A tool which mines partial execution traces to improve static analysis

Gianluca Amato, Maurizio Parton, and Francesca Scozzari

Università "G. d'Annunzio" di Chieti e Pescara – Dipartimento di Scienze

Abstract. We present a tool which performs abstract interpretation based static analysis of numerical variables. The novelty is that the analysis is parametric, and parameters are chosen by applying a variant of principal component analysis to partial execution traces of programs.

Abstract interpretation based static analysis [5] may be used to prove runtime properties of program variables such as "all the array indexes are contained within the correct bounds". It discovers assertions which hold when execution reaches specific program points. The expressive power of assertions depends on the particular choice of the *abstract domain*. The simplest abstract domain for numerical properties is the *interval domain* [4], which allows assertions of the form $m \leq x \leq M$ where x is a program variable and m, M are constants.

A lot of research is devoted to explore the trade-off between precision, expressive power and computational cost of abstract domains. In this context, we have recently proposed a family of parametric *parallelotope domains* [1]. They are similar to the interval domain, except that intervals are expressed in a non-standard basis in the vector space of variable's values. The non-standard basis is the parameter of the domain: given a change of basis matrix A, our domain includes all the assertions of the form $\boldsymbol{m} \leq A\boldsymbol{x} \leq \boldsymbol{M}$, where $\boldsymbol{x}$ is the vector of program variables and A is fixed for the entire analysis. When the basis is cleverly chosen, parallelotopes approximate the invariants with a greater precision than intervals, as illustrated in Figures 1, 2 and 3 on a partial execution trace.

In order to find the "optimal" basis, we propose a new technique based on a pre-analysis of the partial execution traces of the program. First, we collect the values of numerical variables in all the program points for different inputs. Then, we apply to the sample data a statistical technique called *orthogonal simple component analysis* (OSCA) [2], which is a variant of *principal component analysis* (PCA). It finds a new orthonormal coordinate system maximizing the variance of the collected values. More explicitly, PCA finds new axes such that the variance of the projection of the data points on the first axis is the maximum among all possible directions, the variance of the projection of the data points on the second axis is the maximum among all possible directions which are orthogonal to the first axis, and so on. If we apply PCA to the values collected from partial executions traces of the program in Figure 1, we get the new basis (x', y') in Figure 3. OSCA returns an approximation of PCA such that the principal components are proportional to vectors of small integers, a property which helps the

```
xyline = function(x)
{
   assume(x>=0)
   y=-x
   while(x>y) {
     y= y+1
   }
}
```

Fig. 1. The example program `xyline`.

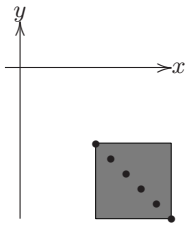

Fig. 2. Interval abstraction of a partial execution trace, observed at program point ①.

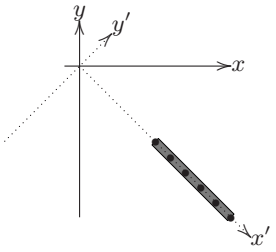

Fig. 3. Parallelotope abstraction with axes rotated by 45 degrees.

correct implementation of parallelotope operators. For the program in Figure 1, OSCA finds the change of basis matrix $\begin{bmatrix} 1 & 1 \\ -1 & 1 \end{bmatrix}$ whose columns correspond to the axes (x', y') in Figure 3. Whereas the standard analysis on the interval domain is not able to discover any invariant for the `while`-statement, the parallelotope domain is able to find out that $x + y = 0$ and the combined analysis finds out that $x \geq -1, y \leq 1, x + y = 0$, at the program point ①.

1 Using the tool

We have implemented in the R programming language a tool which performs the following steps:

1. Given a program written in an imperative fragment of the R language, the tool instruments the program in order to collect variables's values in all the program points.
2. On the collected values, the tool computes the PCA (using the standard R library), which is afterward refined to get the OSCA. The result is a matrix which describes the (hopefully) optimal basis.
3. The tool performs a static analysis of the program using intervals, parallelotopes and their combination. As a result, it returns a set of assertions for each program point.

The easiest way to use the tool is to start the R interactive environment, load the tool and the program to analyze, and use the function `compare.analyses`. When the function to analyze has no arguments, it is enough to use:

```
compare.analyses( <function name> )
```

If the function requires some arguments, we need to provide user-supplied values. These are not needed for the static analysis, but as input for the instrumented program. User-supplied values are passed in the second argument of

`compare.analyses` as a list of value assignments, where each value assignment is a map from variable names to values. For instance, if we want to analyze the example program `xyline`, using the input values $10, 20, 50$ for x, we write

```
compare.analyses(xyline,list(list(x=10),list(x=20),list(x=50)))
```

Note that in R the type `list` is used both for lists and maps.

The result of `compare.analyses` is a list with five components. The first two components are the matrices generated by PCA and OSCA. In our case:

	x	y
PC1	0.7072070	0.7070065
PC2	−0.7070065	0.7072070

	x	y
PC1	−1	1
PC2	1	1

The other three components are the results of the static analyses with the domains of boxes, parallelotopes and their combination. The tool returns a set of assertions for each program point, which are generally displayed as an annotated program.

```
"[ y=0 ]"
assume(x > 0)
"[ 0<=x , y=0 ]"
y = -x
"[ 0<=x , y<=0 ]"
while ({
    "[ ]"
    x > y
}) {
    "[ ]"
    x = x - 1
    "[ ]"
    y = y + 1
    "[ ]"
 }
"[ ]"
```

```
"[ ]"
assume(x > 0)
"[ ]"
y = -x
"[ x+y=0 ]"
while ({
    "[ x+y=0 ]"
     x > y
}) {
    "[ -x+y<=0 , x+y=0 ]"
    x = x - 1
    "[ -x+y<=1 , x+y=-1 ]"
    y = y + 1
    "[ -x+y<=2 , x+y=0 ]"
}
"[ 0<=-x+y , x+y=0 ]"
```

```
"[ y=0 : ]"
assume(x > 0)
"[ 0<=x , y=0 : -x+y<=0 , 0<=x+y ]"
y = -x
"[ 0<=x , y<=0 : -x+y<=0 , x+y=0 ]"
while ({
    "[ -1<=x , y<=1 : -x+y<=2 , x+y=0 ]"
    x > y
}) {
    "[ 0<=x , y<=0 : -x+y<=0 , x+y=0 ]"
    x = x - 1
    "[ -1<=x , y<=0 : -x+y<=1 , x+y=-1 ]"
    y = y + 1
    "[ -1<=x , y<=1 : -x+y<=2 , x+y=0 ]"
}
"[ -1<=x<=0 , 0<=y<=1 : 0<=-x+y<=2 , x+y=0 ]"
```

The analysis with the box domain does not depend on the result of the PCA. In this case, the analyses is not able to determine any constraints, if not the trivial ones before the beginning of the `while`. For the parallelotope domain, the axes are rotated according to the change of basis matrix in the second component, and therefore the domain is able to express intervals of the form $m \leq -x + y \leq M$ and $m \leq x + y \leq M$. The tool shows that, at the end of the program, the constraints $0 \leq -x + y$ and $x + y = 0$ hold, but it cannot prove any upper bound for $-x + y$. Finally, the domain which combines boxes and parallelotopes enhances the precision of both analyses.

The function `compare.analyses` takes many optional parameters which may heavily modify the result of the analyses. For example, the parameter `vars` allows to specify the list of variables to be considered during the analysis. The standard behaviour includes all the variables in the program since, for most domains, considering more variables (and thus more relationships) improves the result of the static analysis. Our tool shows that, in some cases, reducing the

space of variables may considerably improve the precision of the parallelotopes and combined domains.

For example, consider the standard bubblesort program on the right. If we perform an analysis with the standard parameters, the combined domain proves that `"[ 0<=b , 0<=j , 0<=t : 0<=b ]"`. The result of the OSCA is the matrix

```
function(k) {
  b = 100
  while (b>=1) {
    j=1
    t=0
    while (j<=(b-1)) {
      if (k[j]>k[j+1]) {
        tmp = k[j+1]
        k[j+1] = k[j]
        k[j]=tmp
        t=j
      }
      j=j+1
    }
    if (t==0) return(k)
    b=t
  }
  return(k)
}
```

	b	j	t	tmp
PC1	0	−1	−1	1
PC2	0	1	0	1
PC3	0	−1	2	1
PC4	1	0	0	0

It is worth noting that the variable *tmp* is included in the first three simple components, although it contains values from the array k, hence it is not correlated to the variables b, j and t which are used to index the array.

If we perform the analysis with the option `vars=c("b","j","t")` which excludes the variable *tmp*, we get the change of basis matrix:

	b	j	t
PC1	0	1	1
PC2	0	−1	1
PC3	1	0	0

and the combined domain is able to find more precise constraints:

"[1<=b<=100 , 0<=j<=100 , 0<=t<=99 : 0<=j+t<=199 , −100<=−j+t<=0 , 1<=b<=100]"

If the result of the statistical analysis of traces is not satisfactory, the tool has an option to provide a user-supplied change of basis matrix.

2 Implementation

The tool has been almost entirely implemented in R. This has at least three advantages. First of all, the analyzed language is R itself, and not an ad-hoc, artificial language. The second advantage is that we exploit metaprogramming on R, viewing a program both as a list and a function. Finally, R is very well-suited for statistical applications and manipulation of vectors, which are native types. On the contrary, the main disadvantage is that the performance of the analyzer in R is not comparable to other analyzers', since R uses a call-by-value semantics and is not well-suited for manipulating complex data structures. Anyway, we believe that it is a good choice for rapid prototyping.

The program to be analyzed is instrumented by inserting, at each program point, a call to a function which collects the values of variables. The same function can also interrupt the program after a certain number of steps. The option `whileonly` considers only a single program point for each while cycle, just before checking the while guard. From several experiments, it does not seem to make a lot of difference.

The instrumented program is executed and the collected values are stored in a matrix, which is fed to the native function *prcomp* which computes PCA. The resulting matrix is then refined by the OSCA, that we have implemented by scratch, since, at the best of our knowledge, there exists no available implementation. The resulting change of basis matrix is the input for the static analysis. Since static analysis must return only correct results, we need to ensure that numerical approximations do not introduce any error. In the case of the box domain, it is enough to appropriately round the result of operations in such a way that boxes are always overapproximated. To this aim, we have written a small foreign procedure (in C language) to change the floating point rounding mode of the CPU. For the parallelotope and combined domains, we have used exact rational arithmetic through the GMP library. We also wrote a wrapper library, to support infinite values.

3 Conclusion

This is the first tool which uses partial trace information for feeding a subsequent static analysis. The tool is still a prototype, which should be improved in many ways. We may use techniques of code coverage to improve the quality of partial execution traces. We may partition the set of program variables into groups and perform PCA separately on each group. We may also partition the program code itself, and perform a different PCA on each partition. As a future work, the tool could also be extended with different statistical methods, in order to discover better bases, and with a user-friendly front-end, especially for parameter tuning. Moreover, porting the code of the parallelotope domain to a faster programming language, possibly within some well-known library such as `APRON` [6] or `PPL` [3], would make it available to a wider community, while improving performance. Finally, the tool is available at the web page `http://www.sci.unich.it/~amato/random`.

References

1. G. Amato, M. Parton, and F. Scozzari. Deriving numerical abstract domains via principal component analysis. To appear in *Proc. Static Analysis Symposium*, 2010.
2. K. Anaya-Izquierdo, F. Critchley, and K. Vines. Orthogonal simple component analysis: a new, exploratory approach. To appear in the *Annals of Applied Statistics*, 2010.
3. R. Bagnara, P. M. Hill, and E. Zaffanella. The Parma Polyhedra Library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. *Science of Computer Programming*, 72(1–2):3–21, 2008.
4. P. Cousot and R. Cousot. Static determination of dynamic properties of programs. *Proc. Int'l Symposium on Programming*, pp. 106–130, 1976.
5. P. Cousot and R. Cousot. Abstract interpretation and applications to logic programs. *The Journal of Logic Programming*, 13(2–3):103–179, 1992.
6. B. Jeannet and A. Miné. APRON: A library of numerical abstract domains for static analysis. *Proc. Computer Aided Verification*, *LNCS* 5643, pp. 661–667, 2009.