Block-wise abstract interpretation by combining abstract domains with SMT

Jiahong Jiang[1], Liqian Chen[1], Xueguang Wu[1], and Ji Wang[1,2]

[1]School of Computer Science, National University of Defense Technology, China
[2]State Key Laboratory of High Performance Computing, China
{jhjiang,lqchen,xueguangwu,wj}@nudt.edu.cn

Abstract. Statement-wise abstract interpretation that calculates the abstract semantics of a program statement by statement, is scalable but may cause precision loss due to limited local information attached to each statement. While Satisfiability Modulo Theories (SMT) can be used to characterize precisely the semantics of a loop-free program fragment, it is challenging to analyze loops efficiently using plainly SMT formula. In this paper, we propose a block-wise abstract interpretation framework to analyze a program block by block via combining abstract domains with SMT. We first partition a program into blocks, encode the transfer semantics of a block through SMT formula, and at the exit of a block we abstract the SMT formula that encodes the post-state of a block w.r.t. a given pre-state into an abstract element in a chosen abstract domain. We leverage the widening operator of abstract domains to deal with loops. Then, we design a disjunctive lifting functor on top of abstract domains to represent and transmit useful disjunctive information between blocks. Furthermore, we consider sparsity inside a large block to improve efficiency of the analysis. We develop a prototype based on block-wise abstract interpretation. We have conducted experiments on the benchmarks from SV-COMP 2015. Experimental results show that block-wise analysis can check about 1**x** more properties than statement-wise analysis does.

Keywords: Abstract interpretation, SMT, Abstract domains, Block encoding, Sparsity

1 Introduction

Static analysis based on abstract interpretation (AI) often considers each statement as an individual transfer function, and computes fix-point based on "iteration+widening" strategy [12]. However, the statement-by-statement analysis may cause precision loss due to the limited local information in each statement. It is often the case that the composition of the optimal transformers of individual statements in a sequence does not result in the optimal transformer for the whole sequence [26].

On the other hand, most numerical abstract domains have limitations in expressing disjunctive information, and thus may cause precision loss when dealing with control-flow joins. Satisfiability Modulo Theories (SMT) is expressive for describing constraints, and could represent disjunctions and quantifiers that are common in program semantics. Recently, much attention has been paid on describing semantics of a program through SMT [2][5][24][27][34]. Nevertheless, loops in programs are challenging to cope with in analysis based on purely SMT formulas.

To exploit both advantages of abstract domains and SMT, we propose a framework of block-wise abstract interpretation (BWAI) that extends statement-by-statement analysis to block-by-block analysis by combining SMT and abstract domains. The main idea is following: we first partition a program into several blocks, and then encode a "SMT-expressible" block (e.g., a block without loops) into a SMT formula; we translate the abstract domain representation of the pre-state to a SMT formula at the entry of the block and translate the post-state in SMT formula back to abstract domain representation; in the whole, we compute the fixpoint based on "iteration+widening" strategy block by block and use widening operators at widening points. The strategy of block partitioning is the basis of the BWAI, and two extreme partitioning strategies are to minimize the size of a block and to maximize the size of a block. One extreme strategy to minimize the size of a block considers each statement as a block, in which case the BWAI is degenerated to statement-wise abstract interpretation (SWAI).

Under the BWAI framework, at the exit of a block, we will abstract soundly a SMT formula to the usually less precise abstract element in an abstract domain. Such abstraction may cause precision loss and lead to false positives. Hence, we design a lifting functor on top of base abstract domains to represent and transmit between blocks the useful information that is out of the expressiveness of the base abstract domain but may be helpful for precision of successive analysis. Furthermore, the SMT formula for a block of large size may be so complicated that abstracting it into an abstract element in an abstract domain may be too costly or even run out of memory. To alleviate this problem, we leverage the sparsity inside a large block to improve the efficiency and scalability of BWAI. Finally, we develop a prototype based on BWAI, and have conducted experiments on benchmarks from SV-COMP 2015 [1]. The experimental results show that our BWAI analysis can prove around 66% of the properties in the benchmarks while analysis based on SWAI can prove only around 34% of the properties.

The rest of this paper is organized as follows. Section 2 presents a motivating example of block-wise abstract interpretation (BWAI). In Section 3, we present the BWAI framework. Section 4 presents the lifting functor on top of abstract domains to fit for BWAI. In Section 5, we leverage the block-wise sparsity in a large block to improve the efficiency of analysis. Section 6 describes our implementation together with preliminary experimental results. Section 7 discusses related work. Finally, conclusions as well as future work are given in Section 8.

2 A motivating example

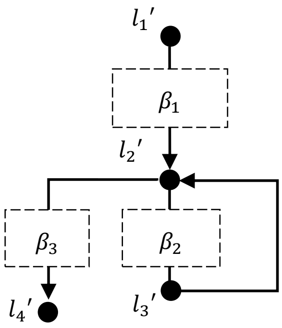
In this section, we give a motivating example shown in Fig. 1(a), which is extracted from *pc_sfifo*.c in the directory "systemc" of SV-COMP 2015. Program *pc_sfifo*.c simulates reading and writing operations on buffers in operating system. Fig. 1(a) shows a fragment of *pc_sfifo*.c. For this example, using SWAI with the octagon abstract domain, we fail to prove the unreachability of the error at line 24. In the following, we illustrate how our BWAI approach works for this example.

```
int q_free, p_dw_st, c_dr_st;
int c_num_read, p_num_write;
...
1 if(brandom()){
2   p_dw_st = 0;
3   c_dr_st = 0;
4   q_free = 1;
5   p_num_write = 0;
6   c_num_read = 0;
7 }else{
8   p_dw_st = 2;
9   c_dr_st = 0;
10  q_free = 0;
11  p_num_write = 1;
12  c_num_read = 0;
13 }
14 while(brandom()){
15  if(p_dw_st == 0){
16    p_dw_st = 1;
17    do_write_p();
18  }
19  if(c_dr_st == 0){
20    c_dr_st = 1;
21    do_read_c();
22  }
23 }
24 if(p_num_write < c_num_read){/*error()*/}
25 ...
```

```
void do_write_p(void){
  if(q_free == 1){
    q_free = 0;
    c_dr_st = 0;
    p_num_write+ = 1;
  }
  p_dw_st = 2;
}

void do_read_c(void){
  if(q_free == 0){
    q_free = 1;
    p_dw_st = 0;
    c_num_read+ = 1;
  }
  c_dr_st = 2;
}
```

(a)



(b)

Fig. 1. A motivating example extracted from SV-COMP 2015

To perform BWAI, we first partition the program and get the block-wise control flow graph (CFG) as shown in Fig. 1(b), where β_1 contains the code fragment from location ℓ_1 to ℓ_{14} in Fig. 1(a) (where ℓ_i represents the program point at the beginning of the i-th line throughout this paper), β_2 contains the code fragment from location ℓ_{14} to ℓ_{23} in Fig. 1(a) (i.e., the loop body), β_3 contains the code fragment outside the loop including locations ℓ_{14}, ℓ_{24} and ℓ_{25} in Fig. 1(a) when *brandom*()==**false**. Note that the code in line 14 (i.e., the head of the **while** loop) turns to an *assume* statement in blocks β_2 and β_3. Hence, program points ℓ'_1, ℓ'_2, ℓ'_3 and ℓ'_4 in Fig. 1(b) are respectively corresponding to locations ℓ_1, ℓ_{14}, ℓ_{23} and ℓ_{25} in Fig. 1(a). And the location ℓ'_2 is the widening point in Fig. 1(b).

When analyzing the block-wise CFG, we characterize the transfer semantics of a block using a SMT formula. E.g., the transfer semantic of block β_1 can be encoded into SMT formula “$\varphi_1^{trans} \triangleq ite(brandom1 == \textbf{true}, (p\_dw\_st = 0) \wedge (c\_dr\_st = 0) \wedge (q\_free = 1) \wedge (p\_num\_read = 0) \wedge (c\_num\_read = 0), (p\_dw\_st = 2) \wedge (c\_dr\_st = 0) \wedge (q\_free = 0) \wedge (p\_num\_read = 1) \wedge (c\_num\_read = 0)$”. Then, we compute the post-state of block β_1 based on the SMT formula given a pre-state, and get an abstract element in an abstract domain at location ℓ_2'. However, converting a SMT formula into a specific abstract domain representation may cause precision loss. E.g., when analyzing block β_1 given a pre-state $\top$, we get the abstract Octagon representation at location ℓ_2' as “$(-1 \leq p\_dw\_st - q\_free \leq 2) \wedge ... \wedge (0 \leq p\_dw\_st \leq 2 \wedge 0 \leq q\_free \leq 1)$”, which causes precision loss (e.g., we in fact know “$((p\_dw\_st = 0) \wedge (c\_dr\_st = 0)) \vee ((p\_dw\_st = 2) \wedge (c\_dr\_st = 0))$” according to the SMT formula φ_1^{trans} that encodes precisely the concrete transfer semantics of β_1). And eventually this precision loss leads to the failure of proving the unreachability of the error at line 24 in Fig. 1(a).

To prove the property, we need more expressive information at location ℓ_2'. In this paper, we choose a predicate set for each block and partition the post state at the exit location of the block according to the value of the predicates. E.g., the predicate sets we choose for block β_1 and β_2 are $\mathbb{P}_1 = \mathbb{P}_2 = \{p_0, p_1, p_2, p_3, p_4\}$, where $p_0 \triangleq (p\_dw\_st == 0)$, $p_1 \triangleq (c\_dr\_st == 0)$, $p_2 \triangleq (q\_free == 0)$, $p_3 \triangleq (q\_free == 1)$ and $p_4 \triangleq (p\_num\_write - c\_num\_read < 0)$. Assume the base abstract domain is Octagon. With the predicate set $\mathbb{P}_1$, we partition the post-state of β_1 at location ℓ_2' and transmit the disjunctive information to analysis of block β_2. Finally, after the fixpoint iteration converges, we could get at location ℓ_2' the invariant “$0 \leq p\_num\_write - c\_num\_read \leq 1$” which proves unreachability of the error at line 24 in Fig. 1(a).

3 Block-wise abstract interpretation framework

3.1 Block partitioning and block encoding

We first present a cutpoint-based approach [19] to partition a program into blocks. The main idea is to select cutpoints from program points, and take the program fragment between two adjacent cutpoints as a block. Let the tuple $\langle \mathcal{L}, \mathcal{E}, \ell_0, \mathcal{L}_e \rangle$ denote the CFG of a program $\mathcal{P}$, where $\mathcal{L}$ is the set of nodes denoting program points, $\mathcal{E}$ is the set of transfer edges, $\ell_0 \in \mathcal{L}$ is the entry node of $\mathcal{P}$, which has no incoming edges, and $\mathcal{L}_e \subseteq \mathcal{L}$ is the set of exit nodes, which have no outcoming edges.

We use $\textbf{SubGraph}(\ell_i, \ell_j) \triangleq \langle \mathcal{L}_{(i,j)}, \mathcal{E}_{(i,j)}, \ell_i, \ell_j \rangle$ to represent the subgraph determined by the node ℓ_i and ℓ_j (with a unique entry node ℓ_i and a unique exit node ℓ_j), where $\mathcal{L}_{(i,j)}$ is the set of nodes in all paths from ℓ_i to ℓ_j and $\mathcal{E}_{(i,j)}$ is the set of the corresponding edges. We call $\textbf{SubGraph}(\ell_i, \ell_j)$ is $\mathcal{L}'$-free if $\mathcal{L}_{(i,j)} \cap \mathcal{L}' = \emptyset$, where $\mathcal{L}' \subseteq \mathcal{L}$ is a subset of program points. Assume $\mathcal{T}$ (e.g., Linear Real Arithmetic, **LRA**) is one theory of SMT. We call an expression is $\mathcal{T}$-Encodable, if it can be encoded by theory $\mathcal{T}$. E.g., expressions that only involve linear computations on program variables of real number type are

LRA-Encodable expressions. We call $\textbf{SubGraph}(\ell_i, \ell_j)$ is $\mathcal{T}$-Encodable if all expressions appearing in $\mathcal{E}_{(i,j)}$ are $\mathcal{T}$-Encodable. Here, we provide the syntactic description of a $\mathcal{T}$-Encodable block β:

$$\tau ::= \textbf{skip} | x := \textbf{exp}, \qquad \beta ::= \tau | \textbf{if}(\textbf{b})\{\beta_1\}\textbf{else}\{\beta_2\} | \beta_1; \beta_2$$

where **exp** is a $\mathcal{T}$-Encodable expression, τ is a **skip** or assignment statement, **b** is a $\mathcal{T}$-Encodable condition expression and β_1, β_2 are $\mathcal{T}$-Encodable blocks. From the syntactic description, we know that $\mathcal{T}$-Encodable block β is loop-free and we assume that β has a unique entry point ℓ_β^{en} and a unique exit point ℓ_β^{ex}.

We say a subset of program points $\mathcal{L}_c \subseteq \mathcal{L}$ is a set of cutpoints w.r.t. the theory $\mathcal{T}$, if $\mathcal{L}_c$ satisfies the following conditions: 1) $\ell_0 \in \mathcal{L}_c$ and $\mathcal{L}_e \subseteq \mathcal{L}_c$; 2) all program points at the head of loops are in $\mathcal{L}_c$; 3) program points before and after statements that are not $\mathcal{T}$-Encodable in SMT are in $\mathcal{L}_c$; 4) $\forall \ell_i \in \mathcal{L}_c \setminus (\{\ell_0\} \cup \mathcal{L}_e)$, $\exists \ell_j \in \mathcal{L}_c$, s.t. ℓ_i and ℓ_j determine a subgraph (i.e. $\textbf{SubGraph}(\ell_i, \ell_j)$) and $\textbf{SubGraph}(\ell_i, \ell_j)$ is $\mathcal{L}_c \setminus (\{\ell_i\} \cup \{\ell_j\})$-free. An extreme set of cutpoints is $\mathcal{L}$ itself, in which case we consider each individual statement as a block. Based on a set of chosen cutpoints, a program can be partitioned into blocks and we get a **CFG with blocks**, denoted as a tuple $\langle \mathcal{L}, \mathcal{L}_c, \mathcal{E}, \mathcal{B}, \ell_0, \mathcal{L}_e \rangle$, where $\mathcal{B}$ is a set of blocks. If a block involves **skip** statements only, we consider this block as an empty block and do not show in the CFG with blocks. Now we present our strategy to choose the set of cutpoints and the corresponding partitioning strategy based on cutpoints.

To make the analysis as precise as possible, we try to partition a program into blocks such that the code size of each block is as large as possible. Hence, we propose a greedy block partitioning strategy (GBP), that is, we only take as cutpoints the loop heads and program points before and after the statement that is not $\mathcal{T}$-Encodable in SMT.

Now we present how to encode the transfer semantics of $\mathcal{T}$-Encodable block β via a SMT formula φ. First, we assume the block β is in a SSA (Static Single Assignment) form [15] such that: 1) in all paths from ℓ_β^{en} to ℓ_β^{ex}, each variable is assigned at most once; 2) the index of each variable in the **then**-branch and **else**-branch is unified at each join point. We use the standard SSA algorithm [15] to translate a program fragment into this format. Let ξ: $\mathcal{B} \to \mathcal{F}$ denote the map from a set of blocks $\mathcal{B}$ to a set of SMT formulas $\mathcal{F}$. We define $\xi(\textbf{skip}) \triangleq \textbf{true}$; $\xi(x := \textbf{exp}) \triangleq (x = \theta(\textbf{exp}))$, where $\theta(\textbf{exp})$ is the SMT encoding for the expression **exp**; $\xi(\textbf{if}(\textbf{b})\{\beta_1\}\textbf{else}\{\beta_2\}) \triangleq ite(\theta(\textbf{b}),\ \xi(\beta_1), \xi(\beta_2))$, where $\theta(\textbf{b})$ is the SMT encoding for condition expression **b**; $\xi(\beta_1; \beta_2) \triangleq \xi(\beta_1) \wedge \xi(\beta_2)$. We use function ξ to encode the whole $\mathcal{T}$-Encodable block as a SMT formula.

3.2 Block-wise iteration strategy combining SMT and abstract domains

In BWAI, the transfer semantics of a block is encoded as a SMT formula, while at cutpoints we maintain abstract domain representation. Hence, before and after cutpoints, we need conversion operators to convert abstract domain representation into a SMT formula and also to convert a SMT formula into abstract domain representation.

Let $[\![\beta_j]\!]^\sharp : \mathcal{S}^\sharp \to \mathcal{S}^\sharp$ characterize the abstract semantics of block β_j, where $\mathcal{S}^\sharp$ denotes the set of abstract states. Assume that the entry point and exit point of β_j are ℓ'_j and ℓ'_{j+1} respectively in the CFG with blocks. If the abstract value at cutpoint ℓ'_j is a_j, then the abstract value at cutpoint ℓ'_{j+1} will be $a_{j+1} \triangleq [\![\beta_j]\!]^\sharp(a_j)$. In the following, we will show how to calculate $[\![\beta_j]\!]^\sharp$.

Let ν: $\mathcal{A} \to \mathcal{F}$ be the map from a set of abstract values in an abstract domain to a set of SMT formulas, and $\nu(a) = \varphi$ where a is an abstract value and φ is a SMT formula. It is often exact to translate abstract value a to the corresponding SMT formula φ, because constraints in most numerical abstract domains could be encoded as SMT formulas directly. E.g., the constraint "$x \in [m_i, m_s]$" in the Box domain could be encoded as a SMT formula "$x \geq m_i \wedge x \leq m_s$".

Let $\zeta : \mathcal{F} \to \mathcal{A}$ be the map from a set of SMT formulas $\mathcal{F}$ to a set of abstract values in an abstract domain $\mathcal{A}$. It is worthy noting that a SMT formula φ is often out of the expressiveness of numerical abstract domains, e.g., when φ involves disjunctions. Computing the function ζ is essentially a problem of symbolic abstraction that aims to calculate the consequence of φ in the abstract domain $\mathcal{A}$. To guarantee the soundness of static analysis, we need a sound conversion operator ζ. Let $\mathbf{Sol}(\varphi)$ denote the solution set of the constraints corresponding to SMT formula φ. We call abstract value a a *sound* abstraction of φ in domain $\mathcal{A}$ if $\mathbf{Sol}(\varphi) \subseteq \mathbf{Sol}(\nu(a))$. We call abstract value a the *best* abstraction in domain $\mathcal{A}$ of SMT formula φ if 1) $\mathbf{Sol}(\varphi) \subseteq \mathbf{Sol}(\nu(a))$ and 2) for all $a' \in \mathcal{A}$, s.t. $\mathbf{Sol}(\varphi) \subseteq \mathbf{Sol}(\nu(a'))$, we have $a \sqsubseteq^\sharp a'$ where $\sqsubseteq^\sharp$ is the inclusion operator in the abstract domain $\mathcal{A}$.

We compute the function ζ via optimization techniques based on SMT (namely, SMT-opt). The SMT-opt problem is to solve "$\mathbf{max}\ e\ s.t.\ \varphi$", where φ is a SMT formula and e is an objective function in SMT format [7][24]. In this paper, we only consider using of abstract domains based on templates which include a large subset of commonly used abstract domains, such as boxes, octagons [25], TCMs [32], etc. The templates determine objective functions and the given SMT formula encoding the post-state determines the constraint space in the SMT-opt problem. We then get the corresponding template abstract domain representation by computing the maximum and minimum value of objective function under the SMT constraint space. It is worthy noting that based on SMT-opt, we get the sound abstraction of a SMT formula in a template abstract domain. Let e_i ($1 \leq i \leq n$, where n is the number of templates) be a template, we get c_i by solving the SMT-opt problem "$\mathbf{max}\ e_i\ s.t.\ \varphi$", and thus get $e_i \leq c_i$ as a constraint in the template abstract domain representation. Overall, $\bigwedge_{i=1}^{n} e_i \leq c_i$ gives the resulting constraint representation in the template abstract domain. Obviously, $\mathbf{Sol}(\varphi) \subseteq \mathbf{Sol}(e_i \leq c_i)$. We have $\mathbf{Sol}(\varphi) \subseteq \mathbf{Sol}(\bigwedge_{i=1}^{n} e_i \leq c_i)$, which guarantees soundness of ζ via SMT-opt. In fact, we get the best abstraction of a SMT formula in a template abstract domain based on SMT-opt. E.g., to get the best abstract representation of Octagon domain for φ, we solve a series of SMT-opt problems like "**max** $(\pm x \pm y)$ *s.t.* φ", which would give the best abstraction of formula φ in the Octagon domain.

Overall, to compute the post abstract value of the transfer semantic of a block represented by a SMT formula φ_j^{trans}, we first transform the abstract value a_j at the entry of the block to SMT format φ_j^{pre} such that $\varphi_j^{pre} = \nu(a_j)$, then abstract the SMT formula "$\varphi_j^{pre} \wedge \varphi_j^{trans}$" to an abstract value a_{j+1}, such that $a_{j+1} = [\![\beta_j]\!]^\sharp(a_j) \triangleq \zeta(\varphi_j^{pre} \wedge \varphi_j^{trans})$.

Now, we briefly describe the iteration strategy on CFG with blocks in the framework of BWAI, to compute the abstract fix-point of a program. In the whole, we still solve the fix-point based on "iteration+widening" strategy. We deal with the statements which are not encoded by SMT based on abstract domains (e.g., the statements which are not $\mathcal{T}$-expressible), the same as in SWAI. When analyzing the block β which is encoded by SMT, we transform the abstract domain representation at the entry point ℓ_β^{en} to a SMT formula, and abstract the post-state in SMT format to an abstract value in an abstract domain at the exit point ℓ_β^{ex} based on the ζ operator. At the widening points $\mathcal{L}_w$, we still use the widening operator of the abstract domain. Overall, our iteration strategy in the framework of BWAI is extended from the one of SWAI, but combines SMT and abstract domains.

4 Abstract domain lifting functor for BWAI

In the framework of BWAI, inside a block we take the advantage of SMT to encode the semantics of the block and use abstract domain representation to transfer information between blocks. Hence, when the block involves behaviors which are beyond the expressiveness of the chosen abstract domain but could be encoded precisely by SMT formula (e.g., disjunctive behaviors), the analysis of BWAI is often more precise than that of SWAI for this block. However, under BWAI, we have to convert the SMT-format representation to specific abstract domain representation at the exit point of a block, which may cause precision loss. E.g., in the motivating example shown in Sect. 2, at the exit point of block β_1, if we use convex abstract domain to abstract the SMT formula that involves disjunction, it would cause precision loss and eventually leads to the failure of proving the unreachability of the error. To express and pass the disjunctive information between blocks, we seek to use abstract domains that could characterize disjunctions. In this section, we present a lifting functor for abstract domains to express disjunctions in order to fit for BWAI for the sake of precision.

The main idea of the lifting functor is to generate a predicate set $\mathbb{P}_i$ for each block β_i through a pre-analysis, and the predicates are all $\mathcal{T}$-Encodable. After analyzing block β_i, we partition the post-state at the exit point $\ell_{\beta_i}^{ex}$ according to different evaluation results of predicates in the set $\mathbb{P}_i$. Note that several approaches are available to utilize predicates to partition the state [8][16][18]. In this paper, we take the advantage of binary decision tree (BDT) [8] to implement the partitioning of a state with respect to predicates. A branch node in the BDT stores a predicate, and each leaf stores abstract value in the base domain under specific evaluation results of predicates. We denote the binary decision tree in parenthesized form

$$[\![p_1 : [\![p_2 : (a_1), (a_2)]\!], [\![p_2 : (a_3), (a_4)]\!]]\!]$$

where p_1, p_2 are predicates in $\mathbb{P}_i$, and $a_j (1 \leq j \leq 4)$ is an abstract value in an abstract domain. It encodes that if p_1 and p_2 are true then a_1 holds, if p_1 is true and p_2 is false then a_2 holds, if p_1 is false and p_2 is true then a_3 holds, or if p_1 and p_2 are false then a_4 holds. Note that the above abstract domain representation precisely encodes the same information as the following SMT formula $(p_1 \wedge p_2 \wedge \nu(a_1)) \vee (p_1 \wedge \neg p_2 \wedge \nu(a_2)) \vee (\neg p_1 \wedge p_2 \wedge \nu(a_3)) \vee (\neg p_1 \wedge \neg p_2 \wedge \nu(a_4))$.

4.1 Predicate selection

In this subsection, we introduce our strategy to determine the predicate set. The criterion of choosing the predicate set for the current block is to improve the precision of successive analysis through transmitting necessary disjunctive information. Hence, we mainly select predicates for current block from branch conditions in successive blocks. Note that in this paper an **assert(b)** statement is transformed into a branch test statement (as in the benchmarks of SV-COMP 2015) and thus the properties to be checked in successive blocks are in fact also added into the predicate set.

We call block β_j is the **direct syntactic successor** of block β_i, if in the CFG with blocks, there exists one path from β_i to β_j without passing through other blocks. Let $\mathbf{DirSynSucc}(\beta_i)$ denote the set of direct syntactic successors of block β_i and $\beta_j \in \mathbf{DirSynSucc}(\beta_i)$. To transmit information of β_i to block β_j at the location $\ell^{ex}_{\beta_i}$, we need to choose a predicate set for β_i to partition the abstract state at the location $\ell^{ex}_{\beta_i}$. Our strategy is to pick the branch conditions in block β_j. Let $\eta : \mathcal{B} \to 2^{\mathcal{C}}$ denote the map from a set of block $\mathcal{B}$ to the powerset of branch conditions appearing in $\mathcal{B}$. We define $\eta(\mathbf{skip}) \triangleq \emptyset$, $\eta(x := \mathbf{exp}) \triangleq \emptyset$, $\eta(\mathbf{if}(b)\{\beta_1\}\mathbf{else}\{\beta_2\}) \triangleq \{b\} \cup \eta(\beta_1) \cup \eta(\beta_2)$, and $\eta(\beta_1; \beta_2) \triangleq \eta(\beta_1) \cup \eta(\beta_2)$. In fact, $\eta(\beta_i)$ collects the branch conditions from the block β_i. E.g., for Fig. 1(b), we have $\eta(\beta_2) = \{p\_dw\_st == 0, c\_dr\_st == 0, q\_free == 0, q\_free == 1\}$ and $\eta(\beta_3) = \{p\_num\_write - c\_num\_read < 0\}$. Since both β_2 and β_3 are syntactic successors of β_1. Hence, for block β_1 we choose the predicate set as $\eta(\beta_2) \cup \eta(\beta_3)$, i.e., $\mathbb{P}_1 = \{p\_dw\_st == 0, c\_dr\_st == 0, q\_free == 0, q\_free == 1, p\_num\_write - c\_num\_read < 0\}$. In general, the predicate set we choose for block β_i is $\mathbb{P}_i \triangleq \bigcup\{\eta(\beta_k) | \beta_k \in \mathbf{DirSynSucc}(\beta_i)\}$.

The complexity of the analysis based on the lifting functor for BWAI is exponential to the height of the BDT which is equal to the number of predicates in $\mathbb{P}$. If $\mathbb{P}$ contains n predicates, the height of BDT is n, and the abstract state at the exit point ℓ^{ex}_{β} is a disjunction with 2^n individuals. In practice, we must balance between precision and efficiency through adjusting the size of predicate set. The predicate set is tightly coupled with each individual block, so the complexity is determined by the size of predicate set for each block locally. In practice, we set a threshold N_{β_i} for each block to restrict the maximum size of predicate set.

4.2 Abstract domain lifting functor based on BDT

Let $\mathcal{FUNC} : \mathcal{PS} \times \mathcal{D} \rightarrow \mathcal{BDT}$, where $\mathcal{PS}$ is a set of predicate sets, $\mathcal{D}$ is a set of base abstract domains, and $\mathcal{BDT}$ is a set of BDT domains on top of base abstract domains. $\mathcal{FUNC}$ defines a lifting functor of base abstract domains in the BWAI framework. Given a predicate set $\mathbb{P}$ and a base abstract domain $\mathcal{A}$, $\mathcal{FUNC}(\mathbb{P}, \mathcal{A})$ gives a domain in BDT format, denoted by $\mathcal{PA}$. The predicates in $\mathbb{P}$ determine the elements on the branch nodes of BDT, $\mathcal{A}$ determines the base abstract domain representation on the leaves of BDT, and each element in the $\mathcal{PA}$ is of BDT format.

The concretization function of $\mathcal{PA}$, denoted by $\gamma_{\mathcal{PA}}$, and abstraction function, denoted by $\alpha_{\mathcal{PA}}$, can be extended easily from $\gamma_{\mathcal{A}}$ and $\alpha_{\mathcal{A}}$ in the base domain $\mathcal{A}$. The concretization function of domain $\mathcal{PA}$ is defined as $\gamma_{\mathcal{PA}}(pa) \triangleq \gamma_{\mathcal{A}}(\bigvee_0^{2^k-1} a_i)$, where pa is the element in the domain $\mathcal{PA}$, k is the size of the predicate set, a_i $(0 \leq i \leq 2^k - 1)$ is the abstract value in the base abstract domain on the i-th leaf. The abstraction function $\alpha_{\mathcal{PA}}$ is computed by calling the abstraction function $\alpha_{\mathcal{A}}$ in the base domain $\mathcal{A}$ multiple times. For example, assume $\bigwedge_{i=1}^{m} \sum_{j=1}^{n} A_{ij} \times x_j \leq c_i$ denotes the constraint system of a template polyhedron, where A_{ij} is the fixed coefficient, m is the number of constraints, n is the dimension of variables. According to different evaluations of predicates in $\mathcal{P}$, we call the SMT-based optimization $\mathbf{max}(\sum_{j=1}^{n} A_{ij} \times x_j)$ *s.t.* $(\psi_p \wedge \varphi)$ for each linear template $(\sum_{j=1}^{n} A_{ij} \times x_j)$, where ψ_p is the conjunction of the constraints corresponding to the predicates on the top-down path in the BDT, φ is the SMT formula characterizing the current state. Then we get a base abstract domain representation on a leaf of the BDT. Other domain operators for $\mathcal{PA}$ derived by the lifting functor can be implemented on top of the domain operators of the base domain $\mathcal{A}$, similarly as in [8].

Example 1. Assume the base domain we use is Octagon. For the example in Fig. 1(a), if we use SMT inside blocks but only use the base domain at cutpoints, we will fail to prove the unreachability of the error in line 24 in Fig. 1. However, if we use the lifting functor over the Octagon domain, we first get $\mathbb{P}_1 = \mathbb{P}_2 = \{p_0, p_1, p_2, p_3, p_4\}$ and $\mathbb{P}_3 = \emptyset$, where $p_0 \triangleq (p\_dw\_st == 0)$, $p_1 \triangleq (c\_dr\_st == 0)$, $p_2 \triangleq (q\_free == 0)$, $p_3 \triangleq (q\_free == 1)$ and $p_4 \triangleq (p\_num\_write - c\_num\_read < 0)$. At exit point of a block, we use the respective predicate set to partition the post-state and get the abstract domain representation in BDT format. E.g., in Fig. 1(b), after the fixpoint iteration converges, at ℓ_2', we get the abstract value in BDT format on top of octagons as $[\![p_0 : [\![p_1 : (oct_1), (oct_2)]\!], [\![p_1 : (oct_3), (oct_4)]\!]]\!]$ (note that the BDT is reduced because different values of $p_2,\ p_3,\ p_4$ do not change the abstract value at leaves), where $oct_1 = (p\_num\_write - c\_num\_read = 0) \wedge ... \wedge (p\_num\_write = 0) \wedge (c\_num\_read = 0)$, $oct_2 = (p\_num\_write - c\_num\_read = 0) \wedge ... \wedge (1 \leq p\_num\_write \leq +\infty) \wedge (1 \leq c\_num\_read \leq +\infty)$, $oct_3 = (p\_num\_write - c\_num\_read = 1) \wedge ... \wedge (p\_num\_write = 1) \wedge (c\_num\_read = 0)$ and $oct_4 = \bot$. Eventually, at location ℓ_2' in Fig. 1(b) we get the invariant "$0 \leqslant p\_num\_write - c\_num\_read \leqslant 1$" which implies the unreachability of the error in line 24 in Fig. 1(a).

5 BWAI considering sparsity in a large block

BWAI based on the greedy block partitioning (GBP) strategy described in Sect. 3.1 could get the most precise analysis inside a block. Nevertheless, when the number of branch conditions in a block is too large, the size of the predicate set for the previous block becomes large, which will result in a large BDT representation and degrade the efficiency of analysis. Especially, sometimes the SMT formula for a block of large size may be too complicated to be solved as a SMT-opt problem. In this paper, we say a block is a *large* block, if the number of assignment statements in this block is greater than the threshold N_{assign} or the number of branch conditions in this block is greater than the threshold N_{branch}, where the two thresholds N_{assign} and N_{branch} are set by users. To improve the scalability of analysis, we divide a large block into several small ones by exploiting sparsity [30] inside this block.

5.1 Dividing a large block based on variable clustering

In this paper, we divide a large block into a series of small blocks based on the concept of variable clustering [22]. We use $\mathbf{Cluster} : \mathcal{B} \rightarrow 2^{\mathcal{VAR}}$ to denote the map from a set of blocks $\mathcal{B}$ to a powerset of variables appearing in $\mathcal{B}$. Given a block β, $\mathbf{Cluster}(\beta)$ satisfies that 1) for any $s \in \mathbf{Cluster}(\beta)$, we have $s \neq \emptyset$; 2) for any $s1,\ s2 \in \mathbf{Cluster}(\beta)$, we have $s1 \cap s2 = \emptyset$ and $\bigcup \mathbf{Cluster}(\beta) = \mathbf{Vars}(\beta)$, where $\mathbf{Vars}(\beta)$ is the set of variables appearing in β. $\mathbf{Cluster}(\beta)$ defines a partitioning of the variable set of block β. In this paper, we use $\mathbf{Cluster}(\beta)$ to denote the variable clustering for block β. To be more clear, we compute variable clusters based on data dependencies among variables. First, we get the data dependency graph among variables in the block, and generate a cluster for each isolated subgraph. Based on variable clustering, we partition a large block into small blocks such that each block involves variables from the same cluster. To do that, we put a sub-cutpoint between two statements if these two statements involve variables from different variable clusters. The statements between two adjacent sub-cutpoints (cutpoints) define a small block.

5.2 Analysis considering block-wise sparsity

Based on variable clustering, we divide a large block into several small ones such that each small block involves only its own variable cluster. Thus, we only need to consider relations among variables appearing within the same small block. Hence, the dimension of considered variables in abstract values is largely reduced. Furthermore, the semantic dependencies between small blocks may become sparse and thus we only need to propagate the abstract value from the current block to those blocks that have data flow dependency on the current block. Hence, we decompose the semantics of a large block into semantics based on small blocks, and consider both spatial and temporal sparsity inside the large block.

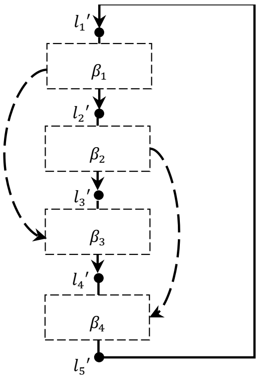
Considering block-wise spatial sparsity. We project out those variables which are not used in the current block from the abstract state at the entry of the block, which could reduce the dimension and also the size of the corresponding SMT formula when analyzing the block.

Considering block-wise temporal sparsity. The block-wise temporal sparsity is based on the following observation: the syntactic successor block of the current block may have no data flow dependency on the current block. We call a block β_j is a **direct semantic successor** of β_i, if 1) β_i and β_j belong to the same "large block", and β_j shares the same variable cluster with β_i; 2) β_j is reachable from β_i; 3) there is no other block β_k between β_i and β_j satisfying 1) and 2). We use $\mathbf{DirSemSucc}(\beta_i)$ to denote the set of direct semantic successors of β_i.

Considering the block-wise temporal sparsity, we propagate abstract values from a block to its direct semantic successor blocks instead of direct syntactic successor blocks, which could avoid unnecessary propagations along the transition edges in CFG with blocks. In addition, we choose the predicate set for a block by extracting branch conditions from its direct semantic successor blocks, instead of its direct syntactic successor blocks. The size of predicate set chosen based on direct semantic successor of a small block is usually much smaller than that chosen by considering the large block as a whole.

```
1 while(brandom()){
2   if(p1 != 0)
3     lk1 = 1;
4   if(p2 != 0)
5     lk2 = 1;
6   if(p1 != 0 && lk1 != 1)
7     //error()
8   if(p2 != 0 && lk2 != 1)
9     //error()
10 }
```

(a)



(b)

Fig. 2. A illustrating example extracted from SV-COMP 2015

Example 2. We take an example extracted from *test_lock*.c in the directory "locks" of SV-COMP 2015. As shown in Fig. 2(a), we consider the loop body as a large block β and get the variable clustering $\{\{lk1,\ p1\}, \{lk2,\ p2\}\}$. Based on the variable clusters, we repartition the block β at locations ℓ_4, ℓ_6, ℓ_8(recall that ℓ_i represents the program point at the beginning of the i-th line) and get the CFG with blocks as shown in Fig. 2(b), i.e., $\beta_1 = \mathbf{SubGraph}(\ell_1, \ell_4)$, $\beta_2 = \mathbf{SubGraph}(\ell_4, \ell_6)$, $\beta_3 = \mathbf{SubGraph}(\ell_6, \ell_8)$ and $\beta_4 = \mathbf{SubGraph}(\ell_8, \ell_{10})$. When we analyze the block β_1, which only involves variables $p1$ and $lk1$, we project out unrelated variables (e.g., $p2$ and $lk2$) based on block-wise spatial sparsity and consider the abstract value over only the variable set $\{p1, lk1\}$.

In Fig.2(b), $\beta_3 \in \textbf{DirSemSucc}(\beta_1)$ and $\beta_4 \in \textbf{DirSemSucc}(\beta_2)$, as shown by the dashed lines. Analysis considering block-wise temporal sparsity propagates the abstract value from β_1 to β_3 instead of β_2. Moreover, the predicate set $\mathbb{P}_1$ for block β_1 is $\{(p1! = 0 \ \&\& \ lk1! = 1)\}$ which is generated according to the branch conditions in its semantic successor block β_3. Note that, without considering block-wise temporal sparsity, we would get a much larger predicate set $\{p1 \ ! = 0, \ p2 \ ! = 0, \ (p1 \ ! = 0 \ \&\& \ lk1 \ ! = 1), \ (p2 \ ! = 0 \ \&\& \ lk2 \ ! = 1)\}$ for the large block (i.e., $\textbf{SubGraph}(\ell_1, \ell_{10})$). For the example shown in Fig. 2(a), analysis considering both block-wise spatial and temporal sparsity can also successfully prove the unreachability of the errors in line 7 and 9. The example in Fig. 2 shows a typical patten of programs in the "locks" directory in SV-COMP 2015. During experiments (in Sect. 6), we find the analysis considering block-wise sparsity is much more efficient than the one without considering block-wise sparsity.

6 Implementation and Experiments

We have implemented the framework of BWAI as a tool prototype named BW-CAI, based on the frontend CIL [29], numerical abstract domain library Apron [23], and SMT optimizer νZ [7]. We have conducted experiments on benchmarks from directories in the C branch of SV-COMP 2015 [1], including directories "loop-lit", "locks", "systemc", "termination-crafted", "termination-crafted-lit" and "termination-restricted-15".

Most benchmarks in the directories "loop-lit", "locks", "systemc" are to simulate the behaviors in operating system. Error locations are inserted manually in each benchmark in these three directories, which are classified as "true-unreachable" and "false-unreachable". "true-unreachable" indicates the error in the example is indeed unreachable, while "false-unreachable" indicates the error in the example is reachable actually. Because abstract interpretation guarantees soundness of the analysis, if we analyze benchmarks with tag "false-unreachable", we find that the errors are all reachable. Hence, we pick the examples with tag "true-unreachable" as the experimental benchmarks and set the unreachability of "false error locations" as the property to check. Examples in the "termination-crafted", "termination-crafted-lit" and "termination-restricted-15" are programs for termination analysis, which are classified to "true-termination" and "false-termination". Similarly, we pick the examples with tag "false-termination" which contain no recurring calls as the experimental benchmarks (since under the framework of abstract interpretation, analysis on "true-termination" programs returns normal termination actually).

Table 1 gives the description information of the benchmarks in the experiments. The column "SV-COMP directories" provides the source location of benchmarks in SV-COMP 2015. We select the examples which are with tag "true-unreachable" as well as "false-termination" and omit examples that contain complicated pointer operations, array accesses and recurring calls. The column "Number of files" gives the number of examples selected in the corresponding directories, totally 98 files. The column "$\overline{\text{LOCs}}$" gives the average number of code lines of programs in the corresponding directories and “$\overline{\#\text{Vars}}$” means the average number of program variables in a program. Among these six directories, the code size of examples in the directory “systemc” is the largest, and the largest single program has 1472 lines of code.

Table 1. Discription of benchmarks from SV-COMP 2015

SV-COMP directories	Number of files	$\overline{\text{LOCs}}$	$\overline{\#\text{Vars}}$
locks	11	163	20
loop-lit	14	21	3
systemc	20	996	65
termination-crafted	16	10	2
termination-crafted-lit	12	11	3
termination-restricted-15	25	21	2

Table 2 summarizes the analysis results on these examples based on the Box and Octagon domain in the framework of SWAI, as well as BWAI with and without using the lifting functor. Column “#Y” presents the number of properties successfully proved, which is the number of error locations proved unreached. Column “t(s)” gives the average analysis time (including the time for block partitioning, block encoding and selecting predicates) in seconds in the corresponding directory when the analysis runs on a double core of 4.0GHz Intel Core i5 on Ubuntu 14.04. In Table 2, we use the GBP strategy by default to partition the program. For the Octagon domain, we use the same variable package [13] manually for SWAI and BWAI, to reduce the cost.

Table 2. Experimental results on benchmarks from SV-COMP 2015

SV-COMP directories (Number of files)	SWAI				BWAI				BWAI + lifting functor			
	Box		Oct		Box		Oct		Box		Oct	
	#Y	t(s)	#Y	t(s)	#Y	t(s)	#Y	t(s)	#Y	t(s)	#Y	t(s)
locks(11)	0	0.28	0	6.40	11	0.81	11	23.30	11	9.13	11	435.14
loop-lit(14)	1	0.09	2	0.12	1	0.17	3	0.43	3	0.95	7	6.77
systemc(20)	0	24.77	0	89.74	0	63.46	0	343.66	1	846.35	5	4733.16
termination -crafted(16)	13	0.08	13	0.09	13	0.12	16	0.14	14	0.35	16	5.22
termination crafted-lit(12)	10	0.08	10	0.09	10	0.13	10	0.19	10	0.44	10	2.13
termination -restricted-15(25)	6	0.09	8	0.09	9	0.14	14	0.20	10	3.05	16	16.75

From the analysis result, we find that if we use the same abstract domain, analysis based on BWAI (even without using lifting functor) is always more precise than that based on SWAI, especially for benchmarks in the directory “locks”. Programs in the directory “locks” are used to simulate the lock and unlock operations in operating system, which often contain one large loop that involves intensive disjunctive behaviors (e.g., disequalities and branches) and the error location is inside the loop body. When using the GBP strategy, analysis based on BWAI could check the property in the loop body based on SMT with no need of converting to abstract domain representation. From the row “locks(11)”, we see that analysis based on Box domain under the framework of BWAI could check all properties in all the 11 benchmarks, while the analysis based on Octagon domain in the framework of SWAI could check none.

Moreover, from the analysis result, we can see that BWAI analysis with lifting functor is more precise than that without lifting functor under the framework of BWAI. From the row “loop-lit(14)”, we see that analysis based on the BDT domain on top of Octagon domain could check 7 examples out of 14, while using only the base Octagon domain could check 3 examples. Most programs in the directory “loop-lit” contain multiple-phases loops which involve disjunctive behaviors, but the property to check is outside the loop. BWAI analysis without lifting functor could not check the property, because the SMT formula needs to be abstracted as base abstract domain representation (such as boxes, octagons, etc.) at the end of loop body. However, when using lifting functor based on BDT, the BWAI could maintain and pass the disjunctive information to the successive analysis. Based on the Box domain, BWAI without lifting functor could prove 44 out of 98 benchmarks, and BWAI with lifting functor could prove 5 more ones; based on the Octagon domain, BWAI without lifting functor could prove 54 out of 98 benchmarks while BWAI with lifting functor could check 11 more ones. Overall, our BWAI with lifting functor could prove around 66% benchmarks (65 out of 98 ones), around one times more than SWAI, which could only check about 34% properties (33 out of 98 ones).

We have also conducted comparison experiments using BWAI with and without considering sparsity. We find that without considering sparsity, the analysis time on each program in “locks” is too long (>5 hours). This is because all programs in “locks” contain at least 11 branches inside a large loop. Hence, when using lifting functor without considering sparsity, the predicate set for the loop body has at least 11 predicates, which results in a very large BDT. Nevertheless, when considering block-wise sparsity, we partition the large loop body into small blocks, and the average size of predicate set for a small block is 2. The analysis time “9.13s” and “435.14s” for the directory “locks” shown in Table 2, is the result when considering block-wise sparsity.

Moreover, we have also conducted the comparison experiments with UFO, which also combines SMT-opt and abstract domains [3][24]. The comparison results between UFO (the version of [24]) and BWCAI (BWAI using BDT on top of Octagon domain) is shown in Table 3. Column “$\#\mathrm{Y}_{\mathrm{com}}$” presents the number of properties which could be proved by both tools. Column “$\#\mathrm{Y}_{\mathrm{UFO}}$” presents the number of properties which could be proved by UFO only, while Column “$\#\mathrm{Y}_{\mathrm{BWCAI}}$” presents the number of properties which could be proved by BWCAI only. Column “$\#\mathrm{N}_{\mathrm{com}}$” presents the number of properties which could be proved by none of them. From Table 3, we can find that UFO could prove more programs than BWCAI in directories “loop-lit” and “systemc”, while BWCAI could prove more programs than UFO in directories “termination-crafted”, “termination-crafted-lit” and “termination-restricted-15”, which shows that the sets of properties proved by the two tools are complementary.

Table 3. Experimental comparison results between BWCAI and UFO

SV-COMP directories (Number of files)	UFO		BWCAI		$\#Y_{\mathrm{com}}$	$\#Y_{\mathrm{UFO}}$	$\#Y_{\mathrm{BWCAI}}$	$\#N_{\mathrm{com}}$
	$\#Y_t$	t(s)	$\#Y_t$	t(s)				
locks(11)	11	0.91	11	435.14	11	0	0	0
loop-lit(14)	10	38.98	7	28.42	7	3	0	4
systemc(20)	18	1278.16	5	4733.16	5	13	0	2
termination-crafted(16)	8	0.22	16	5.22	8	0	8	0
termination-crafted-lit(12)	8	0.16	10	2.13	8	0	2	2
termination-restricted-15(25)	11	2.85	16	16.75	10	1	6	8

Properties in serval examples could not be checked by BWCAI because they need templates whose expressiveness is beyond octagons, while UFO could check them by using interpolants which are less limited to specific templates. E.g., UFO proves the property in "*jm2006_variant_true-unreachable-call*.c" (from the directory "loop-lit"), while BWCAI fails to prove it. However, if adding the template "$x - y + z - i + j$" (that is out of the expressiveness of the Octagon domain), BWCAI could also prove it. On the other hand, UFO *guesses* (typically using interpolants) an inductive invariant by generalizing it from finite paths through the CFG of the program. Hence when the behaviors of the loop in the program are not inductive and the depth of the loop is large, UFO often does not perform well. E.g., "*AlternDiv_false-termination*.c" (from the directory "termination-restricted-15") involves two phases in one loop, and its concrete execution is switching between these two phases back and forth. UFO fails to check the property of it, while BWCAI could prove this property by using BDT on top of the Box domain as well as the Octagon domain. For analysis time, UFO usually costs less time than BWCAI for these programs in Table 3. However, we notice that UFO often costs much more time for the programs that involve multiple-phases loops than other programs. When we manually enlarge the loop bound in those programs, the analysis time based on UFO increases dramatically. E.g., if we manually modify the loop bound from 100 to 10000 in program "*gj2007_true-unreach-call*.c" (from the directory "loop-lit"), the analysis time based on UFO turns into ">1h" from 143s, while the analysis time based on BWAI almost does not change.

7 Related work

The use of block encoding via SMT in software verification has gained much attention recently, especially in software model checking (SMC). Beyer et al. propose large block encoding [5] and adjustable block encoding [6] techniques for SMC based on abstract reachability tree (ART) with CEGAR-based refinement. Their main goal is to improve the efficiency of ART-based software model checking by reducing the number of program paths to explore through large (adjustable) block encoding. Our idea of block encoding is inspired from their work, but our main goal in this paper is to improve the precision of statement-by-statement abstract interpretation through block encoding. Moreover, they use boolean predicate abstraction to represent the abstract successor state in SMC, while we use numerical abstractions to over-approximate the abstract successor state in AI.

Combining decision procedures and abstract interpretation has received increasing attentions recently. Cousot et al. [14] propose to combine abstract interpretation and decision procedures through reduced product or its approximation, e.g., to perform iterated reduction between numerical and SMT-based logic abstract domains. Henry et al. propose a path-sensitive analysis which combines abstract interpretation and SMT-solving, and implement a tool named PAGAI [20]. PAGAI performs fix-point iteration by first focusing temporarily on a certain subset of paths inside the CFG and use "path focusing" [27] technique based on SMT-solving to obtain a new path [21] that needs to enumerate. In this paper we consider a block as a whole, encode all paths in the block as a single SMT formula, and then transform the problem of computing successor abstract value w.r.t a SMT formula into SMT-opt problems.

The recent work by Li et al. [24] on using SMT-based symbolic optimization to implement the best abstract transformer, is the closest related work to our work. They propose an efficient SMT-based optimization algorithm namely SYMBA, and use SYMBA to calculate the best abstract transformer for numerical abstract domains in UFO [2][24]. In this paper, we also use SMT-based optimization technique to compute the abstract value given a SMT formula. In our implementation, we use νZ [7] which is a SMT-based optimizer, but we could also use SYMBA. However, the abstract value we use SMT-based optimization to compute is in a lifting domain of base numerical domains extended with BDT. Besides, we use only abstract interpretation while UFO combines abstraction based over-approximation and interpolation based under-approximation. The experimental comparison results of our approach and UFO [3] using SYMBA in Sect.6 show that the sets of properties proved by the two approaches are complementary.

The problem of computing the best symbolic abstract transformers is first considered by Reps et al. in [31] and has gained much attention recently. Reps et al. have done a series of work on constructing abstract transformers relying on decidable logics [33][34]. The main idea is to use a least and a greatest fix-point computation to maintain an over-approximation and an under-approximation of the desired result. In general, their approach fits for arbitrary numerical abstract domains, but the iteration process may not terminate and thus needs a threshold to stop the iteration, which gives over- and under- approximations for the best abstract transformer. In this paper, we use SMT-based optimizer to compute the best abstract transformer but only for template based abstract domains. Monniaux et al. propose a method for computing optimal abstract transformers over template linear constraint domains but their approach is based on quantifier elimination [26][28], while we use SMT-based optimization.

Recently, to deal with disjunctive properties, a variety of abstract domains have been designed to allow describing disjunctive information inside the domain representation. Examples include abstract domains supporting *max* operation [4], abstract value function [9][11], interval linear constraints [10], set minus [17], decision diagrams [18], etc. More recently, Chen et al. propose a binary decision tree (BDT) abstract domain functor which provides a new prospective on partitioning the trace semantics of programs as well as separating properties in leaves [8]. In this paper, we use an abstract domain lifting functor based on BDT, but we further propose a specific selection strategy for predicate set as the branch nodes in BDT to fit for BWAI. We choose for the current block the predicate set that is determined locally by its direct syntactic/semantical successor blocks, while [8] determines the branch nodes in BDT based on a branch condition path abstraction that abstracts the history of the control flow to the current program point.

8 Conclusion and Future work

We extend statement-by-statement abstract interpretation to block-by-block abstract interpretation, and propose block-wise abstract interpretation (BWAI) by combining abstract domains with SMT. In the framework of BWAI, we use a SMT formula to encode precisely the transfer semantics of a block and then analyze the block as a whole, which usually gives more precise results than using abstract domains to analyze the block statement by statement. Moreover, in order to transmit useful disjunctive information between blocks which is obtained by SMT-based analysis inside a block, we propose a lifting functor on top of abstract domains to fit for BWAI. The lifting functor is implemented based on binary decision trees, wherein the branch nodes are determined by a selection of predicates according to the direct successor relationship between blocks. Furthermore, to improve the efficiency of BWAI, we consider block-wise sparsity in a large block by dividing a large block further into a set of small blocks. Experimental results on a set of benchmarks from SV-COMP 2015 show that our BWAI approach could prove around one times more benchmarks than SWAI (our BWAI approach could prove 66% ones, while SWAI approach could only prove 34% ones).

For the future work, we will consider more flexible block partitioning strategies to balance between precision and efficiency. Also, we plan to develop more powerful SMT-based optimization solvers to support more SMT theories (such as the array theory).

Acknowledgments. We thank Arie Gurfinkel for the help on using UFO. This work is supported by the 973 Program under Grant No. 2014CB340703, the NSFC under Grant Nos. 61120106006, 91318301, 61532007, 61690203, and the Open Project of Shanghai Key Laboratory of Trustworthy Computing under Grant No. 07dz22304201504.

References

1. `http://sv-comp.sosy-lab.org/2015/`.
2. Aws Albarghouthi, Arie Gurfinkel, and Marsha Chechik. From under-approximations to over-approximations and back. In *TACAS'12*, pages 157–172. Springer, 2012.
3. Aws Albarghouthi, Yi Li, Arie Gurfinkel, and Marsha Chechik. UFO: a framework for abstraction- and interpolation-based software verification. In *CAV'12*, pages 672–678. Springer, 2012.
4. Xavier Allamigeon, Stéphane Gaubert, and Eric Goubault. Inferring min and max invariants using max-plus polyhedra. In *SAS'08*, pages 189–204. Springer, 2008.
5. Dirk Beyer, Alessandro Cimatti, Alberto Griggio, M Erkan Keremoglu, and Roberto Sebastiani. Software model checking via large-block encoding. In *FMCAD'09*, pages 25–32. IEEE, 2009.
6. Dirk Beyer, M Erkan Keremoglu, and Philipp Wendler. Predicate abstraction with adjustable-block encoding. In *FMCAD'10*, pages 189–198. IEEE, 2010.
7. Nikolaj Bjørner, Anh-Dung Phan, and Lars Fleckenstein. νZ-an optimizing smt solver. In *TACAS'15*, pages 194–199. Springer, 2015.
8. Junjie Chen and Patrick Cousot. A binary decision tree abstract domain functor. In *SAS'15*, pages 36–53. Springer, 2015.
9. Liqian Chen, Jiangchao Liu, Antoine Miné, Deepak Kapur, and Ji Wang. An abstract domain to infer octagonal constraints with absolute value. In *SAS'14*, pages 101–117. Springer, 2014.
10. Liqian Chen, Antoine Miné, Ji Wang, and Patrick Cousot. Interval polyhedra: An abstract domain to infer interval linear relationships. In *SAS'09*, pages 309–325. Springer, 2009.
11. Liqian Chen, Antoine Miné, Ji Wang, and Patrick Cousot. Linear absolute value relation analysis. In *ESOP'11*, pages 156–175. Springer, 2011.
12. Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL'77*, pages 238–252. ACM, 1977.
13. Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, and Xavier Rival. Why does astrée scale up? *Formal Methods in System Design*, 35(3):229–264, 2009.
14. Patrick Cousot, Radhia Cousot, and Laurent Mauborgne. The reduced product of abstract domains and the combination of decision procedures. In *FSSCS'11*, pages 456–472. Springer, 2011.
15. Ron Cytron, Jeanne Ferrante, Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, 1991.
16. Jeffrey Fischer, Ranjit Jhala, and Rupak Majumdar. Joining dataflow with predicates. In *ESEC-FSE'05*, pages 227–236. ACM, 2005.
17. Khalil Ghorbal, Franjo Ivančić, Gogul Balakrishnan, Naoto Maeda, and Aarti Gupta. Donut domains: Efficient non-convex domains for abstract interpretation. In *VMCAI'12*, pages 235–250. Springer, 2012.
18. Arie Gurfinkel and Sagar Chaki. Boxes: A symbolic abstract domain of boxes. In *SAS'10*, pages 287–303. Springer, 2010.
19. Arie Gurfinkel, Sagar Chaki, and Samir Sapra. Efficient predicate abstraction of program summaries. In *NASA Formal Methods Symposium*, pages 131–145. Springer, 2011.

20. Julien Henry, David Monniaux, and Matthieu Moy. PAGAI: A path sensitive static analyser. *Electronic Notes in Theoretical Computer Science*, 289:15–25, 2012.
21. Julien Henry, David Monniaux, and Matthieu Moy. Succinct representations for abstract interpretation. In *SAS'12*, pages 283–299. Springer, 2012.
22. Kihong Heo, Hakjoo Oh, and Hongseok Yang. Learning a variable-clustering strategy for octagon from labeled data generated by a static analysis. In *SAS'16*. Springer, 2016.
23. Bertrand Jeannet and Antoine Miné. Apron: A library of numerical abstract domains for static analysis. In *CAV'09*, pages 661–667. Springer, 2009.
24. Yi Li, Aws Albarghouthi, Zachary Kincaid, Arie Gurfinkel, and Marsha Chechik. Symbolic optimization with SMT solvers. In *POPL'14*, volume 49, pages 607–618. ACM, 2014.
25. Antoine Miné. The octagon abstract domain. *Higher-order and symbolic computation*, 19(1):31–100, 2006.
26. David Monniaux. Automatic modular abstractions for template numerical constraints. *Logical Methods in Computer Science*, 6(3):501–516, 2010.
27. David Monniaux and Laure Gonnord. Using bounded model checking to focus fixpoint iterations. In *SAS'11*, pages 369–385. Springer, 2011.
28. David P Monniaux. Automatic modular abstractions for linear constraints. In *POPL'09*, volume 44, pages 140–151. ACM, 2009.
29. George C Necula, Scott McPeak, Shree P Rahul, and Westley Weimer. CIL: Intermediate language and tools for analysis and transformation of c programs. In *CC'02*, pages 213–228. Springer, 2002.
30. Hakjoo Oh, Kihong Heo, Wonchan Lee, Woosuk Lee, Daejun Park, Jeehoon Kang, and Kwangkeun Yi. Global sparse analysis framework. *ACM Transactions on Programming Languages and Systems*, 36(3):1–44, 2014.
31. Thomas Reps, Mooly Sagiv, and Greta Yorsh. Symbolic implementation of the best transformer. In *VMCAI'04*, pages 252–266. Springer, 2004.
32. Sriram Sankaranarayanan, Henny B Sipma, and Zohar Manna. Scalable analysis of linear systems using mathematical programming. In *VMCAI'05*, pages 25–41. Springer, 2005.
33. Aditya Thakur, Matt Elder, and Thomas Reps. Bilateral algorithms for symbolic abstraction. In *SAS'12*, pages 111–128. Springer, 2012.
34. Aditya Thakur and Thomas Reps. A method for symbolic computation of abstract operations. In *CAV'12*, pages 174–192, 2012.