Modular Static Analysis using SMT-solving and Abstract Interpretation

Julien Henry

April 5, 2013

1 Block decomposition

We start from the control flow graph of a program. Instead of doing the analysis directly over this graph, we are going to summarize some parts of the graph in order to analyse these parts modularly. We thus obtain a new graph, where the nodes are called *blocks*.

A block may represent:

- a loop or a function summary.
- an abstraction of non-linear and/or floating point operation.
- an abstraction of several paths. This allows to compute an I/O relation between two control points of the CFG and thus prevent the SMT solver to enumerate too many paths (cf Diego).
- a basic-block. In this case, the block is not an abstraction.

Property 1. *The choice of blocks has to verify the following properties:*

- *Each block has one single header, which is the only block with an incoming transition coming from outside.*
- *If we split the header block into 2 blocks (the first one with only outgoing transitions, the second one with only incoming transitions), the graph of sub-blocks represented by the block is acyclic.*

Figure 1 illustrates a possible decomposition into blocks for a usual control flow graph.

Each block has a set of input variables, noted $B(X)$, as well as a set of output variables, noted $B(X')$. We note $B(X, X')$ the set $B(X) \cup B(X')$.

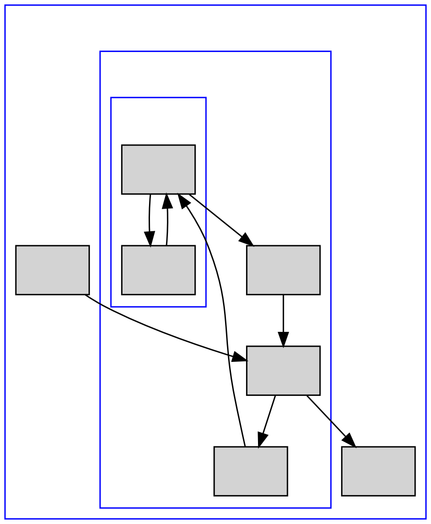


Figure 1: Correct decomposition of a control flow graph into Blocks.

2 Analysis

We start with a program $\mathcal{P}$ (with input variables X and output variables X'), for which we want to prove a property $P(X, X')$. We can assume some properties about the input variables X of the program: we note $C(X)$ these assumptions.

Our algorithm abstracts the program $\mathcal{P}$ as a block B, and tries to prove (as lazily as possible) that there is no trace in B that negates $P(X, X')$. For doing this, it tries first to prove P with rough abstractions of the block subcomponents, and refines these abstractions modularly until they are sufficiently precise to prove P.

2.1 IORelation

We note R_B the i/o relation for block B, and initialize it to $\top$. We iteratively refine this abstraction until we have $R_B \sqcap P(X, X') = \bot$.

To each block representing an abstraction of a subgraph, we associate a mapping function $invariant : \mathcal{A} \longrightarrow \mathcal{A}$, where $\mathcal{A}$ is the abstract domain in use, that maps properties about the input variables $C(X)$ to a overapproximated i/o relation between input and output variables, safe if $C(X)$. This is a partial function that is constructed dynamically during the analysis.

Procedure IORelation aims at updating this partial function with $invariant : A(X) \mapsto R_B$, where $C(X) \sqsubseteq A(X)$, and R_B a "more precise" i/o relation.

For doing this, we construct an SMT formula ρ_B representing the semantics of the block B:

Algorithm 1 Modular Static analysis using SMT

```
1: function IORelation(Block $B$, Property $P(X, X')$, Context $C(X)$)
2:     $R_B \leftarrow \top$
3:     $A(X) \leftarrow \top$
4:     while SMTsolve($C(X) \wedge \rho_B \wedge \neg P(X, X')$) do
5:         $M \leftarrow$ getModel()
6:         $(B_1, \dots, B_n) \leftarrow$ AbstractedBlocks($M$)
7:         Refine($B_1, \dots, B_n, M$)
8:         Recompute $\rho_B$
9:         $A(X), R_B \leftarrow$ Update($B, C(X)$)
10:        if Update has not improved $A(X)$ or $R_B$ then
11:            return ($unknown$)
12:        end if
13:    end while
14:    Define $invariant_B : A(X) \mapsto R_B$
15:    return ($true$)
16: end function
```

- The formulae representing abstraction blocks are constructed using their *invariant* partial function.

$$io_B = \bigwedge_{C(X) \in Dom(invariant_B)} (C(X) \implies invariant_B(C(X)))$$

- Other blocks, transitions, etc. are encoded as in Path Focusing papers [4, 2].

- For blocks representing several paths, we use their *invariant* partial function to plug a summary of these paths, as well as their classical encoding (see Path Focusing). This last encoding is needed to be fully precise, and the summary will serve the SMT-solver to avoid exhaustive exploration of "diamond" formulae.

Then, we keep asking SMT queries to find paths inside the block that negate the property we want to prove. Each time we discover a path leading to $\neg P(X, X')$, we try to cut the trace by refining one (or more) of the abstraction blocks the focus path goes through. We can therefore update the formula ρ_B, since the partial functions *invariant* of some abstraction blocks have changed. We can also recompute an i/o relation for this block, that should be more precise since the subcomponents have been refined.

At some point, it may be the case that we can not refine abstraction blocks anymore, and the error trace is still feasible. In this case, the algorithm answers "unknown", in the sense that the property $P(X, X')$ is not proved correct.

In this way, we can strengthen the invariant incrementally when the previous one is not sufficient, and avoid being unnecessarily precise on the part of the code that is not relevant.

2.2 Update

```
1: function UPDATE(Block $B$, Context $C(X)$)
2:     $A(X), R_B^{over} \leftarrow$ run one of S, G, PF, DIS techniques
3:                                                   ▷ use COMPUTETRANSFORM
4:     Possibly refine $R_B^{under}$
5:         ▷ $A(X)$ are the assumptions that have been used ($C(X) \sqsubseteq A(X)$)
6:     return $A(X), R_B$
7: end function
```

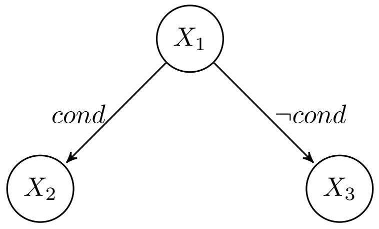
UPDATE$(B, C(X))$ compute an input/output relation for block B that is correct at least in the context $C(X)$. For this, we apply one of the state-of-the-art techniques such as *Guided Static Analysis* [1], *Path-Focusing* [4] or one of its extension (combined technique, disjunctive version)[2].

We could also optionally keep an underapproximation of B to avoid refining useless blocks. This underapproximation can start at $\bot$ and be updated incrementally.

2.2.1 From Context to $A(X)$

Even though we suppose $C(X)$ for the input variables, we start the analysis with $head(X) = \top$, where $head(X)$ is the abstract value attached to the block header. Intuitively, the idea is to compute an i/o relation "as precise as" the one we would obtain when starting with $head(X) = C(X)$, but in a sense more general, i.e true for a larger set of input variables, noted $A(X)$ (then, $C(X) \sqsubseteq A(X)$). We start the analysis with $A(X) = \top$, collect the properties implied by $C(X)$ that we use to improve the precision of our invariant, and update $A(X)$ iteratively.

Suppose we have a state with conditionnal branch:



$$\begin{tikzcd} & X_1 \arrow[dl, "cond"'] \arrow[dr, "\neg cond"] & \\ X_2 & & X_3 \end{tikzcd}$$

If $X_1(X, X') \sqcap cond(X') = \bot$ or $X_1(X, X') \sqcap \neg cond(X') = \bot$, we remain precise without the need of any additionnal assumptions. Otherwise, some properties implied by $C(X)$ could make some transitions become unfeasible. This happens if:

1. $X_1(X, X') \sqcap C(X) \sqcap cond(X') = \bot$:

   in this case, we add to $A(X)$ the projection over X of $\neg cond(X') \sqcap X_1(X, X')$. In other words, we assign $A(X) \leftarrow A(X) \sqcap P_X(\neg cond(X') \sqcap X_1(X, X'))$ and intersect each X_i with the new $A(X)$.

2. $X_1(X, X') \sqcap C(X) \sqcap \neg cond(X') = \bot$:
   similarly, $A(X) \leftarrow A(X) \sqcap P_X(cond(X') \sqcap X_1(X, X'))$.

More generally, suppose we have a path τ that is feasible with the current $A(X)$ (the image $\tau(X_1(X, X'))$ is not $\bot$), but $\tau(X_1(X, X') \sqcap C(X)) = \bot$.

Let $C_1(X'), C_2(X'), \dots, C_n(X')$ be the guards the path τ goes through. We apply quantifier elimination over the formula

$$\exists\ x'_1, x'_2, \dots, x'_k,\ X_1(X, X') \wedge \bigwedge_{1 \le i \le n} C_i(X')$$

where $X' = \{x'_1, x'_2, \dots, x'_k\}$. We obtain an equivalent formula $F(X)$ that we negate and add to $A(X)$:

$$A(X) \leftarrow A(X) \wedge \neg F(X)$$

The analysis continues, considering path τ to be unfeasible. Finally, for each path we were able to "cut", an associate formula has been added to $A(X)$. $A(X)$ may not be a convex polyhedron, we thus can search for a convex polyhedron implied by $C(X)$ and that implies $A(X)$, using SMT-solving:

```
while true do
    $m(X) \leftarrow$ SMT-query $A(X)$
    Generalize $m(X)$ into a convex polyhedron $M(X)$    ▷ see [3] for details
    if $C(X) \wedge \neg M(X)$ is unsat then
        return $M(X)$,
    else
        $A(X) \leftarrow A(X) \wedge \neg M(X)$
    end if
end while
```

Example 1. Suppose we have $X = \{x_0, y_0\}, X' = \{x, y\}$, $C(X) \equiv x_0 > 20$, $X_1(X, X') = x \ge x_0 + 2 \wedge y \ge y_0$, and a path with guards $cond(X') \equiv y \le 0 \wedge x \le 10$. Suppose also that the context $C(X)$ verifies $C(X) \wedge X_1(X, X') \wedge cond(X') \equiv \bot$.

We apply quantifier elimination over the formula

$$\exists\ x\ y,\ (y \le 0) \wedge (x \le 10) \wedge (x \ge x_0 + 2) \wedge (y \ge y_0)$$

We obtain:

$$(x_0 \le 8) \wedge (y_0 \le 0)$$

We then add the negation of this formula to our current $A(X)$:

$$A(X) \leftarrow A(X) \wedge ((x_0 > 8) \vee (y_0 > 0))$$

If, at the end, $A(X) \equiv ((x_0 > 8) \vee (y_0 > 0))$, we can search for a convex polyhedron implied by $C(X)$ and that implies $A(X)$:

- SMT-query of $assert(A(X))$ gives model $(x_0, y_0) = (10, 0)$.
- We generalize the model and obtain $x_0 > 8$.
- SMT-query $C(X) \wedge \neg(x_0 > 8)$ is *unsat*. We thus can choose $x_0 > 8$ as a correct context.

Note that the first SMT query could have returned another model whose generalisation is $y_0 > 0$, which is not implied by $C(X)$. In this case, several SMT-queries are required.

At some point, we may also cross over an abstracted block B'. In this case, we use the "best" abstraction of B' that is already computed. We can choose the smallest element $A' \in Dom(invariant_{B'})$ such that $X_1 \sqcap C(X) \implies A'$ and use $invariant_{B'}(A')$ as the i/o relation of B'. Then, we have to assign $A(X) \leftarrow A(X) \sqcap A'$ and update the X_i's as before.

Note that instead of using one single i/o relation of B', we can use all of the relations that are correct in our context to improve precision (see Algorithm ComputeTransform).

```
function ComputeTransform(Abstract $I(X)$, Block $B$, Context $C(X)$)
    if $B$ is an abstraction then
        $A(X) \leftarrow \top$
        $Temp(X, X') \leftarrow ChangeDimension(I : (X) \rightarrow (X, X'))$
        for all $P(X) \in Dom(invariant_B)$ s.t $C(X) \sqsubseteq P(X)$ do
            $Temp(X, X') = Temp(X, X') \sqcap invariant_B(P(X))$
            $A(X) \leftarrow A(X) \sqcap P(X)$
        end for
        $I'(X') \leftarrow Projection(Temp(X, X'), X')$
        return $A(X), I'(X')$
    else
        Do as usual
    end if
end function
```

2.3 Refine

We have a model M that maps variables to values. We also have a list of abstraction blocks that the trace extracted from the model goes through. Refine recomputes the i/o relation of some of these blocks in order to "cut" the trace (so that the trace becomes unfeasible). For computing these i/o relations, we create a calling context for each block corresponding to the model M (the calling context is then a single point), and an objective, $M(X_{input}, X_{output})$ being the error state we want to avoid.

If the model is included in the underapproximation of a block B_i, there is no hope to cut the trace by refining B_i. More generally, we could temporarily

```
function REFINE(Blocks $B_1, \dots, B_n$, Model $M$)
    $res \leftarrow false$
    $i \leftarrow 1$
    while $res \neq true \wedge i <= n$ do
        if $M(X_i, X_{i+1}) \nsubseteq R_{B_i}^{under}$ then
            $(res) \leftarrow$ IORelation($B_i$, $\neg M(X_i, X_{i+1})$, $M(X_i)$)
        end if
        $i \leftarrow i + 1$
    end while
end function
```

replace in ρ_B the invariant $R_{B_i}^{over}$ by $R_{B_i}^{under}$, and see whether $\rho_B \wedge P(X, X')$ is still *sat*. If so, we do not refine it.

Another criterion for refining or not a subblock could be based on program slicing: there is no need to refine a block that does not modify the relevant variables.

2.4 Remarks

- we can stop the analysis at any moment, since the R_B invariants are always safe.
- the R_B^{under} must not necessarily be an underapproximation of the possible states, since it is only used to choose which block to refine. For instance, we can set R_B^{under} to R_B^{over} when we know that R_B^{over} cannot be refined: for instance, if B abstracts a non-linear operation and R_B^{over} has already been computed using every implemented technique (Miné, Bernstein, etc.)

3 Generation of preconditions

Using this framework, we can generate "interesting" preconditions at the header of blocks (representing loops, functions, etc.). Indeed, suppose we want to find a sufficient precondition for a block B such that the property $P(X, X')$ is true, and that implies $C(X)$. If the algorithm IORELATION proves $P(X, X')$ in the context $C(X)$, we can use the resulting $A(X)$ as a sufficient precondition. However, this technique requires the knowledge of a context $C(X)$ where P is correct. If we do not know such context, we can:

1. Use SMT-solving to find a path in B verifying P.
2. Use $Model(X)$ as the context for running the algorithm IORELATION
3. The algorithm may return *true* and give a precondition $A(X)$ that implies $M(X)$.

We can apply the three last points iteratively to discover “classes” of inputs for which we have P. To do so, we just have to add in the SMT query the negation of the already-discovered preconditions to prevent the solver to give a model that is in the same “class” as a previous one. Note that we should stop the iterations at some point, since it may be possible to keep going indefinitely (there may be an infinite number of such classes ?).

The interest here is that we can discover non-convex preconditions, i.e. a disjunction of conjunctions.

References

[1] Denis Gopan and Thomas W. Reps. Guided static analysis. In *SAS*, volume 4634 of *LNCS*, pages 349–365. Springer, 2007.

[2] Julien Henry, David Monniaux, and Matthieu Moy. Succinct representations for abstract interpretation. In *Static analysis (SAS)*, volume 7460 of *Lecture Notes in Computer Science*, pages 283–299. Springer Verlag, 2012.

[3] David Monniaux. Quantifier elimination by lazy model enumeration. In *Computer-aided verification (CAV)*, number 6174 in Lecture Notes in Computer Science, pages 585–599. Springer Verlag, 2010.

[4] David Monniaux and Laure Gonnord. Using bounded model checking to focus fixpoint iterations. In Eran Yahav, editor, *Static analysis (SAS)*, volume 6887 of *Lecture Notes in Computer Science*, pages 369–385. Springer Verlag, 2011.