduction algorithms discussed next are compatible with on-the-fly model checking.

GOAL The Goal agent programming language [6] facilitates programming of *rational agents* (i.e. agents that pursue their goals) at the cognitive level: agents choose their actions by reasoning about their *beliefs* and *goals*, which are expressed in some knowledge representation language $\mathcal{L}_X$ (e.g. Prolog). The beliefs that a Goal agent has at some point in time are stored in its *belief base*, denoted by Σ. Similarly, the goals of a Goal agent are stored in its *goal base*, denoted by Γ. Goals are *declarative*: they specify *what* the desired state of the world is instead of *how* this state may be brought about. Together, the belief and goal base of an agent constitute its *mental state*, denoted by $\mu = \langle \Sigma, \Gamma \rangle$.[5]

A Goal agent derives its choice of action from its mental state, hence it needs a mechanism to inspect it. To this end, agents evaluate *mental state conditions* (MSC). An MSC, denoted by ψ, is a boolean expression about the beliefs and goals of an agent, according to the following syntax:

$$\begin{aligned} \chi &::= \text{any well-formed formula from } \mathcal{L}_X \\ \psi &::= \mathbf{bel}(\chi) \mid \mathbf{goal}(\chi) \mid \neg\psi \mid \psi \wedge \psi \end{aligned}$$

The semantics of MSCs is defined by the entailment relation $\models_{\text{MS}}$ [6]. Informally, if μ is a mental state then $\mu \models_{\text{MS}} \mathbf{bel}(\chi)$ is true if χ is believed by the agent;

[4] Another well-known approach is *symbolic model checking* using *binary decision diagrams* (BDD) [4, 13]. This approach is based on an abstraction technique different from the techniques discussed here and is out of scope of this work.

[5] Although we do not discuss knowledge, our implementation is able to deal with this; in contrast, modules [6], percepts, and beliefs about dynamic environments that evolve independently of the agent's acting are at present beyond our scope.

similarly, $\mu \models_{\text{MS}} \mathbf{goal}(\chi)$ is true if χ is a goal of the agent. The set of all MSCs, denoted by $\mathcal{L}_{\text{MS}}$, is called the *language of mental state conditions.*

MSCs are used in the definition of *action rules.* An action rule, denoted by ρ, is a statement of the form **if** ψ **then** α in which α is an *action.* An action rule may be read as "if ψ is true, then the agent may consider performing α". In that case, the action rule is said to be *applicable.* The effects that performance of an action have on the mental state of an agent are formalised by the *mental state transformer*, denoted by $\mathcal{M}$. The mental state transformer maps an action and a mental state to a *successor mental state.* $\mathcal{M}$ need not be defined for all mental state–action pairs $\langle \mu, \alpha \rangle$: if $\mathcal{M}$ is undefined for μ and α, this means that α cannot be performed in μ. A precise definition of $\mathcal{M}$ is given in [6].

Let μ be a mental state, and let $\rho =$ **if** ψ **then** α be an action rule. If ρ is applicable in μ (i.e. $\mu \models_{\text{MS}} \psi$) and $\mathcal{M}(\alpha, \mu)$ is defined, then α is called an *option* in μ. During each reasoning cycle, a Goal agent determines its options given its current mental state and set of action rules, and chooses and performs one of them non-deterministically. This is formalised by an *operational semantics.* Let **if** ψ **then** α be an action rule, and let μ be a mental state. Then, the transition relation $\longrightarrow$ is the smallest relation induced by the following transition rule:

$$\frac{\mu \models_{\text{MS}} \psi \qquad \mathcal{M}(\alpha, \mu) \text{ is defined}}{\mu \longrightarrow \mathcal{M}(\alpha, \mu)}$$

The transition relation $\longrightarrow$ is subsequently used to define the transition system $\mathcal{T} = \langle M, \mu_0, \longrightarrow \rangle$ of a Goal agent, in which we assume that M is a finite[6] set of mental states and that μ_0 is the initial mental state of the agent.

Example 1. The source code and transition system of a simple example Goal agent, whose task is to put on two socks, appears in Fig. 1. We use this agent, called `socksAgent`, as a running example throughout this paper.

For model checking Goal agents, we instantiate the set of LTL propositions $\mathcal{P}$ with the language of mental state conditions $\mathcal{L}_{\text{MS}}$. The valuation function $\mathcal{V}$ in this case maps every mental state μ to the MSCs that are true in it, i.e. $\mathcal{V}(\mu) = \{\psi \in \mathcal{L}_{\text{MS}} \mid \mu \models_{\text{MS}} \psi\}$. This allows us to formulate and verify properties about the evolution of beliefs and goals of a Goal agent during its execution.

A final remark on terminology. Although we illustrate our techniques with Goal, they can be applied to other agent languages as well. Therefore, when we write "mental state" in what follows, we do not refer exclusively to a state of a Goal agent, but rather to a state of an agent written in some BDI-based APL.

3 Operations on Mental States

The aim of reduction techniques is to remove sets of transitions from the transition system that do not affect the truth value of the property under investigation. In our framework, we identify such sets of transitions by classifying them

[6] Finiteness is not imposed by Goal, but a model checking termination requirement.

```
main: socksAgent{
  beliefs{
    bothSocksOn :- wearing(sock,left),
                   wearing(sock,right).
  }
  goals{
    wearing(sock,left). wearing(sock,right).
  }
  program{
    if     goal(wearing(sock,left))
      then putOn(sock,left).
    if     goal(wearing(sock,right))
      then putOn(sock,right).
  }
  action-specs{
    putOn(S,X){
      pre{ not(wearing(S,X)) }
      post{ wearing(S,X) }
} } }
```

$$\begin{tikzcd} \mu_0 \arrow[r, "t_0"] \arrow[d, "t_2"'] & \mu_1 \arrow[d, "t_1"] \\ \mu_2 \arrow[r, "t_3"] & \mu_3 \end{tikzcd}$$

$$\mu_0 = \left\langle \emptyset \, , \left\{ \begin{array}{l} \texttt{wearing(sock,left)}, \\ \texttt{wearing(sock,right)} \end{array} \right\} \right\rangle$$

$$\mu_1 = \left\langle \begin{array}{l} \{\texttt{wearing(sock,left)}\}\,, \\ \{\texttt{wearing(sock,right)}\} \end{array} \right\rangle$$

$$\mu_2 = \left\langle \begin{array}{l} \{\texttt{wearing(sock,right)}\}\,, \\ \{\texttt{wearing(sock,left)}\} \end{array} \right\rangle$$

$$\mu_0 = \left\langle \left\{ \begin{array}{l} \texttt{wearing(sock,left)}, \\ \texttt{wearing(sock,right)} \end{array} \right\} , \emptyset \right\rangle$$

Fig. 1. Example agent. On the left, its source code; on the right, its transition system.

in terms of *operations*. Informally, we may think of an operation, denoted by τ, as a function that transforms states μ to other states μ'. In that case, τ is said to be *applied* to μ. More specifically, we characterise an operation in terms of the CHANGES that it brings about, and the STATEMENT in the source code from which it can be induced. Below, let $\mathcal{T} = \langle M, \mu_0, \longrightarrow \rangle$ be the transition system of some agent program P, and let $t = \langle \mu, \mu' \rangle \in \longrightarrow$ be a transition.

CHANGES — Grouping individual transitions in $\mathcal{T}$ according to the changes that they bring about enables us to express that the order in which two operations can be applied is without consequence (relevant in POR). To formalise this notion, let $\mathsf{Ch}(t)$ denote the change between μ and μ'.

STATEMENT — Characterising operations by statements allows us to remove sets of transitions from $\mathcal{T}$ by deleting statements from P's source code. This enables us, for instance, to reduce $\mathcal{T}$ by performing static analysis of the program text alone (relevant in PBS). To formalise this notion, let $\mathsf{St}(t)$ denote the set of statements in P's source code from which t can be induced.

Example 2. In case of GOAL, $\mathsf{Ch}(t)$ denotes the beliefs and goals to be added and deleted to get from μ to μ', and $\mathsf{St}(t)$ denotes the action rules that induce t. Applied, for instance, to transition $t_0 = \langle \mu_0, \mu_1 \rangle$ of `socksAgent` in Fig. 1 yields: $\mathsf{Ch}(t_0) = \langle \Sigma + \{\texttt{wearing(sock,left)}\} - \emptyset, \Gamma + \emptyset - \{\texttt{wearing(sock,left))}\}\rangle$ and $\mathsf{St}(t_0) = \{\texttt{if goal(wearing(sock,left)) then putOn(sock,left)}\}$.

We now define an operation τ formally.

Definition 1. *An operation is a pair $\tau = \langle T, s \rangle$ in which s is a statement and $T \subseteq \longrightarrow$ is the largest set such that for all $t, t' \in T$: $\mathsf{Ch}(t) = \mathsf{Ch}(t')$ and $s \in \mathsf{St}(t)$.*

Example 3. We identify the following operations of `socksAgent` in Fig. 1:

$$\tau_0 = \langle \{t_0, t_3\}, \texttt{if goal(wearing(sock,left)) then putOn(sock,left)} \rangle$$
$$\tau_1 = \langle \{t_1, t_2\}, \texttt{if goal(wearing(sock,right)) then putOn(sock,right)} \rangle$$

We use the following notation and definitions. The set of all possible operations is denoted by Ω_τ. If $\tau = \langle T, s\rangle$ is an operation, then we use $\mathsf{Tran}(\tau)$ and $\mathsf{Stat}(\tau)$ as a shorthand for, respectively, T and s. We call $\mathsf{Stat}(\tau)$ the statement that *induces* τ, and say that τ is *enabled* in a state μ if there exists a μ' such that $\langle \mu, \mu' \rangle \in \mathsf{Tran}(\tau)$; we write $\tau(\mu)$ as a shorthand for μ'. The set of all enabled operations in μ is denoted by $\mathsf{En}(\mu)$, i.e. $\mathsf{En}(\mu) = \{\tau \in \Omega_\tau \mid \tau \text{ is enabled in } \mu\}$. The set of all operations $\mathsf{Ops}(s)$ that a statement s can induce is called its *operation class*, defined as $\mathsf{Ops}(s) = \{\tau \in \Omega_\tau \mid \mathsf{Stat}(\tau) = s\}$. Finally, for brevity, we write $\mathsf{Ch}(\tau)$ to denote the change brought about by any $t \in \mathsf{Tran}(\tau)$, and write $\mathsf{Ch}(s)$ to denote the set at least having $\bigcup_{\tau \in \mathsf{Ops}(s)} \mathsf{Ch}(\tau)$ as a subset.

3.1 Variable Assignments versus Mental States

State space reduction techniques have originally been developed for use with transition systems whose states are characterised by variables and their values, henceforth called *variable assignment*. By carefully analysing which variables change by applying operations on states (i.e. when moving from one state to the next), relations on operations essential to the application of reduction algorithms can be computed. For instance, one can determine whether enabledness of an operation τ' is affected by the application of an operation τ, by comparing the variables that τ mutates and τ' accesses. We call the sets of variables an operation τ accesses and mutates its *read set*, denoted by $\mathsf{Read}(\tau)$, and its *write set*, denoted by $\mathsf{Write}(\tau)$, respectively. These sets are not used only for determining whether enabledness of operations depends on the application of (other) operations, but also to determine if the application of an operation influences the truth value of LTL formulas. Importantly, analyses based on read and write sets can be done by inspection of the source code alone: the read and write set of an operation τ can be determined straightforwardly by inspecting the variables occurring in the statement that induces τ, i.e. $\mathsf{Stat}(\tau)$. This is of great value, because it allows for *off-line computation* of (most of the) reduction algorithms. This means that the computation of these algorithms does not depend on information that is available only during model checking. Because processing information that is available only at runtime (i.e., while we run the actual model checking algorithm that searches for a counterexample) is likely to be more expensive (e.g., because subroutines of the algorithm need be computed for each state in a transition system), off-line algorithms reduce the overhead at runtime to a minimum.

Example 4. Suppose two operations $\tau, \tau' \in \Omega_\tau$ such that $\mathsf{Stat}(\tau) = [x := x + 1]$ and $\mathsf{Stat}(\tau') = [y := z + 42]$ are enabled simultaneously in some variable assignment ν, e.g. because they belong to different concurrent processes (and x, y, z are shared variables). Then: $\mathsf{Read}(\tau) = \mathsf{Write}(\tau) = \{x\}$ and $\mathsf{Read}(\tau') = \{z\}$ and $\mathsf{Write}(\tau') = \{y\}$. Because $\mathsf{Read}(\tau) \cap \mathsf{Write}(\tau') = \mathsf{Write}(\tau) \cap \mathsf{Read}(\tau') = \emptyset$, application of τ cannot cause τ' to become disabled and vice versa.

When model checking agent programs, however, states are *not* characterised by variable–value pairs, but by mental attitudes, which are very different: how and which mental attitudes change over time is not stated explicitly in the program text, e.g. due to underspecification. We elaborate on this in Sect. 3.2. Hence, in agent verification, we cannot use directly the analysis techniques known from traditional model checking to compute the relations essential to the application of reduction algorithms: the gap between variable assignments and mental states need be bridged. Specifically, to be able to reuse existing reduction algorithms for agents, we need to answer (in the next subsection) the following questions:

1. What are the elements constituting read and write sets when dealing with mental states of agents, which are not composed of variable–value pairs?
2. Given a definition of read and write sets for mental states of agents, can we still compute them off-line?

3.2 Read Sets and Write Sets for Mental States

Ad 1. We aim at a definition of read and write sets for mental states that is sufficiently generic in the sense that these definitions should accommodate multiple APLs. This is nontrivial, because mental states look different in each APL, i.e. the mental attitudes constituting a mental state vary between different languages. To this end, we introduce the notion of an APL-specific *condition language*, denoted by $\mathcal{L}_K$, whose elements are *conditions*, denoted by κ. Informally, the idea is that the read set of an operation τ contains those conditions that *must* be true for τ to be enabled, while τ's write set contains those conditions whose truth value changes due to application of τ. Thus, $\mathsf{Read}(\tau) \subseteq \mathcal{L}_K$ and $\mathsf{Write}(\tau) \subseteq \mathcal{L}_K$. The only requirement that $\mathcal{L}_K$ must satisfy is that it should have the set of propositions $\mathcal{P}$ as a subset, i.e. $\mathcal{P} \subseteq \mathcal{L}_K$: this allows us to determine, by means of write set analysis, whether a transition can affect the truth value of a property. Apart from that, $\mathcal{L}_K$ can be tailored completely to the needs of the APL.

Example 5. In the context of GOAL, the condition language equals the language of MSCs, i.e. $\mathcal{L}_K = \mathcal{L}_{\text{MS}}$ (recall that $\mathcal{P} = \mathcal{L}_{\text{MS}}$ for GOAL).

Next, to accommodate formal definitions, we assume an entailment relation $\models_K$, relating (mental) states to conditions that are true in them, and a function $\mathcal{I}$ mapping a mental state μ to the subset of $\mathcal{L}_K$ that is true in μ, i.e. $\mathcal{I}(\mu) = \{\kappa \in \mathcal{L}_K \mid \mu \models_K \kappa\}$. Read and write sets are then defined formally as follows.

Definition 2. *Let τ be an operation. Then:*

$$\begin{array}{lll}
\mathsf{Read}(\tau) & = & \{\kappa \in \mathcal{L}_K \mid \textit{there exist states } \mu, \mu' \textit{ s.t. } \tau \in \mathsf{En}(\mu), \tau \notin \mathsf{En}(\mu') \\
 & & \qquad\qquad \textit{and } \kappa \in \mathcal{I}(\mu) \textit{ and } \mathcal{I}(\mu') = \mathcal{I}(\mu) \setminus \{\kappa\} \qquad\qquad \} \\
\mathsf{Write}^+(\tau) & = & \bigcup_{\langle \mu, \mu' \rangle \in \mathsf{Tran}(\tau)} \mathcal{I}(\mu') \setminus \mathcal{I}(\mu) \\
\mathsf{Write}^-(\tau) & = & \bigcup_{\langle \mu, \mu' \rangle \in \mathsf{Tran}(\tau)} \mathcal{I}(\mu) \setminus \mathcal{I}(\mu') \\
\mathsf{Write}(\tau) & = & \mathsf{Write}^+(\tau) \cup \mathsf{Write}^-(\tau)
\end{array}$$

We call $\mathsf{Write}^+(\tau)$ *and* $\mathsf{Write}^-(\tau)$ *the positive and negative write sets of τ;* $\mathsf{Write}(\tau)$ *is sometimes referred to as τ's total write set.*

We use the distinction between positive and negative write sets in Sect. 3.3. The distinction is important, because it allows us, for instance, to state that some transition τ can enable a transition τ': in that case, the *positive* write set of τ coincides with the read set of τ'. Conversely, if τ's *negative* write set does *not* coincide with the read set of τ', τ cannot disable τ'. Note that "not disabling" is different from "enabling", and in general, Write^+ and Write^- are not each other's complement: $\mathcal{L}_K \setminus \mathsf{Write}^+(\tau) \neq \mathsf{Write}^-(\tau)$ and $\mathcal{L}_K \setminus \mathsf{Write}^-(\tau) \neq \mathsf{Write}^+(\tau)$.

Example 6. Consider operation τ_0 of `socksAgent`, defined in Ex. 3. For convenience, we restrict this example to the MSC set $\{$`goal(wearing(sock,left))`, `goal(wearing(sock,right))`, `bel(bothSocksOn)`$\} \subset \mathcal{L}_{\text{MS}}$. Now, the positive write set of τ_0 equals $\{$`bel(bothSocksOn)`$\}$, while both its read set and negative write set equal $\{$`goal(wearing(sock,left))`$\}$. From this, we can deduce that τ_0 disables itself, while it has no effect on enabledness or disabledness of τ_1.

Ad 2. As outlined in Sect. 3.1, off-line computation of read and write sets is important, because it reduces the resource consumption of reduction algorithms at runtime. For imperative programming languages, as shown in Ex. 4, this can be done easily. Unfortunately, in case of agent programs, the situation is more complex: conditions from $\mathcal{L}_K$ often do not occur explicitly in the agent's source code, and cannot be simply extracted from it without further analysis.

Example 7. Consider the read and write sets of operation τ_0 of `socksAgent` given in Ex. 6. While τ_0's read set can be determined straightforwardly from the action rule `if goal(wearing(sock,left)) then putOn(sock,left)`, this is not the case for its write set for two reasons. First, the removal of the goal `wearing(sock,left)` occurs automatically due to GOAL's semantics, and is not specified explicitly in the program text. Second, the derivation of `bothSocksOn` using the Prolog rule in the belief base (see Fig. 1) cannot be detected by inspection of this action rule alone.

Switching to a more general perspective, we must deal with two issues when computing read and write sets for GOAL agents. First, not all beliefs and goals that an operation adds or deletes can be derived from the source code of a GOAL agent, making it difficult to determine which MSCs incur a change of truth value. Second, as changing the belief base by an operation also changes the consequences that can be derived from Prolog rules, we need an algorithm to approximate these. The issue is that this algorithm must run on only the source code and that the content of the belief base at runtime is unknown.

Thus, we may need to derive read and write sets with more complex analysis techniques. Unfortunately, it may be impossible to compute *precise* read and write sets using the source code alone due to underspecification of the agent or the occurrences of uninstantiated variables combined with Prolog-style reasoning as sketched in the previous example. There are two ways to resolve these issues: acquire sufficient information by generating the entire transition system, or use *approximation techniques.* We prefer the latter, because the former is incompatible with on-the-fly model checking. We stress that approximation is unnecessary

Table 1. Formal definition of relations on operations and statements.

Relation	Precise (for operations τ, τ')	Approximate (for statements s, s')
Visibility	$\mathsf{Vis}(\tau, \phi)$ iff $\mathit{Props}(\phi) \cap \mathsf{Write}(\tau) \neq \emptyset$	$\mathfrak{Vis}(s, \phi)$ iff $\mathit{Props}(\phi) \cap \mathfrak{Write}(s) \neq \emptyset$
Enables	$\mathsf{Enables}(\tau, \tau')$ iff $\mathsf{Read}(\tau') \cap \mathsf{Write}^+(\tau) \neq \emptyset$	$\mathfrak{Enables}(s, s')$ iff $\mathfrak{Read}(s') \cap \mathfrak{Write}^+(s) \neq \emptyset$
Independence	$\mathsf{Indep}(\tau, \tau')$ iff $H^{en}_{\mathsf{Indep}}(\tau, \tau')$ and $H^{comm}_{\mathsf{Indep}}(\tau, \tau')$	$\mathfrak{Indep}(s, s')$ iff $H^{en}_{\mathfrak{Indep}}(s, s')$ and $H^{comm}_{\mathfrak{Indep}}(s, s')$

in an IPL context, because there, read and write sets can be obtained with straightforward source code inspection.

The key property any approximation technique for read and write sets must satisfy is that of *over*-approximation: to ensure that model checking with reduction algorithms yields the same results as without, approximate read and write sets (denoted here in $\mathfrak{font}$) need to over-approximate the precise sets. Formally:

Property 1. Let s be a statement. For all $\tau \in \mathsf{Ops}(s)$: $\mathsf{Read}(\tau) \subseteq \mathfrak{Read}(s)$ and $\mathsf{Write}^+(\tau) \subseteq \mathfrak{Write}^+(s)$ and $\mathsf{Write}^-(\tau) \subseteq \mathfrak{Write}^-(s)$ and $\mathsf{Write}(\tau) \subseteq \mathfrak{Write}(s)$.

Intuitively, over-approximation of read and write sets is required because these sets are used to determine dependencies between operations: the less dependencies present, the more reduction can be obtained. Thus, if all operations depend on each other, no reduction is gained. By over-approximating, dependencies that actually do not exist are nevertheless assumed. Although this may cause reduction algorithms to be less effective, correctness is assured. Henceforth, we assume all sets $\mathfrak{Read}$ and $\mathfrak{Write}$ to satisfy Property 1 (e.g. in the proof of Lemma 1).

3.3 Relations on Operations

Next, we use read and write sets to define relations on operations known from existing literature [4] on reduction techniques (see Table 1), and used by the algorithms in Sect. 4. Our contribution is that we define each relation not only in terms of precise read and write sets, but also in terms of their approximate counterparts. The resulting *approximate relations* can be computed before the transition system is generated (instead of during its generation), i.e. off-line. This reduces computational overhead of reduction algorithms at runtime to a minimum, and ensures compatibility with on-the-fly model checking. We prove lemmas to show how the precise and approximate relations relate to each other.

The first relation we discuss is the *visibility relation* Vis. Let τ be an operation, and let ϕ be an LTL formula. Then, $\mathsf{Vis}(\tau, \phi)$ states that application of τ can affect the truth value of ϕ; the formal definition can be found in Table 1. Because Vis is defined in terms of precise write sets, which typically cannot be computed off-line (see Sect. 3.2), we introduce the *approximate visibility relation* $\mathfrak{Vis}$, which is an approximation of Vis defined in terms of approximate write sets (see Table 1). Relations Vis and $\mathfrak{Vis}$ are related by the following lemma.

Table 2. Independence conditions, definitions, and heuristics.

Condition	Heuristic	Approximate heuristic
ENABLEDNESS : $\tau \in \mathsf{En}(\tau'(\mu))$	$H^{en}_{\mathsf{Indep}}(\tau, \tau')$: $\mathsf{Read}(\tau') \cap \mathsf{Write}^{-}(\tau) = \emptyset$	$H^{en}_{\mathfrak{Indep}}(s, s')$: $\mathfrak{Read}(s') \cap \mathfrak{Write}^{-}(s) = \emptyset$
COMMUTATIVITY : $\tau(\tau'(\mu)) = \tau'(\tau(\mu))$	$H^{comm}_{\mathsf{Indep}}(\tau, \tau')$: $\mathsf{Ch}(\tau) \cap \mathsf{Ch}(\tau') = \emptyset$	$H^{comm}_{\mathfrak{Indep}}(s, s')$: $\mathsf{Ch}(s) \cap \mathsf{Ch}(s') = \emptyset$

Lemma 1. *Let s be statement, let τ be an operation such that $\mathsf{Stat}(\tau) = s$, and let ϕ be an LTL formula. If $\mathsf{Vis}(\tau, \phi)$, then $\mathfrak{Vis}(s, \phi)$.*

Proof. By definition of Vis in Table 1, Props$(\phi) \cap \mathsf{Write}(\tau) \neq \emptyset$. Also, because $\mathfrak{Write}$ satisfies Property 1, $\mathfrak{Write}(s) \subseteq \mathsf{Write}(\tau)$. Hence, Props$(\phi) \cap \mathfrak{Write}(s) \neq \emptyset$. The lemma then follows from the definition of $\mathfrak{Vis}$ in Table 1. □

Thus, $\mathfrak{Vis}(s, \phi)$ is true if s induces an operation τ whose application affects the truth value of ϕ, as such over-approximating the relation Vis.

The second relation we discuss is the *enables relation* $\mathsf{Enables}$. Let τ, τ' be operations. Then, $\mathsf{Enables}(\tau, \tau')$ states that application of τ to some state μ can cause τ' to become enabled, i.e. τ is enabled in μ while τ' is not, but in the state that results from applying τ to μ, τ' *is* enabled. The formal definition (in terms of precise read and write sets) occurs in Table 1, together with the definition of the *approximate enables relation* $\mathfrak{Enables}$ (in terms of approximate read and write sets). Relations $\mathsf{Enables}$ and $\mathfrak{Enables}$ are related by the following lemma, whose proof is analogous to that of Lemma 1 (omitted for reasons of space).

Lemma 2. *Let s, s' be statements, and let τ, τ' be operations such that $\mathsf{Stat}(\tau) = s$ and $\mathsf{Stat}(\tau') = s'$. If $\mathsf{Enables}(\tau, \tau')$, then $\mathfrak{Enables}(s, s')$.*

Thus, $\mathfrak{Enables}(s, s')$ is true if s induces an operation τ whose application can enable an operation τ' induced by s', over-approximating the relation $\mathsf{Enables}$.

The third relation we discuss is the *independence relation* Indep. Let τ, τ' be operations. Then, $\mathsf{Indep}(\tau, \tau')$ is true if the *independence conditions* in the left column of Table 2 hold for each state μ of the transition system: ENABLEDNESS states that independent operations cannot *disable* each other, while COMMUTATIVITY states that applying independent operations in either order results in the same state. In practice, checking the independence conditions in each state would be too much a computational burden. Therefore, as usual [4], Indep is defined heuristically (see Table 1 and the middle column of Table 2).

We approximate ENABLEDNESS with condition H^{en}_{Indep} given in Table 2, which is guaranteed to be true if ENABLEDNESS is true. The intuition behind it is that if an operation τ does not disable an operation τ', then τ cannot make a condition κ on which enabledness of τ' depends (i.e. $\kappa \in \mathsf{Read}(\tau')$) false. Similarly, COMMUTATIVITY is approximated with $H^{comm}_{\mathsf{Indep}}$ in Table 2. The intuition behind $H^{comm}_{\mathsf{Indep}}$ is that if the orders in which operations τ and τ' can be applied both lead to the same state, the changes that they bring about are disjoint, i.e. τ does not (partially) undo changes brought about by τ' and vice versa.

Definitions of H^{en}_{Indep} and $H^{comm}_{\mathsf{Indep}}$ (similar to those in [4]) are in terms of operations instead of statements: to be able to compute independences before actual model checking, we require the latter. Therefore, as before, we introduce the *approximate independence relation* $\mathfrak{InDep}$, in whose definition (see Table 1) the precise heuristics have been replaced by their approximate counterparts $H^{en}_{\mathfrak{InDep}}$ and $H^{comm}_{\mathfrak{InDep}}$ (see the right column of Table 2). Relations Indep and $\mathfrak{InDep}$ are related by the following lemma; its proof is analogous to that of Lemma 1.

Lemma 3. *Let s, s' be statements, and let τ, τ' be operations such that* $\mathsf{Stat}(\tau) = s$ *and* $\mathsf{Stat}(\tau') = s'$. *If* $\mathfrak{InDep}(s, s')$, *then* $\mathsf{Indep}(\tau, \tau')$.

We use $\mathsf{Dep}(\tau, \tau')$ (and $\mathfrak{Dep}(s, s')$) as a shorthand for "$\mathsf{Indep}(\tau, \tau')$ is false" (and "$\mathfrak{InDep}(s, s')$ is false"), and call τ, τ' (and s, s') *dependent*.

4 State Space Reduction

In a nutshell, the idea of state space reduction is as follows. Let $\mathcal{T} = \langle M, \mu_0, \longrightarrow \rangle$ be the *complete* transition system. The aim of reduction techniques is to find a *reduced* transition system $\widehat{\mathcal{T}} = \langle \widehat{M}, \mu_0, \widehat{\longrightarrow} \rangle$ such that $\widehat{M} \subseteq M$ and $\widehat{\longrightarrow} \subseteq \longrightarrow$. The idea is that $\widehat{M}$ and $\widehat{\longrightarrow}$ may be *significantly smaller* than M and $\longrightarrow$, and that investigating $\widehat{\mathcal{T}}$ will require less resources (time and memory) than inspection of $\mathcal{T}$ would. To ensure that model checking $\widehat{\mathcal{T}}$ for φ yields the same results as model checking $\mathcal{T}$, henceforth referred to as *correctness*, $\widehat{\mathcal{T}}$ should be both SOUND and COMPLETE with respect to $\mathcal{T}$ and φ [7]. Let $\boldsymbol{\Pi}$ be the set of computations in $\mathcal{T}$, and let $\widehat{\boldsymbol{\Pi}}$ be the set of computations in $\widehat{\mathcal{T}}$. Then:

SOUND — If $\boldsymbol{\pi} \in \boldsymbol{\Pi}$ s.t. $\boldsymbol{\pi} \models \neg\varphi$, then there exists a $\widehat{\boldsymbol{\pi}} \in \widehat{\boldsymbol{\Pi}}$ s.t. $\widehat{\boldsymbol{\pi}} \models \neg\varphi$.
COMPLETE — If $\widehat{\boldsymbol{\pi}} \in \widehat{\boldsymbol{\Pi}}$ s.t. $\widehat{\boldsymbol{\pi}} \models \neg\varphi$, then there exists a $\boldsymbol{\pi} \in \boldsymbol{\Pi}$ s.t. $\boldsymbol{\pi} \models \neg\varphi$.

In the remainder, we describe and define two reduction techniques, PBS and POR, in terms of the relations given in Sect. 3.3. We stress that these techniques by themselves and the ideas behind them are not new: both have extensively been studied in the context of imperative languages. Their coherent definition for agents in terms of the same relations, however, is a contribution of ours. This requires the following efforts. With respect to PBS, we redefine data structures used in traditional PBS in terms of relations given in Sect. 3.3. With respect to POR, we can straightforwardly apply the existing *ample set method*, which is already defined in terms of relations similar to those of Sect. 3.3; a novelty, however, is the introduction of a heuristic that generalises SPIN's [4].

4.1 Property-Based Slicing

The aim of *property-based slicing* (PBS) is to remove statements from the source code of the system to be verified that do not *influence* the (negated) property $\neg\varphi$. Removal of such statements may cause certain states and transitions to be eliminated from the transition system, thus yielding a reduction. A PBS algorithm is run *before* generation of the transition system commences (and *without* the need for generation of the complete transition system). The challenge of PBS is to remove as much code as possible while preserving correctness.

PBS algorithms represent the source code of the system under verification as a graph [15]. Such a graph makes explicit how execution of one statement can influence the execution of other statements as well as the property to be checked. Moreover, it enables the formulation of the PBS problem as a graph reachability problem. In our PBS algorithm, we use *influence graphs.* Informally, the influence graph with respect to a set of statements S (by which some program P is defined) and a (negated) property $\neg\varphi$ is a graph whose vertices are statements and $\neg\varphi$, and whose edges are elements of the visibility and enables relation.

Definition 3. *Let S be the set of statements by which some program P is defined, and let $\neg\varphi$ be a negated property. The influence graph $\mathcal{G}(S, \neg\varphi) = \langle \mathcal{N}, \mathcal{E} \rangle$ is a digraph with $\mathcal{N} = S \cup \{\neg\varphi\}$ and $\mathcal{E} = \{\, \langle s, \neg\varphi \rangle \in S \times \{\neg\varphi\} \mid \mathfrak{Vis}(s, \neg\varphi) \,\} \cup \{\, \langle s, s' \rangle \in S \times S \mid \mathfrak{Enables}(s, s') \,\}$.*

The first line of the definition of $\mathcal{E}$ represents the notion of *direct influence* on $\neg\varphi$: every edge $\langle s, \neg\varphi \rangle$ indicates that there exists an operation $\tau \in \mathsf{Ops}(s)$ that can influence the truth value of a proposition in $\neg\varphi$. The second line of $\mathcal{E}$'s definition represents the notion of *indirect influence* on $\neg\varphi$: every edge $\langle s, s' \rangle$ indicates that there exist operations $\tau \in \mathsf{Ops}(s)$ and $\tau' \in \mathsf{Ops}(s')$ such that τ can enable τ'. If s' influences $\neg\varphi$ (directly or indirectly), s influences $\neg\varphi$ indirectly.

Closely related to influence is the notion of *routes.* A route through an influence graph is a finite sequence of vertices $s_0 \cdots s_n \neg\varphi$, abbreviated $s_0 \rightsquigarrow \neg\varphi$, such that every statement occurs only once on a route, i.e. if $i \neq j$ then $s_i \neq s_j$ for all $0 \leq i, j \leq n$, and every route ends in $\neg\varphi$. The set of all routes through an influence graph $\mathcal{G}(S, \neg\varphi)$ is denoted by $Routes(\mathcal{G}(S, \neg\varphi))$. The idea central to our PBS algorithm is that every statement that is *not* on any route through the influence graph $\mathcal{G}(S, \neg\varphi)$ can safely be removed from the source code: these statements have no influence on the truth value of $\neg\varphi$. The algorithm takes a set of statements S as input, and computes a reduced set of statements $\widehat{S}$ by constructing an influence graph and computing routes. To determine if a route exists, the algorithm starts at a vertex s, and explores the influence graph until the vertex $\neg\varphi$ is reached, or no more reachable yet unexplored vertices are left.[7]

Existing PBS algorithms work in roughly the same way: the program is represented as a graph, reducing the PBS problem to graph reachability analysis. A key difference is that in our approach, the connection between operations and statements is made very explicit,[8] allowing for a rigid proof of correctness. We have not found similar explicit connections in the existing literature on PBS.

[7] Several optimisations may be implemented. For instance, if a depth-first exploration strategy is applied, all vertices on the depth-first stack at the moment $\neg\varphi$ is reached also have a route to $\neg\varphi$, making additional searches for these statements unnecessary.

[8] The visibility and enables relations ($\mathfrak{Vis}$ and $\mathfrak{Enables}$) are defined in terms of read and write sets on statements ($\mathfrak{Read}$ and $\mathfrak{Write}$), which are related to read and write sets on operations (Read and Write) by Property 1, which are defined in terms of individual transitions of the transition system.

Theorem 1. *Our PBS algorithm preserves soundness and completeness.*

Proof (Sketch). We adopt the premise that if a computation $\boldsymbol{\pi}$ satisfies $\neg\varphi$, i.e. $\boldsymbol{\pi} \models \neg\varphi$, then an operation that influences $\neg\varphi$ is applied during $\boldsymbol{\pi}$'s generation.

SOUNDNESS *If $\boldsymbol{\pi} \models \neg\varphi$ and by our premise, there exists a computation $\boldsymbol{\pi}'$ such that $\boldsymbol{\pi}' \models \neg\varphi$ and that is generated exclusively by applying influential operations. Hence, as the algorithm retains all statements that can induce influential operations, $\boldsymbol{\pi}'$ is also a computation in the reduced transition system.*

COMPLETENESS *Because the algorithm does not introduce new statements to the set S, no new transitions are introduced either.* $\square$

We note that the adopted premise in the previous proof is *false* if $\neg\varphi$ (in NNF) contains $\bigcirc$ or $\mathcal{R}$ operators: $\bigcirc\,\phi$ can be true without application of an influential operation if ϕ is already true in the current state, while $\phi\ \mathcal{R}\ \phi'$ can be true if ϕ' is true from the current state onwards without an influential operation ever being applied (i.e. ϕ never becomes true). Thus, the PBS algorithm is only applicable if $\neg\varphi$ is in the $\{\bigcirc, \mathcal{R}\}$-free fragment of LTL.

4.2 Partial Order Reduction

Next, we present a *partial order reduction* (POR) algorithm in terms of the relations of Sect. 3.3. POR algorithms try to exploit the observation that the various orders in which certain events can take place are irrelevant with respect to a certain property. Once such a situation is identified, a POR algorithm forces the model checker to choose only one *representative* order and to disregard all the others. While a PBS algorithm is applied prior to the generation of the reduced transition system, a POR algorithm is run *during* its generation (and *without* the need for generation of the complete transition system first).

There are various approaches to POR. Here, we focus on the *ample set method* [4] as it fits the relations of Sect. 3.3 seamlessly. The idea is to construct a reduced transition system by selecting only a subset of all enabled operations in each state (and disregarding the other enabled operations). To preserve correctness, such a subset, called an *ample set* and denoted by $\mathsf{Ample}(\mu)$, must satisfy the following:

C0 (Emptiness) $\mathsf{Ample}(\mu) = \emptyset$ iff $\mathsf{En}(\mu) = \emptyset$.

C1 (Ample Decomposition) In the complete transition system, on any path starting from some state μ, an operation dependent on an operation from $\mathsf{Ample}(\mu)$ cannot appear before some operation from $\mathsf{Ample}(\mu)$ is executed.

C2 (Invisibility) If $\mathsf{En}(\mu) \neq \mathsf{Ample}(\mu)$, all operations in $\mathsf{Ample}(\mu)$ are visible.

C3 (Cycle Closing) If a cycle contains a state in which an operation τ is enabled, then it also contains a state μ such that $\tau \in \mathsf{Ample}(\mu)$.

Details about these conditions are given in [4].

Let μ be a state. A naive implementation of the ample set method would be to check for all subsets of $\mathsf{En}(\mu)$ whether the four conditions are satisfied, and then pick one such subset as ample set. The problem with such an implementation, however, is that checking **C1** is computationally just as hard as the model checking problem for the complete transition system [4]. Therefore, in practice, rather than checking **C1** for an arbitrary subset of enabled operations, a heuristic approach that finds a set of operations that is *guaranteed* to satisfy **C1** is used. We call such a set a *candidate set*. Such an approach does not always lead to an ample set that yields the greatest reduction possible, but can be effective nevertheless. Once candidate sets are chosen, they need only be checked for **C0**, **C2**, and **C3**, which are easy to compute. Our idea for choosing candidate sets is to first select a subset of S, denoted by $\widehat{S}$, which satisfies the following:

Property 2. Let S be the set of statements defining a program. Then, for all $s' \in \widehat{S}$, there does not exist a $s \in S \setminus \widehat{S}$ s.t. (i) $\mathfrak{Dep}(s, s')$ and (ii) $\mathfrak{Enables}(s, s')$.

Once a set $\widehat{S}$ satisfying Property 2 is found, the set of all enabled operations in a state μ that can be induced by a statement $s \in \widehat{S}$ is selected as a candidate set C, i.e. $C = \mathsf{En}(\mu) \cap \bigcup_{s \in \widehat{S}} \mathsf{Ops}(s)$. It is guaranteed that C satisfies **C1**.

Lemma 4. *If $\widehat{S}$ satisfies Property 2, $C = \mathsf{En}(\mu) \cap \bigcup_{s \in \widehat{S}} \mathsf{Ops}(s)$ satisfies* **C1**.

Proof (Sketch). There are two situations in which **C1** *may be violated, which differ by whether τ is induced by a statement s' outside $\widehat{S}$ or in it. In the former case, if $s' \notin \widehat{S}$, there exists a statement in $\widehat{S}$ on which s' depends (because τ is dependent on an operation in C). This situation is covered by condition (i) of Property 2. In the latter case, if $s' \in \widehat{S}$, then τ is not enabled in the current state (because $\tau \notin C$). Hence, there exists another statement s that enables s'. If $s \notin \widehat{S}$, then $\mathfrak{Enables}(s, s')$, hence this situation is covered by condition (ii) of Property 2. Otherwise, if $s \in \widehat{S}$, the previous argument can be applied inductively.* □

In practice, the challenge is finding suitable sets $\widehat{S}$ as efficiently as possible. A straightforward approach is iterating over all elements in the power set of S, and checking Property 2 for each $\widehat{S} \in 2^S$. However, as this requires time exponential in the number of statements, this is not a good idea. Instead, we let the search for sets $\widehat{S}$ be guided by the definition of $\mathfrak{Dep}$: we search for sets $\widehat{S}$ that are guaranteed to satisfy (i) of Property 2. This search can be done in time linear in the number of statements $|S|$ and the size of $\mathfrak{Dep}$, and yields at most $|S|$ sets $\widehat{S}$ instead of $2^{|S|}$ for which (ii) of Property 2 need be checked. The idea is to regard the relation $\mathfrak{Dep}$ as a graph whose vertices are statements and whose edges are elements of the relation. Because every edge is an element of $\mathfrak{Dep}$, each statement belonging to a set $\widehat{S}$ cannot have edges to statements outside $\widehat{S}$: a set $\widehat{S}$ satisfying (i) of Property 2 corresponds to a *connected component* in the graph, which can be found with a depth-first search [9]. Such a search runs in time linear in the number of vertices and edges. As there cannot be more connected components than vertices, this approach yields at most $|S|$ sets $\widehat{S}$. The previous comprises the key difference with SPIN's POR implementation: in SPIN, sets $\widehat{S}$ satisfying Property 2 are always singletons. We have generalised this with an approach that reduces the problem to finding connected components. Note that our approach's applicability is not limited to agents, but extends to, for instance, SPIN as well.

The POR algorithm is run each time successors of a state μ are required during model checking. It first computes sets of operations satisfying **C1** as outlined above and then performs simple checks for **C0**, **C2** (using $\mathfrak{Vis}$), and **C3**. If no set satisfying all conditions can be found, all successors in μ are returned. Like all POR algorithms, the algorithm described is applicable only if the property under investigation is in the *stuttering invariant* subset of LTL: it may not contain $\bigcirc$ operators. Also, it is compatible with on-the-fly model checking, provided the remarks made in [8] are taken into account.

Theorem 2. *Our POR algorithm preserves soundness and completeness.*

Proof (Sketch). The algorithm is, essentially, the algorithm in [4] with a different approach to generating **C1**. *Soundness and completeness thus follow from the ample set method's correctness as proven in Sect. 10.6 of [4] and Lemma 4.* $\square$

5 Implementation & Experience

We have implemented the algorithms discussed in the previous section as extensions to the *interpreter-based* Goal model checker introduced in [11]. The idea of the interpreter-based approach to agent verification is to implement model checking algorithms on top of an existing agent interpreter. An alternative approach is to encode the semantics of the agent language in a format that an existing model checker can process and to use this existing model checker for actual verification. Interpreter-based model checking, however, has been shown to consume less resources and offers immediate language support without the need for complex translations [11].

With respect to the implementation of reduction techniques, the interpreter-based approach has another benefit: the model checking algorithms implemented on top of the existing agent interpreter can easily be extended with implementations of reduction algorithms. In contrast, if existing model checkers are used for agent verification, such extensions are likely to be less straightforward to implement. As a result, one is bound to use reduction techniques that ship with the existing model checker, but that are not tailored to the agent language that the agent program is written in. It has been shown [2] that generic reduction algorithms may not work well on translated agent programs.

The PBS and POR algorithm discussed are defined in terms of the same relations on operations. From a software engineering point of view, the implementation of these techniques benefits from this in two ways: SHARED-CODE-BASE and RUNTIME-SYNERGY.[9]

SHARED-CODE-BASE — We implemented a library for analysis of action rules and computation of the visibility, enables, and (in)dependence relation. The implementations of the PBS and POR algorithms both use this library.

RUNTIME-SYNERGY — Computation of the visibility, enables, and dependence relation occurs at most once each verification run. Subsequently, the PBS and POR implementations can both use the results of these computations; no duplicate calculations are performed.

[9] Note we address the recommendation of [14] that research in state space reduction should not only focus on new techniques, but also on combining existing ones.

To investigate whether our PBS and POR algorithms are able to significantly reduce resource consumption, we have carried out several small experiments involving non-deterministic single-agent systems. In what we call the *blender experiments*, we have investigated an agent whose task is to put bananas and oranges into a blender to make juice. In the *blocks counter experiments*, the subject of verification is an agent that breaks down towers of blocks, while counting to some natural number. Finally, in the *wumpus experiments*, we have model checked agents that must navigate through an unknown maze in search of a heap of gold, while avoiding bottomless pits and a vicious cave animal: the wumpus. With these experiments, we aim at investigating whether PBS and POR algorithms for agent languages like GOAL have the same potential as in traditional model checking. Below, we give a synopsis; details appear in [10].

With respect to PBS, the blender and blocks counter experiments show that the reduction can be significant: the measured decrease of the state space ranged from 75% to 99%, the reduction in runtime (including PBS computation) ranged from 43% to 97%, and the measured reduction in memory consumption (including PBS computation) ranged from 25% to 88%. However, in the wumpus experiment, a reduction in resource consumption was not achieved: in fact, the entire verification procedure took longer to finish with PBS enabled than without PBS, although the difference was less than three seconds for the most complex wumpus agent. The reason is that a wumpus agent's tasks (exploring the cave, grabbing the gold, hunting the wumpus) all influence each other, i.e. all action rules are on a route in the influence graph. Consequently, no action rules are removed by the PBS algorithm, hence no reduction is obtained, despite the spending of resources on its computation. A prerequisite for the PBS algorithm to yield a reduction is, thus, that the property under investigation concerns a task of the agent that is not influenced by its other tasks. This prerequisite is satisfied by the agents in the other two experiments: putting bananas in a blender does not influence putting oranges in a blender (and vice versa), and deconstructing a tower does not influence counting (and vice versa).

Similar to the PBS results, our blender and blocks counter experiments with POR show that this technique can yield significant reductions, particularly if the agent under consideration is (i) *loosely coupled*, meaning that there are few dependencies between the different tasks that it needs to carry out (the case in the blocks counter experiments),[10] or (ii) significantly underspecified (the case in the blender experiments). While the former has already been pointed out in existing POR literature, the latter seems specific to the application of POR to agents, as underspecification in imperative languages is rare. Using POR, the measured reduction of the state space ranged from 59% to over 99%, the reduction in runtime (including POR computation) ranged from 34% to 98%, and the measured reduction in memory consumption (including POR computation) ranged from 8% to 50%. As the agents in the wumpus experiments are neither loosely coupled

[10] This is a stronger requirement than the PBS prerequisite regarding influence, because influence is a directed relation (e.g. A can influence B, while B does not influence A), while dependence is undirected (e.g. A depends on B iff B depends on A).

nor underspecified, no reduction is obtained using POR. We speculate that non-deterministic agent programs are, in general, tighter coupled than concurrent imperative systems. Therefore, POR may be less often applicable in an agent context than in traditional model checking. Further investigations are, however, necessary to confirm or disprove this conjecture.

6 Related Work & Conclusion

Related work Both PBS and POR have extensively been studied in traditional model checking. An extensive survey with many references is given in [14]. Here, we focus on state space reduction techniques for agent model checking.

To the best of our knowledge, PBS has been studied in an agent context only by Bordini et al. [2, 3], who have designed a PBS algorithm for AgentSpeak systems. Their algorithm is based on earlier work on slicing logic programs [16], because *plans* in AgentSpeak are similar to guarded clauses in logic programming. The algorithm of Bordini et al. slices AgentSpeak programs by removing such plans from agents, and is, like other PBS algorithms, based on a graph representation of the program. An important difference between Bordini et al. and our work is that we have defined our PBS algorithm generically, i.e. not tailored to any specific APL. However, we do not consider our effort a generalisation of Bordini et al., because we have not based our PBS algorithm on [16] or [2, 3]. Instead, we see our work as a second and independent attempt to applying PBS to agents; it would be interesting to instantiate our framework for AgentSpeak, and compare the performance of the algorithm of Bordini et al. to ours.

To the best of our knowledge, POR has only been studied in an agent context by Lomuscio et al. [12]. While both our work and the work of Lomuscio et al. are based on the ample set method and applied in a context in which a depth-first strategy is used for the generation of the transition system, our approach differs in a number of ways. Most notably, [12] focuses on the verification of *models* of agent-based systems, while we consider verification of actual agent *programs*. Other work in the latter direction is the AIL framework [5] and its model checker AJPF [1]; a comparison between the aforementioned interpreter-based model checker for GOAL and AJPF appears in [11].

Conclusion We have introduced a framework, based on operations on mental states of agents, that facilitates the definition and implementation of the existing PBS and POR techniques in a unifying way. We have argued that existing state space reduction algorithms do not fit agent programs seamlessly due to the different nature of mental states (compared to variable assignments), and proposed a solution. The resulting definition of read and write sets for agents is the heart of our framework. With these and the relations defined in terms of them, in principle, we can readily reuse existing reduction algorithms. Nevertheless, we have also advanced the theory of PBS and POR to some extent: with respect to PBS, we have a very explicit connection between the algorithm and the transition system (absent in previous contributions), while with respect to POR, we have introduced an alternative heuristic to be used for ample set computation (Property 2). Finally, by defining two different techniques in terms of the same relations, we gain implementation benefits: shared code-base and runtime synergy.

We identify three directions for future work: (i) expanding our experience with both techniques to gain a better understanding of when their application can be beneficial and to what extent, (ii) instantiating the framework for multi-agent systems, and (iii) extending the framework to open systems.

References

1. R. Bordini, L. Dennis, B. Farwer, and M. Fisher. Automated verification of multi-agent programs. In *Proc. of ASE*, pages 69–78, 2008.
2. R. Bordini, M. Fisher, W. Visser, and M. Wooldridge. State-space reduction techniques in agent verification. In *Proc. of AAMAS*, pages 896–903, 2004.
3. R. Bordini, M. Fisher, M. Wooldridge, and W. Visser. Property-based slicing for agent verification. *Journal of Logic and Computation*, 19(6):1385–1425, 2009.
4. E. Clarke, O. Grumberg, and D. Peled. *Model checking*. The MIT Press, 2000.
5. L. Dennis, B. Farwer, R. Bordini, M. Fisher, and M. Wooldridge. A common semantic basis for BDI languages. In M. Dastani, A. E. F. Seghrouchni, A. Ricci, and M. Winikoff, editors, *Programming Multi-Agent Systems*, volume 4908/2008 of *LNCS*, pages 124–139. 2008.
6. K. Hindriks. Programming rational agents in GOAL. In A. Seghrouchni, J. Dix, M. Dastani, and R. Bordini, editors, *Multi-Agent Programming*, chapter 4, pages 119–157. 2009.
7. G. Holzmann. *The SPIN model checker*. Addison-Wesley, September 2003.
8. G. Holzmann, D. Peled, and M. Yannakakis. On nested depth first search. In *The SPIN Verification Systems*, volume 32 of *DIMACS*, pages 23–31. 1997.
9. J. Hopcroft and R. Tarjan. Efficient algorithms for graph manipulation. Technical Report STAN-CS-71-207, Stanford University, March 1971.
10. S.-S. Jongmans. Model checking GOAL agents. Master's thesis, Delft University of Technology, August 2010. Available at `http://repository.tudelft.nl`.
11. S.-S. Jongmans, K. Hindriks, and M. van Riemsdijk. Model checking agent programs by using the program interpreter. In J. Dix, J. Leite, G. Governatori, and W. Jamroga, editors, *CLIMA*, volume 6245/2010 of *LNCS*, pages 219–237. 2010.
12. A. Lomuscio, W. Penczek, and H. Qu. Partial order reductions for model checking temporal epistemic logics over interleaved multi-agent systems. *Fundamenta Informaticae*, 101(1-2):71–90, 2010.
13. A. Lomuscio and F. Raimondi. MCMAS: A model checker for multi-agent systems. In H. Hermanns and J. Palsberg, editors, *TACAS*, volume 3920/2006 of *LNCS*, pages 450–454. 2006.
14. R. Pelanek. Fighting state space explosion: review and evaluation. In D. Cofer and A. Fantechi, editors, *Formal Methods for Industrial Critical Systems*, volume 5596/2009 of *LNCS*, pages 37–52. 2009.
15. F. Tip. A survey of program slicing techniques. Technical Report CS-R9438 1994, CWI, 1994.
16. J. Zhao, J. Cheng, and K. Ushjima. Literal dependence net and its use in concurrent logic programming environment. In *Proc. of the Workshop on Parallel Logic Programming*, pages 127–141, 1994.