PBMC: Symbolic Program Slicing for Program Verification

Habib Saissi †, Péter Bokor ‡, Neeraj Suri †

†Technische Univeristät Darmstadt
‡Bosch/Siemens Home Appliances, Berlin
{saissi, pbokor, suri}@deeds.informatik.tu-darmstadt.de

Abstract. This paper proposes a novel optimization of bounded model checking (BMC) for better run-time efficiency. Specifically, we define *projections*, an adaptation of dynamic program slices, and instruct the bounded model checker to check projections only. Given state properties over a subset of the program's variables, we prove that the proposed optimization preserves the soundness of BMC.
We propose a symbolic encoding of projections and implement it for a prototype language of concurrent programs. We have developed a tool called PBMC to evaluate the efficiency of the proposed approach. Our evaluation with various concurrent programs justifies the potential of projections to efficient verification.

1 Introduction

Automated verification of complex programs is known to be a hard problem. The complexity of the task grows exponentially when the considered programs exhibit concurrent behavior [28]. Bounded model checking (BMC) [8] is a widely used verification technique, e.g., [12]. In BMC, a formula encoding the behavior of the program is computed and passed to an SMT/SAT solver along with the negation of the property. The solver then checks whether there exists an execution leading to a state violating the property. Thanks to the recent advances in the field of SAT solving, bounded model checking is becoming a practical solution for sound verification of concurrent programs [1, 2, 6]. The efficiency can be greatly improved by constraining the search space depending on the property of interest. Concretely, the encoding of the program is constrained by excluding behavior that is irrelevant or redundant with respect to the property. For example, the solver can be instructed to partially order (instead of totally order) transitions of the program, e.g., [6].

In this paper, we propose *projections* to constrain the search space of a bounded model checker. Conceptually, projections are slices of a program with respect to a set of variables. They are especially useful for the analysis of concurrent programs. For example, a projection with respect to the local state of a process may exclude transitions of other non-interfering processes. As a result, the interleavings of the excluded transitions (exponential in the number

of transitions) do not have to be considered by the solver. Intuitively, projections consist of executions which only contain transitions directly or indirectly affecting the variables of interest.

Contributions. Our first contribution is that we introduce projections, an adaptation of program slices to general transition systems, and show that they preserve the relevant safety properties of a program. The idea of projections is a general one and it is independent of how the program states are explored. Our second contribution is that we present a symbolic encoding of projections for concurrent programs; we call the encoding PBMC (projection-based BMC) because it can be used for efficient bounded model checking. Note that although we concentrate on concurrent programs our result equally holds for single-threaded programs. Interestingly, PBMC can be seen as a form of *dynamic program slicing* [5]. In contrast to existing program analysis approaches where static program slicing is applied prior to the actual analysis, e.g. [12,14,26][1], PBMC enumerates slices on-the-fly. The resulting slices are as precise as the dynamic ones because they are calculated based on feasible executions. To the best of our knowledge, PBMC is the first application of dynamic program slicing with BMC. Our final contribution is that we implement a prototype of PBMC and use it to verify simplified versions of a set of concurrent programs where program slicing in its traditional form fails to reduce the size of the program. The experiments show substantial verification time reductions compared to traditional BMC.

The paper is organized as follows. Section 2 and 3 provide a motivating example and discusses the related work. In Section 4, we formalize and prove the correctness of projections. Section 5 describes the symbolic encoding of projections and Section 6 shows our evaluation results.

2 Motivating Example

We motivate our approach using a simple example. Consider the program shown in Figure 1a consisting of three concurrent and sequential processes sharing different variables and an array B. We label the instructions on the different program locations l_1, l_2, l_3 and l_4. Assume that we are verifying a property which involves only the variable y. That is, we are interested in the values that y can take in any possible run of the program. There are 12 possible interleavings of the instructions. Assuming that initially $a = 0, b = 1, x = 0$ and $B[k] = 10+k$ for every k-th position in the array, we list in Figure 1b the possible runs which may be relevant. Every transition in the runs corresponds to a concrete execution of an instruction. For instance, t_5 and t_2 are two different transitions, but correspond to the same instruction l_4. When analyzing the program statically[2], one can only reason in terms of instructions. By doing that, it is clear that the program contains three *"dependencies"*: The execution of l_3 always *depends on* that of l_1 because l_1 always writes to variable a which l_3 reads from. For the same

[1] Please refer to Section 3 for more details.

[2] Ignoring our assumption about the initial values of the variables.

reason, the execution of l_4 depends on l_3. More interestingly, the execution of l_3 depends on l_2 only if $a = b$. This corresponds in Figure 1b to the sequence σ_3, as transition t_3 writes in a position in B that t_7 reads from. We refer to σ_1, σ_2 and σ_3 as *projections* of the program on the set of variables $\{y\}$. Intuitively, in order to preserve all the possible values that y can take, it is enough to consider sequences where every transition t either writes to y, or there is another transition after it which also writes to y and transitively depends on t. For example in σ_3, transition t_3 is included because it influences t_7 which in turn influences t_8. Transition t_8 is kept in the projection since it writes to y. In general, projecting a run on a set of variables F means that we keep every transition that writes to variables in F or affects another transition after it already included in the projection.

$P_1\{$ $\quad$ $P_2\{$ $\quad$ $P_3\{$

$\quad l_1 : a = 1;$ $\quad$ $\quad l_2 : B[b] = 5;$ $\quad$ $\quad l_3 : x = B[a];$

$\quad \}$ $\quad$ $\quad \}$ $\quad$ $\quad l_4 : y = x + 7;$

$\quad$ $\quad$ $\quad$ $\quad$ $\quad \}$

(a) An example concurrent program.

σ_1	σ_2	σ_3
$t_1 : x = 10$	$t_3 : a = 1$	$t_3 : a = 1$
$t_2 : y = 17$	$t_4 : x = 11$	$t_6 : B[1] = 5$
	$t_5 : y = 18$	$t_7 : x = 5$
		$t_8 : y = 12$

(b) 3 projections on y out of 12 possible runs.

Fig. 1: A motivating example

Given that the safety property of interest involves a subset of variables, the verification time of such programs can be greatly reduced if we constrain the exploration to projected executions. We use BMC to symbolically describe the behavior of the program and add constraints that characterize projections. By doing so, we constrain the search space of the model checker but still preserve all possible values that the variables in the property can take. If a state violating the property is reachable, a projection is returned. Furthermore, as projections contain only relevant instructions, they are easier to interpret and analyze.

Note that static program slicing techniques [30] applied to this program would not help in the reduction of the search space as it will return a copy of the whole program. This is due to the conditional dependencies of instructions accessing arrays and the concurrency, in which case it's not clear before execution whether an instruction affecting a variable of interest v will be actually executed before another instruction assigning a value to v. For instance, it's not clear whether l_1 will be executed before l_3. Also, if the values of a and b depend on some non-deterministic behavior, e.g., concurrency or user input, it might not be possible

to predict whether $a = b$. In this case, static program slicing conservatively over-approximate the slices.

3 Related Work

Program Slicing. PBMC is the first approach to combine dynamic slicing and BMC. Our proposed technique is based on a notion similar to program slicing [30]. Static program slicing has been used to reduce the size of programs under verification, e.g., [12, 14, 26]. In these approaches a slice of the program is computed and passed to the model checker. However, the returned slice is an over-approximation of the instructions that are in fact relevant to the slicing criterion. This is due to the fact that inferring dependencies statically, a prerequisite for deriving slices, is hard with the presence of concurrency [21, 26]. In PBMC, formulas describing precisely when dependencies occur are generated passed to a solver along with the verification formula. In dynamic slicing [5], the program is run with concrete input and the slicing is done directly on the execution path. Although the slices returned by dynamic slicing techniques are accurate, they only concern the considered execution path. To use dynamic slicing with verification, one would have to enumerate all paths which contradicts the purpose of using slicing for reducing the number of paths to be explored. The slicing in PBMC is dynamic since only reachable, and therefore feasible, paths are sliced.

Partial-Order Reduction. A common technique against state space explosion is partial-order reduction (POR) [17]. Whereas POR's reduction comes from executing commutative transitions in one representative order, in PBMC it is based on the notion of conflicting read/write operations. Existing POR semantics, however, do not subsume projections. Widely-known POR semantics such as stubborn sets [9, 28], persistent sets [17], and ample sets [13] guarantee preserving all deadlocks of the program. In some cases [17], the preservation of deadlocks also entails that of local states. On the other hand, projections do not necessary preserve all deadlocks nor all local states. Existing POR techniques include [4, 9, 15, 18, 20, 29] among others.

BMC. An interesting way of combining slicing with BMC is described in [16]. Tunneling and slicing based reduction makes use of slicing to decompose a BMC formula into disjoint smaller instances covering subsets of the program. These formulas are constructed such that the original formula is satisfiable only if at least one of the smaller instances can be satisfied. Nevertheless, the used slicing is static and therefore is imprecise.

Our approach for reduction is similar to MPOR [20], where constraints are added to the BMC formula to guide the search. The constraints used in this approach are based on Mazurkiewicz's traces [23], the underlying semantics for most POR theories. Our symbolic encoding of the transition system enhances the encoding used in MPOR. Furthermore, POR and projections are orthogonal techniques that can be used in combination for better reductions as demonstrated

in [14]. This is the case because the definition of path projections alone still allows for two Mazurkiewicz equivalent paths to be considered in the search. We argue, that our encoding can be augmented by the constraints of MPOR for better performance. To see this consider two executions t_4, t_1, t_2, t_3 and t_2, t_1, t_4, t_3 such that t_3 depends on both t_1 and t_2. From both executions we can derive projections t_1, t_2, t_3 and t_2, t_1, t_3, respectively. Since there is no dependency between t_2 and t_1, both projected executions are Mazurkiewicz equivalent. It follows then that it is sufficient to consider one of the projections.

Encodings. We adapted the encoding used in [20] which does not require unwinding of loops as in [12], [10] or [27]. Yet, unwinding loops may be beneficial and allow different encoding, e.g., using single static assignment form to reduce the number of variables in the formula. The idea behind projections is independent of the used encoding and therefore can be adapted for use with other BMC formulas. For instance in CBMC [6], transitions are associated with clock variables that reflect how they are (partially-)ordered. Intuitively, a path corresponds to a partial order over transitions where only dependent ones are strictly ordered. Thus, constraints are used to enforce a total order on the dependent transitions. Using such an encoding, the model checker might still explore some partial orders which are not relevant to the property. Hence, two dependent transitions will still have to be ordered although they might not have any influence on the property. Given a subset of variables, we argue that projection constraints can be added to such an encoding to further reduce the number of interleavings of dependent transitions. This can be done by constraining the used read-from relation according to the definition of projections.

Other Symbolic Approaches. Another possibility is to use slicing on-demand to refine the search for assertion violations. For instance, Path slicing [19] is a technique that has been implemented within the Blast model checker [7] which makes use of counterexample guided refinement techniques [11]. In Blast, slicing is used to simplify the counterexample analysis phase that serves the purpose of refining the search. Our approach is different from path slicing in the sense that the search for bugs is constrained from the beginning using projections to guide the solver toward feasible counterexamples. PBMC, and BMC based approaches in general, are fundamentally different from Blast, and other tools such as [22, 24, 25, 29], where the verification formulas are generated and refined incrementally with the help of the solver. In BMC, a single formula describing the whole program is computed statically and the exploration work is deferred to the SMT solver. A comprehensive discussion of the advantages and disadvantages of incremental generation and refinement of the verification formula over BMC approaches is beyond the scope of this work.

4 Property Preservation with Projections

4.1 System Model

We abstract programs by general transition systems, where a transition may read and/or write a set of variables.

General Transition Systems. Formally, the system is defined as a tuple $TS = (S, S_0, T)$ where S, $S_0 \subseteq S$, and $T \subseteq S \rightharpoonup S$ are the set of states, initial states, and the set of *transitions*, respectively. In the rest of paper, we will always write s_0 to refer to an initial state.

A program defines a list of atomic instructions (e.g., lines of code). In every state, the program can execute a transition, corresponding to an instruction, and deterministically move to a unique successor state. Formally, a transition $t \in T$ is a partial function such that $t(s) = s'$ iff t can be executed in s and it leads to state s'. In that case, we say that t is enabled in s. For convenience, we also write $s \xrightarrow{t} s'$ if $t(s) = s'$.

A finite path σ in the transition system TS is a sequence $s_0 \xrightarrow{t_1} s_2 \xrightarrow{t_2} \ldots \xrightarrow{t_n} s_n$, also written as $s_0 \xrightarrow{t_1,t_2,\ldots,t_n} s_n$, such that $t_{i+1}(s_i) = s_{i+1}$ for all $0 \leq i < n$. In that case, we write $\sigma \in TS$.

In addition, we assume that every state s assigns a value $s(v)$ to every $v \in V$, where V is the set of variables. Given a set $F \subseteq V$, we refer to the values assigned by s to variables $v \in F$ by $s(F)$. We write $s(F) = s'(F)$ for two states s and s', if for all $v \in F$, $s(v) = s'(v)$.

Dependency Relation. The execution of a transition involves reading from and writing to a subset of variables. We assign to every transition t a *read/write set* of variables, denoted as $r(t)/w(t) \subseteq V$ respectively. A transition t is said to read from (write to) a variable v if $v \in r(t)$ ($v \in w(t)$). The read set of a transition contains all variables that may have an influence on whether a transition is enabled and the outcome of its execution. On the other hand, the write set of a transition consist of the variables that it might modify.

Formally, $w(t)$ is defined such that for every $v \in V$, $v \in w(t)$ iff there are $s, s' \in S$ such that $s \xrightarrow{t} s'$ and $s(v) \neq s'(v)$. Note that the write set of a transition does not include a variable it never modifies. We define the read set $r(t)$ as the smallest set such that for every $s, s' \in S$ and $v \in r(t)$ such that $s(v) = s'(v)$ then,

- t is enabled in s iff t is enabled in s', and
- if t is enabled in s then for every $v' \in w(t)$, $t(s)(v') = t(s')(v')$.

We define a dependency relation $D \subseteq T \times T$ to model any interference between transitions. A transition t depends on a transition t' if t reads from a variable that t' writes to. In that case, we also say that t influences t' and write $(t, t') \in D$.

Definition 1 (Dependency Relation) *Given two transitions t and $t' \in T$, we say that t' depends on t and write $(t, t') \in D$ iff $r(t') \cap w(t) \neq \emptyset$.*

Note that two transitions only writing to the same variable are not considered to be dependent as the execution of one of them before the other does not influence the behavior of latter.

4.2 Projections

In this Section, we propose the projection semantics and present a theorem that guarantees that preserving projections on a set of variables is a sufficient condition for preserving properties defined over those variables.

First, we give a formal definition of path projections on a set of variables. Intuitively, a projection of a path σ on a set of variables F is a sequence of transitions containing every transition t that either writes into a variable in F, or there is a transition t' after it such that t' depends on t and t' is also in the projection.

Definition 2 (Projection) *Given a set of variables $F \subseteq V$ and a path $\sigma = s_0 \xrightarrow{t_1,...,t_n} s_n$, $\sigma|_F = t_{j_1}, t_{j_2}, ..., t_{j_k}$ is said to be a projection of σ on F, if $1 \leq j_1 < j_2 < ... < j_k \leq n$ and for all $1 \leq i \leq n$, $i \in \{j_1, j_2, ..., j_k\}$ iff:*

(a) $w(t_i) \cap F \neq \emptyset$, or
(b) there exists $j \in \{j_2, j_3, \ldots, j_k\}$ such that $i < j$ and $(t_i, t_j) \in D$.

Property Preservation. Our main result is that projections can be used to constraint the search space of a model checker. For that purpose we must guarantee that projections on a set of variables preserve the properties of those variables. Given two transition systems TS, TS' and a set of variables $F \subseteq V$, for every path $\sigma = s_0 \xrightarrow{t_1,t_2,...,t_n} s_n \in TS$ there exists a path $\sigma' = s_0 \xrightarrow{t'_1,t'_2,...,t'_m} s'_m \in TS'$ such that $s_n(F) = s'_m(F)$. We then say that TS' preserves the properties of F in TS. Furthermore, given two transition systems TS and TS', we say that TS' *preserves* the projections of F in TS, if for every path $\sigma = s_0 \xrightarrow{t_1,t_2,...,t_n} s_n \in TS$ there exists a path $\sigma' = s_0 \xrightarrow{t'_1,t'_2,...,t'_m} s_m \in TS'$ such that $\sigma|_F = t'_1, t'_2, ..., t'_m$.

Theorem 1 (Property Preservation) *Let TS and TS' be two transition systems and $F \subseteq V$ a set of variables. If TS' preserves the projections of F in TS then it also preserves the properties of F in TS.*

We prove Theorem 1 via a series of lemmas. First, we introduce the following auxiliary definitions: Let t_i, t_j be two transitions, $\alpha = t_1 t_2 \ldots t_n$ a sequence of transitions and $\sigma = s_0 \xrightarrow{\alpha} s_n$ be the resulting path. For convenience, we will write $t_i \in \sigma$ and $t_i \in \alpha$ if $i \in \{1, 2, \ldots, n\}$. Furthermore, if $j \in \{i+1, \ldots, n\}$, we write $t_i <_\sigma t_j$ or $t_j >_\sigma t_i$.

First, we show that between two successive transitions in a projection $\sigma|_F$, the values assigned to F and the variables read by any transition in $\sigma|_F$ after the second transition remain unmodified by all transitions outside the projection.

Lemma 1 *Let $\sigma = s_0 \xrightarrow{t_1,t_2,...,t_n} s_n$ be a path, $F \subseteq V$ a set of variables and t_k, $t_{k'}$ two transitions such that $t_k <_\sigma t_{k'}$ and for every $t_{k-1} <_\sigma t_i <_\sigma t_{k'+1}$ $t_i \notin \sigma|_F$. For every $t_{k-1} <_\sigma t_j <_\sigma t_{k'+1}$, $t_q >_\sigma t_{k'}$ such that $t_q \in \sigma|_F$ and $v \in r(t_q)$, $s_j(F) = s_{k-1}(F)$ and $s_j(v) = s_{k-1}(v)$.*

Proof. Let σ be a path, $\sigma|_F$ its projection on a variable set $F \subseteq V$, and two transitions t_k and $t_{k'}$ as described above. We know that for all $t_{k-1} <_\sigma t_j <_\sigma t_{k'+1}$, $w(t_j) \cap F = \emptyset$. Otherwise, t_j would be included in $\sigma|_F$ between t_k and $t_{k'}$ (Def. 2). This means that $s_{j-1}(F) = s_j(F) = s_{k-1}(F)$. Given a transition $t_q \in \sigma|_F$ such that $t_q >_\sigma t_{k'}$, we assume that there is a variable $v \in r(t_q)$ such that for a $j \in \{k, \ldots, k'\}$, $s_j(v) \neq s_{k-1}(v)$. Let j be the first such an index. This implies that $s_{k-1}(v) = s_{j-1}(v) \neq s_j(v)$. We then have $v \in w(t_j)$ and therefore $r(t_q) \cap w(t_j) \neq \emptyset$. From Definition 1 it follows that $(t_j, t_q) \in D$. Consequently, t_j should also be included in $\sigma|_F$. This contradicts our initial assumption.

With the help of Lemma 1 we show that every projection is also a path and that it reaches a state where the assigned values to variables in F are the same as in the original path.

Lemma 2 *Let $\sigma = s_0 \xrightarrow{t_1, t_2, \ldots, t_n} s_n$ be a path and $\sigma|_F = t_{j_1}, t_{j_2}, \ldots, t_{j_k}$ its projection on variable set $F \subseteq V$. Then there exists a path $\sigma' = s_0 \xrightarrow{t_{j_1}, t_{j_2} \ldots, t_{j_k}} s'_k$ such that $s_n(F) = s'_k(F)$.*

Proof. We separately consider the case where $\sigma|_F$ is empty, i.e. contains no transition. In this case, we have for every $t_i \in \sigma$, $t_i \notin \sigma|_F$. From Lemma 1, it follows that $s_0(F) = s_n(F)$ and σ' exists as an empty path.

Now we assume that $\sigma|_F$ contains at least one transition. We start by proving, for every $1 \leq q \leq k$, the existence of the path that consists of the first q transitions of $\sigma|_F$, that $s'_q(F) = s_{j_q}(F)$ and that for every $v \in r(t_j)$ such that $t_j >_\sigma t_{j_q}$ and $t_j \in \sigma|_F$, $s'_q(v) = s_{j_q}(v)$. The proof is an induction on the number of the first q transitions in the projection. Consider the case of the first transition t_{j_1} in the projection. Since $\forall i \in \{1, \ldots, j_1 - 1\}$ $t_i \notin \sigma|_F$, we know that $s_0(F) = s_{j_1 - 1}(F)$, and that for every $v \in r(t_{j_1})$, $s_0(v) = s_{j_1 - 1}(v)$ (Lemma 1). Thus, since t_{j_1} is enabled in $s_{j_1 - 1}$, it is also enabled in s_0, and there is a state s'_1 such that $s_0 \xrightarrow{t_{j_1}} s'_1$ and $s'_1(v) = s_{j_1}(v)$ for every $v \in w(t_{j_1})$ (read set definition). Since for every $v \in F$ such that $s_0(v) \neq s'_1(v)$ is in $w(t)$ (write set definition), it follows that $s'_1(F) = s_{j_1 - 1}(F)$. We assume now that the property holds for the first q transitions and prove it after considering the $q{+}1$-th transition $t_{j_{q+1}}$. From Lemma 1 it follows then that $s_{j_q}(F) = s_{j_{q+1} - 1}(F)$ and $s_{j_q}(v) = s_{j_{q+1} - 1}(v)$ for every $v \in r(t_j)$ such that $t_j > t_{j_q}$ and $t_j \in \sigma|_F$. Using the induction assumption it follows then that $s'_q(F) = s_{j_{q+1} - 1}(F)$ and $s'_q(v) = s_{j_{q+1} - 1}(v)$. Consequently, t_{q+1} is enabled in s'_q and there exists a state s'_{q+1} such that $s'_q \xrightarrow{t_{q+1}} s'_{q+1}$, $s'_{q+1}(F) = s_{j_{q+1}}(F)$ and $\forall v \in w(t_{j_{q+1}})$, $s'_{q+1}(v) = s_{j_{q+1}}(v)$ (read/write set definitions). Let $v \in r(t_j)$ such that $s'_{q+1}(v) \neq s_{j_{q+1}}(v)$. This means that there is a variable $v' \in w(t_{j_{q+1}})$ such that $s'_{q+1}(v') \neq s_{j_{q+1}}(v')$ which is a contradiction.

Now that we have proved the property, we know that for $q = k$ we have $s'_k(F) = s_{j_k}(F)$ and that the path σ' exists. We know that for every $i \in \{j_k + 1, \ldots, n\}$ we have $t_i \notin \sigma|_F$ and $w(t_i) \cap F = \emptyset$ since otherwise t_i would be included in the projection (Def. 2). It implies then that $s_n(F) = s_{j_k}(F) = s'_k(F)$.

Proving Theorem 1 is now straightforward. Note that Lemma 2 also shows that a projection preserving transition system TS' always exists.

Proof. Let $\sigma = s_0 \xrightarrow{t_1,t_2,\ldots,t_n} s_n$ be a path in TS and F a set of variables. The projection preservation implies that there exists a path $\sigma' = s_0 \xrightarrow{t'_1,t'_2\ldots t'_m} s'_m$ in TS' such that $\sigma|_F = \sigma'$. From Lemma 2 follows that $s_n(F) = s'_m(F)$.

We have just proved that every reachable combination of values that the variables in F can take, is also reachable through a projection. In other words, Theorem 1 allows us to narrow down the search space of a model checker to projections, while still preserving the soundness of the verification.

5 PBMC: A Symbolic Implementation

In this Section, we show how we implemented projections semantics for process-based concurrent programs in PBMC.

5.1 Process-Based Concurrent Programs.

First, we informally describe how general concurrent programs can be expressed as a transition system. We assume a general shared memory model where a set of *processes* communicate via *shared variables*. In the corresponding transition system, a state consists of variables, and every transition is associated with a process. Processes are *sequential*. Sequentiality means that two transitions that are enabled in a state must be from different processes. Hence, in a state s, a process has at most one enabled transition. Sequential processes can be modeled using an auxiliary variable for every process, called program counter. The program counter variable of a process can only be accessed by the process itself and designates the instruction that can be executed next.

5.2 Projection Encoding

Given a concurrent program, a property, and a fixed depth k, bounded model checking encodes a formula that an SMT/SAT solver can check for satisfiability. The property is true for some path iff the formula is satisfiable. More precisely, the formula is of the form $\Phi = \rho \wedge \Psi$ where ρ denotes the property formula and Ψ encodes a path of length k. The formula Φ is satisfiable iff there exists a path of at most k steps that satisfies ρ. To check whether the property ρ is valid for every possible path of a maximal length k, it suffices to replace it with its negation in Φ and prove the unsatisfiability of the resulting formula.

In the following, we explain how we encode Ψ to implement projections. The basic (unprojected) encoding adapts the structure used in [20]. Let F be the set of variables which appears in the property formula. To model the changes affecting the state of the program throughout the path we create for every $v \in V$ and $0 \leq i \leq k$ a variable v^i to represent the content of v in the i-th state of the path.

Core Formula. A path is only valid if it starts from an initial state. We add a constraint I to encode this fact.

$$I := \bigwedge_{v \in V} \left(v^0 = s_0(v)\right)$$

Let L be the set of all the instructions in the program. For every transition $t \in T$, we refer to the instruction it corresponds to, with $inst(t) \in L$. Given an instruction $l \in L$, let $trans(l)$ be the set of transitions that are mapped to it. In every step $0 \leq i \leq k-1$, we model the possible selection of an instruction l using a formula denoted as T_l^i. If no instruction is selected for a step i, for instance because the length of the returned path is smaller than k, an additional constraint M makes sure that the variables remain unmodified for that step. To guide the solver to only consider projections on F, we add a constraint C_F. Setting C_F to *true* results in the solver considering every possible path. To encode all possible projections on set F, we obtain the following formula:

$$\Psi := I \wedge M \wedge C_F \wedge \bigwedge_{i=0}^{k-1} \bigwedge_{l \in L} T_l^i$$

Transition Encoding. For each instruction $l \in L$, we add an instruction constraint l^i that represents the changes that occur when l is executed at step i. We introduce variables sel^i that encode the instruction choice in every step: $sel^i = l$ iff instruction l was selected for execution in step i. Not that, due to process sequentiality, the selection of an instruction l implies the execution of a corresponding unique transition $t \in trans(l)$ given by the variables v^i. To describe the selection of instructions at different steps we make use of the sel^i variables and the instruction constraints:

$$T_l^i := sel^i = l \implies l^i$$

If an instruction l is selected for execution at step i, then l^i should hold, i.e, the variables should be updated accordingly. Otherwise, if no instruction is selected at step i, every variable in the system remains unchanged:

$$M := \bigwedge_{i=0}^{k-1} \left(\bigwedge_{l \in L} sel^i \neq l \implies \bigwedge_{v \in V} v^{i+1} = v^i \right)$$

If the depth value k is larger than the length of the path satisfying Ψ, some steps are filled with "dummy" instructions[3]. In this case, the solver will spend some time trying to figure out in which position to place the dummy instructions. We found it more efficient to force the solver to place those instructions at the beginning of the path such that there are no gaps, i.e., no dummy instruction is chosen after a "non-dummy" instruction has been selected. We do this by adding a formula that further constrains the assignment of the sel^i variables. We omit the formula as it is an optimization not necessary for the correctness of Ψ.

[3] More precisely, the solver will assign a value to sel^i which does not correspond to any of the instructions.

Projection Encoding. We describe how the projection constraint C_F is generated for a set of variables F. The dependency relation D is encoded using variables d_{ij} which are true iff there is a transition t_1 is executed at step i and a transition t_2 executed at j such that $(t_1, t_2) \in D$. Specifically, we have:

$$d_{ij} := \bigvee_{t_1 \in T} \left(sel^i = id(t_1) \wedge \bigvee_{t_2 \in \{t | (t_1,t) \in D\}} sel^j = id(t_2) \right)$$

C_F directly translates the definition of projections (Definition 2):

$$C_F := \bigwedge_{i=0}^{k-1} \left(\bigvee_{t \in \{t' | w(t') \cap F \neq \emptyset\}} sel^i = id(t) \vee \bigvee_{j=i+1}^{k-1} d_{ij} \vee \bigwedge_{l \in L} sel^i \neq l \right)$$

Informally, for every selected transition t_i either it writes into a variable included in F or there is a transition t_j in the projection after it such that $(t_i, t_j) \in D$. The last clause allows dummy transitions to be selected without rendering the formula unsatisfiable.

Examples. We show how we encode instructions based on the examples of simple assignments and if statements in our process-based concurrent system model. To model process sequentiality, we define program counter variables $pc_p \in V$ for every process p.

Let l be an *assignment* $x := e$ at a process moving the program counter from loc_1 to loc_2, then $l^i :=$

$$pc_p^i = loc_1 \wedge pc_p^{i+1} = loc_2 \wedge x^{i+1} = e^i \wedge \bigwedge_{v \in V \setminus \{x, pc_p\}} v^{i+1} = v^i$$

Considering an *if statement* $if(c)$ that moves the program counter from loc_1 to location loc_2 if c evaluates to true and to loc_3 otherwise, we have $t^i :=$

$$pc_p^i = loc_1 \wedge \big((c \wedge pc_p^{i+1} = loc_2) \vee (\neg c \wedge pc_p^{i+1} = loc_3)\big) \wedge \bigwedge_{v \in V \setminus \{pc_p\}} v^{i+1} = v^i$$

Dependency Encoding. To illustrate how the dependency relation is encoded, we consider the motivating example in Section 2. Because of conflicting read/write accesses, we have three potential dependencies: l_3 depends on l_1, l_4 depends on l_3 and l_3 depends on l_2. The first two dependencies hold for any two transitions associated with the instructions. For instance, every transition corresponding to l_3 depends on every transition associated with l_1. On the other hand, the third dependency holds only if $a = b$. To encode d_{ij}, we must consider every possible dependency. First, there is a dependency if $sel^i = l_3$ and $sel^j = l_4$ or $sel^i = l_1$ and $sel^j = l_3$. For the dependency between l_3 and l_2, we must include the condition $a = b$. Concretely, the following should hold: $sel^i = l_2$, $b^i = a^j$ and $sel^j = l_3$. In summary, to have a dependency between step i and j the the following formula should hold:

$$d_{ij} := (sel^i = l_1 \wedge sel^j = l_3) \vee (sel^i = l_3 \wedge sel^j = l_4) \vee (sel^i = l_2 \wedge (b^i = a^j \wedge sel^j = l_3))$$

The size of a dependency formula depends on the number of potentially dependent instructions and not on the transitions.

Implementation. We implemented PBMC using the above encoding in the Python language. The prototype is based on the Z3 SMT solver and makes use of its Python API [3]. We developed a simplified language that provides basic programming constructs such as assignments, if statements and while loops. The tool supports boolean and integer variables and arrays through the efficient implementation of their respective theories in Z3. Every program contains a header with declarations of variables, the number of processes in the program, an optional initial state assignment and a list of properties to be verified. The body of the program lists the instructions of every process separately in the style of the example shown in Figure 1a.

We now explain the workflow of PBMC. First, the program is parsed and per instruction read/write summaries are created. For instructions accessing an integer or boolean variables the read/write sets are the same for every matching transition. In the case of instructions involving arrays, we also take note of the accessed index. Based on the gathered summaries, the dependencies are inferred. For every two instructions l_1, l_2, we add a dependency for the corresponding transitions if l_1 writes to a variable v that l_2 reads from. If v is an array, we add the condition that the indexes are equal. Next, the tool translates the parsed program into a Z3 formula as previously shown. Subsequently, the found dependencies are used to construct the projection constraints which are added to the formula along with the negated property to be checked and the optional initial state formula. Then, the solver is called to check the satisfiability of the whole formula. In the last step, the output of the solver is interpreted and a counter example path, if existing, is reconstructed. The returned path is a projection that leads to a state where the property is violated. The tool can be started with parameters to set up the length of paths to be considered and whether projection should be applied.

6 Experiments and Evaluation

In this section, we present preliminary experiments and evaluation of PBMC. We challenge our approach by choosing four benchmarks where static program slicing would return a mere copy of the program, and therefore be ineffective, to demonstrate the potential of using projections in program verification.

Next, we present the used benchmarks:

- **Litmus Tests (Litmus):** In our first benchmark, we generate random instructions accessing shared and local variables. The property we check in this example is whether variables assume certain values. For this case, we use 5 configurations ranging from 4 to 8 processes.
- **Indexer:** Our second benchmark is the indexer program taken from [15]. In this program, a shared hash table is accessed concurrently by different processes. Every process attempts to insert data into a location of the hash table. If it is already occupied, the process calculates a new hash value and retries again. This operation is repeated until an empty location is found. In the indexer program, dependencies between variable accesses result from

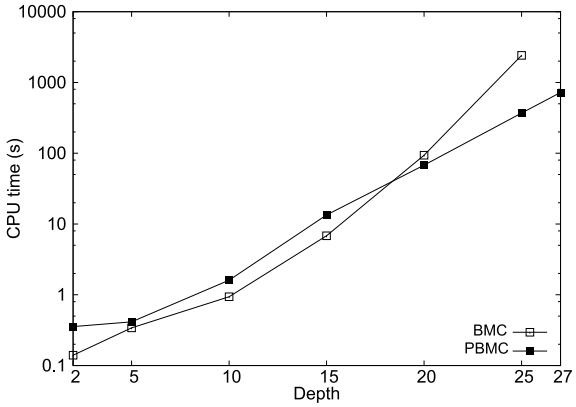

Fig. 2: Comparing the total verification time of BMC and PBMC with different path depths for Litmus 5 and an unsatisfiable property.

writing to and reading from the same hash table location. The property we consider for this example is whether a hash value collisions can occur, which is known to be not the case for the configurations we consider in our setup [18]. We use two configurations with 2 and 4 processes.

– **File System (FSys):** This example was also adapted from [15]. In this benchmark, files are associated with inode data structures which point to memory locations where informations about files are stored. For every memory location there is a busy bit indicating whether it has been allocated to an inode. Each inode and busy bit is guarded by a distinct lock to avoid race conditions. When a process picks an inode and no memory was yet allocated for it, it tries to allocate a free memory location. Here dependencies are hard to detect statically because it is not clear in advance in which order inodes will be assigned by the processes. We check for buffer overflow errors in this benchmark and use one configuration which consist of 5 processes.

– **Dining Philosophers (DPhil):** We implemented the dining philosophers algorithm in our prototype language. The version we use is deadlock and livelock free. While every philosopher (process) maintains a local state, they share an array of chopsticks. To check for the availability of chopsticks, philosophers access the shared array. For simplicity, since collisions can only occur between neighboring processes, we ignore dependencies involving non-neighboring processes. A mutex is used to guarantee atomicity of operations on the shared array elements. The property we are interested in is whether two neighbor philosophers can be eating at the same time. To challenge our tool we inject a bug in the program and evaluate its capacity of finding counter example paths. The injected bug misuses the shared mutex and thus violates the mutual exclusion property. We set the unrolling depth large enough such that the counter example can be found. For this example we use seven different configurations ranging from 5 to 15 processes.

Configuration			CPU Time (s)				time red.
Name	Depth	Prop.	BMC		PBMC		
			Solv.	Total	Solv.	Total	
Indexer 2	10	UNSAT	19.531	20.116	5.088	**5.773**	71.30%
Indexer 4	15	UNSAT	15979.298	15981.531	5881.442	**5884.755**	63.18%
FSys 5	30	UNSAT	37.058	**47.376**	1.825	58.139	—
FSys 5	60	UNSAT	206.297	**227.427**	14.879	227.942	—
FSys 5	70	UNSAT	627.547	651.991	40.469	**325.797**	50%
FSys 5	90	UNSAT	949.518	981.573	10.478	**467.558**	52.37%
FSys 5	100	UNSAT	735.898	771.039	51.768	**617.664**	19.90%
DPhil 5	10	SAT	5.921	**8.830**	7.671	12.665	—
DPhil 7	10	SAT	14.391	**19.162**	14.945	22.585	—
DPhil 10	10	SAT	51.229	59.460	39.607	**52.139**	12.31%
DPhil 12	10	SAT	77.121	88.028	66.544	**82.618**	6.14%
DPhil 15	10	SAT	219.824	235.581	182.689	**204.752**	13.08%
Litmus 4	20	SAT	10.649	**11.348**	13.988	15.819	—
Litmus 5	20	SAT	605.336	**606.475**	654.573	657.102	—
Litmus 6	20	SAT	3888.401	3889.363	908.573	**911.550**	76.56
Litmus 7	20	SAT	2611.024	2612.562	349.708	**353.544**	86.46%
Litmus 8	20	SAT	>2h	>2h	59.738	**64.031**	Infeasible w/o proj.

Table 1: Comparison of BMC and PBMC in different settings.

For every example we use two setups: BMC and PBMC. In general, we observed a trade-off between the complexity of the generated constraints and the amount of reduction achieved during the actual solving. In Figure 2, we show the amount of time spent by PBMC and BMC to verify an unsatisfiable property for the litmus test example with 5 processes with different path depths. The fluctuations of the performance are explained by the fact that SMT solvers make extensive use of heuristics to explore the search space. For small path depth values, the overhead of creating the constraints and handling them by the SMT solver outperforms the reduction that is achieved by using PBMC. Only after reaching a threshold path depth of 20, we observed a clear improvement over BMC. Since the creation of the constraints can be done separately, one can efficiently reduce that threshold by parallelizing the constraint generation process. Moreover, after reaching depth 27 BMC runs out of memory (after two hours) while PBMC finishes the verification within approximately 13 minutes.

In Table 1, we measure the improvement brought by our approach over BMC in terms of CPU time in different setups. In the name column, we append the number of processes to the name of the used benchmark. For every experiment we specify the unrolling depth used and whether the property was satisfiable. For satisfiable properties we write SAT and UNSAT otherwise. The solving time column shows the amount of time spent by the solver to return an answer excluding the initial analysis and formula building steps. On the other hand, total time includes all the steps. In the reduction column, we give the reduction percentage,

in terms of total time, of PBMC in comparison with BMC. From comparing the total and solving times in the table, one can see that as the program complexity increases, the time required for the two initial steps in PBMC's workflow becomes insignificant. This means that for small configurations PBMC brings no improvement in the performance, as the total time is dominated by the time spent on analyzing the program and constructing the formula. On the other hand, PBMC clearly outperforms BMC for larger configurations due to substantial reductions in solving time which becomes more significant. In summary, the results in the table confirms the global trend that was shown in Figure 2. The relatively small reduction in the dining philosopher example can be explained by the extensive use of the globally shared mutex. In that example, all the transitions depend on the ones manipulating the mutex. This results in a large number of dependencies involving all the variables, including those in the property. In general, the larger the setting is with fewer dependencies involving the variables in the property of interest, the better is the reduction of PBMC.

7 Conclusion

We have presented projections, a dynamic slicing notion that can be combined with BMC and proved its correctness. Also, we have implemented PBMC, a bounded model checker that incorporates projections using a novel BMC encoding. By augmenting the BMC formula with projections, PBMC restrains the search space of the model checker and improves on the efficiency over traditional BMC. Our evaluation with examples of concurrent programs has shown major reductions in terms of verification time compared to traditional BMC encoding, even in cases where static slicing proves ineffective.

References

1. Esbmc. http://www.esbmc.org/. Accessed: 2015-04-10.
2. Llbmc. http://llbmc.org/. Accessed: 2015-04-10.
3. Z3. http://z3.codeplex.com/. Accessed: 2015-04-10.
4. P. Abdulla, S. Aronis, B. Jonsson, and K. Sagonas. Optimal dynamic partial order reduction. *ACM SIGPLAN Notices*, 49(1):373–384, 2014.
5. H. Agrawal and J. R. Horgan. Dynamic program slicing. In *ACM SIGPLAN Notices*, volume 25, pages 246–256. ACM, 1990.
6. J. Alglave, D. Kroening, and M. Tautschnig. Partial orders for efficient bounded model checking of concurrent software. In *CAV'13*, pages 141–157. Springer, 2013.
7. D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar. The software model checker blast. *International Journal on Software Tools for Technology Transfer*, 9(5-6):505–525, 2007.
8. A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. *Symbolic model checking without BDDs*. Springer, 1999.
9. P. Bokor, J. Kinder, M. Serafini, and N. Suri. Supporting domain-specific state space reductions through local partial-order reduction. In *ASE'11*, pages 113–122. IEEE Computer Society, 2011.

10. S. Burckhardt, R. Alur, and M. M. Martin. Checkfence: checking consistency of concurrent data types on relaxed memory models. In *ACM SIGPLAN Notices*, volume 42, pages 12–21. ACM, 2007.
11. E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *CAV'00*, pages 154–169. Springer, 2000.
12. E. Clarke, D. Kroening, and F. Lerda. A tool for checking ansi-c programs. In *TACAS'04*, pages 168–176. Springer, 2004.
13. E. M. Clarke, O. Grumberg, and D. Peled. *Model checking*. MIT press, 1999.
14. M. B. Dwyer, J. Hatcliff, M. Hoosier, V. Ranganath, T. Wallentine, et al. Evaluating the effectiveness of slicing for model reduction of concurrent object-oriented programs. In *TACAS'06*, pages 73–89. Springer, 2006.
15. C. Flanagan and P. Godefroid. Dynamic partial-order reduction for model checking software. In *ACM Sigplan Notices*, volume 40, pages 110–121. ACM, 2005.
16. M. Ganai and A. Gupta. Tunneling and slicing: towards scalable bmc. In *DAC'08*, pages 137–142. IEEE, 2008.
17. P. Godefroid, J. van Leeuwen, J. Hartmanis, G. Goos, and P. Wolper. *Partial-order methods for the verification of concurrent systems: an approach to the state-explosion problem*, volume 1032. Springer Heidelberg, 1996.
18. G. Gueta, C. Flanagan, E. Yahav, and M. Sagiv. *Cartesian partial-order reduction*. Springer, 2007.
19. R. Jhala and R. Majumdar. Path slicing. In *ACM SIGPLAN Notices*, volume 40, pages 38–47. ACM, 2005.
20. V. Kahlon, C. Wang, and A. Gupta. Monotonic partial order reduction: An optimal symbolic partial order reduction technique. In *CAV'09*, pages 398–413. Springer, 2009.
21. J. Krinke. Advanced slicing of sequential and concurrent programs. In *Software Maintenance, 2004. Proceedings. 20th IEEE International Conference on*, pages 464–468. IEEE, 2004.
22. A. Lal, S. Qadeer, and S. K. Lahiri. A solver for reachability modulo theories. In *CAV'12*, pages 427–443. Springer, 2012.
23. A. Mazurkiewicz. Trace theory. In *Petri nets: applications and relationships to other models of concurrency*, pages 278–324. Springer, 1987.
24. K. L. McMillan. Lazy abstraction with interpolants. In *CAV'06*, pages 123–136. Springer, 2006.
25. C. S. Păsăreanu, W. Visser, D. Bushnell, J. Geldenhuys, P. Mehlitz, and N. Rungta. Symbolic pathfinder: integrating symbolic execution with model checking for java bytecode analysis. *Automated Software Engineering*, 20(3):391–425, 2013.
26. V. P. Ranganath and J. Hatcliff. Slicing concurrent java programs using indus and kaveri. *International Journal on Software Tools for Technology Transfer*, 9(5-6):489–504, 2007.
27. N. Sinha and C. Wang. On interference abstractions. In *ACM SIGPLAN Notices*, volume 46, pages 423–434. ACM, 2011.
28. A. Valmari. The state explosion problem. In *Lectures on Petri nets I: Basic models*, pages 429–528. Springer, 1998.
29. B. Wachter, D. Kroening, and J. Ouaknine. Verifying multi-threaded software with impact. In *FMCAD'13*, pages 210–217. IEEE, 2013.
30. M. Weiser. Program slicing. In *ICSE'81*, pages 439–449. IEEE Press, 1981.