Advanced Program Analyses and Verifications

Thi Viet Nga NGUYEN François IRIGOIN
Centre de Recherche en Informatique - Ecole des Mines de Paris
35 rue Saint Honoré, 77305 Fontainebleau Cedex, France
email: nguyen,irigoin@cri.ensmp.fr

***Abstract*— Software validation and verification are critical for almost systems whose failure is unacceptable. This is a very expensive process, typically costing 50% of the total software development costs. We show in this paper how program analyses can be used automatically to perform efficient and effective verifications, e.g *array bound checking* and *uninitialized variable checking*. These verifications are based on *array regions*, the analyses that collect information about the way array elements used and defined by programs.**

With useful knowledge of the program properties provided by these static analyses, the amount of additional code necessary to instrument the application is reduced to a minimum. Run-time checks are generated only when we do not know how to prove that they are useless. Dynamic errors that may cause potentially failure are caught and their consequences can be reduced by exception handling. Information provided by our analyses is helpful for the debugging process of large scale programs. All verifications are implemented in PIPS, the research compiler developed at École des Mines de Paris. Qualitative and quantitative experiments have done with industrial applications up to hundreds of thousand lines of code.

Keywords: array bound checking, compilation, program analysis, software verification, static analysis, uninitialized variable.

I. Introduction

Program analysis aims at statically discovering properties of the run-time behavior of a program without actually running it. Information obtained from this static analysis can be used to help program comprehension, program debugging, compiler optimization, program documentation and to assist program verification. The latter is essential to guarantee the reliability, robustness, and quality of software systems. It tries to prove the program correctness or to reduce risks that can cause the system to fail. A well known example of system failure is the explosion of Ariane 5, a 10 year project costing $7 billions, that was caused by a software error in the computer responsible for calculating the rocket's movement.

Software verification is an expensive process, typically costing upwards of 50% of the total software development costs, even 80% in some safety-critical systems such as nuclear reactors, air traffic control systems, embedded systems, telecommunication. There are many verification techniques which, by their definition, foundations and application domains are different. They include software testing, theorem proving, model checking and program analysis.

We develop in this paper precise and efficient program analyses that have a potential to reduce the verification costs and to increase quality of large scale software systems. With useful knowledge of the program properties provided by these analyses, the amount of additional code necessary to instrument the application is reduced. Run-time checks are generated only when we do not know how to prove that they are useless. Dynamic errors that may cause potentially failure are caught and their consequences can be reduced by exception handling. The specific applications studied in the paper are array bound and uninitialized variable checking. These verifications are performed by using array regions, advanced array element accesses analyses. All the applications are implemented in PIPS [6], a source-to-source Fortran compiler. It consists of several program analyses dealing with call graph computation, dependences, transformers, preconditions, use-def chains, array regions and of program transformations such as loop transformations, constant folding, dead code elimination, partial redundancy elimination.

The paper is organized as follows. Section 2 describes array region analyses. Section 3 presents the array bound checking method and Section 4 the uninitialized variable checking. Conclusions are given in the last section.

II. Array Region Analyses

Array region analysis collects information about the way array elements used and defined by programs [4], [3]. There are initially two kinds of array regions, READ and WRITE regions, that represent the effects of program statements on array elements. IN regions are introduced to represent the order in which references to array elements are executed, i.e array data flow information. They summarize the set of array elements whose values are *imported* by the current piece of code. One array element is imported by a fragment of code if there exists at least one use of the variable whose value has not been defined earlier in the fragment itself.

A region has the approximation MUST if every element in the region is accessed with certainty, and the approximation MAY if its elements are simply potentially accessed. The approximation of a region is EXACT if the region exactly represents the requested set of array elements. We have MUST $\subseteq$ EXACT $\subseteq$ MAY.

In PIPS, array regions are approximated by convex polyhedron [10] containing affine equalities and inequalities. These constraints link the region parameters that represent the array dimensions to the value of the program integer scalar variables. For example, `WRITE-EXACT-{`ϕ_1`=I, `ϕ_2`=2}` is the WRITE region of array `A` in assignment `A(I,2)=5`. The region parameters ϕ_1 and ϕ_2 respectively represent the first and second dimensions of `A`.

Regions are built from elementary statements to compound statements in the program control flow graph, and interprocedurally through procedure calls. At each meet point of a control flow graph, the region information from different control branches are merged with a convex hull operator. The approximation of regions is conservative.

A. Elementary Statement

Each array reference in an assignment is converted to an elementary region. This region is EXACT if and only if the subscript expressions are affine functions of the program integer scalar variables. The effect depends on the reference position in the statement. The IN regions of an assignment are equivalent to its READ regions. Regions of the same array and the same effect are merged by using the convex hull operator. If the resulting region contains array elements that do not belong to the original regions, it becomes a MAY region.

B. Conditional Statement

The READ/WRITE regions of a conditional statement contain the READ/WRITE regions of the true and false branches, which are restrained by the test condition. The regions of the test condition are added to the READ regions of this conditional statement.

The IN regions of a conditional statement contain the READ regions of the test condition, plus the IN regions of the true branch if the test condition is evaluated true, or the IN regions of the false branch if the test condition is evaluated false.

C. Loop Statement

The READ/WRITE region corresponding to the body of a loop is a function of the loop index value. During the propagation of regions, we need to unify READ/WRITE regions corresponding to different, but successive, instances of the loop body in order to get the summary region of the loop. This union operation is equivalent to eliminating the loop index i from the system which consists of the region predicate and the constraint on the iteration space: $lb \leq i \leq ub$. The operation is exact if the lower and upper bounds of the loop index are affine functions and the elimination of loop index is exact.

The IN regions of a loop contain array elements imported by each iteration but not previously written by the preceding iterations. Given the IN and WRITE regions of the loop body, the loop IN regions contain the imported array elements of the loop condition, plus the imported elements of the loop body if this condition is evaluated true. Then, when the loop is executed again, in the program state resulting from the execution of the loop body, they are added to the set of loop imported array elements in which all elements written by the first execution are excluded.

D. Sequence of Statements

Array regions at different program points correspond to different program states. To obtain READ/WRITE array regions of a sequence of statements, we must translate array regions associated to different statements to the state preceding the sequence. The translation consists of adding to the predicate of the region the constraints abstracting the effects of the statement between the two states. Then variables of the original state are eliminated. Only variables referring to the destination state are kept in the resulting polyhedron. The elimination of variables from a convex polyhedron may introduce integer points that do not belong to the actual projection. In this case, we have a MAY region.

The IN regions of a sequence $s_1; s_2; ..s_n;$ contain all elements imported by the first statement s_1, plus the elements imported by $s_2; ..s_n;$ after the execution of s_1, but not written by the latter.

E. Interprocedural Array Regions

The interprocedural propagation of regions is a reverse invocation order traversal on the call graph, i.e the callees are analyzed first. The summary region of a procedure is computed by eliminating local effects from the region of the procedure body. At each call site, the summary regions of the called subroutine are translated from the callee's name space into the caller's name space, and then used during the intraprocedural analysis of the callers.

The running example in Figure 1 is used to illustrate array regions and the READ and WRITE regions computed for this example are given in Figure 2. Array B

```
SUBROUTINE SUB(A,B,M,N)
REAL A(N), B(N,M)
K = FOO()
DO I = 1,N
  DO J = 1,N
    B(J,K) = J + K
  ENDDO
  K = K + 1
  DO J = 1,N
    B(J,K) = J*J - K*K
    A(I) = A(I) + B(J,K) + B(J,K-1)
  ENDDO
ENDDO
END
```

Fig. 1. Running Example

is written by the first J loop and the WRITE region of this loop is <B(ϕ_1,ϕ_2)-WRITE-EXACT-{1<=ϕ_1<=N,ϕ_2==K}>. B is also written and read by the second J loop, we have the corresponding loop regions: <B(ϕ_1,ϕ_2)-WRITE-EXACT-{1<=ϕ_1<=N,ϕ_2==K}> and <B(ϕ_1,ϕ_2)-READ-EXACT-{1<=ϕ_1<=N,K-1<=ϕ_2<=K}>. Since the value of K is incremented by 1 between the two loops, after propagating up the regions of the second J loop, the WRITE region of B becomes <B(ϕ_1,ϕ_2)-WRITE-EXACT-{1<=ϕ_1<=N,ϕ_2==K+1}> and the union of two WRITE regions of B is: <B(ϕ_1,ϕ_2)-WRITE-EXACT-{1<=ϕ_1<=N,K<=ϕ_2<=K+1}>. Information is lost when propagating up the regions of the outside loop I, because the value of K (K = FOO()) is unknown. Regions of B become MAY.

Figure 3 shows the IN regions computed for the running example. In the body of the second J loop, array elements A(I), B(J,K) and B(J,K-1) are imported by the second statement. Since B(J,K) is previously defined by the first statement, only A(I) and B(J,K-1) are imported by the loop body. The IN regions of the second J loop are: <B(ϕ_1,ϕ_2)-IN-EXACT-{1<=ϕ_1<=N,ϕ_2==K-1}> and <A(ϕ_1)-IN-EXACT-{ϕ_1==I}>. The value of K is incremented by 1, so after propagating upward the above IN regions through statement K = K + 1, the region of array B becomes: <B(ϕ_1,ϕ_2)-IN-EXACT-{1<=ϕ_1<=N, ϕ_2==K}>. Once again, all array elements in this region are defined by the first J loop, so only array elements of A are imported by the code fragment in this example.

```
    SUBROUTINE SUB(A,B,M,N)
    REAL A(N), B(N,M)
C   <A(φ1)-READ-EXACT-{1<=φ1<=N}>
C   <A(φ1)-WRITE-EXACT-{1<=φ1<=N}>
C   <B(φ1,φ2)-READ-MAY-{1<=φ1<=N}>
C   <B(φ1,φ2)-WRITE-MAY-{1<=φ1<=N}>
    K = FOO()
C   <A(φ1)-READ-EXACT-{1<=φ1<=N}>
C   <A(φ1)-WRITE-EXACT-{1<=φ1<=N}>
C   <B(φ1,φ2)-READ-EXACT-{1<=φ1<=N, K<=φ2<=K+1}>
C   <B(φ1,φ2)-WRITE-EXACT-{1<=φ1<=N, K<=φ2<=K+1}>
    DO I = 1,N
C   <B(φ1,φ2)-WRITE-EXACT-{1<=φ1<=N, φ2==K}>
      DO J = 1,N
C   <B(φ1,φ2)-WRITE-EXACT-{φ1==J, φ2==K}>
        B(J,K) = J + K
      ENDDO
      K = K + 1
C   <A(φ1)-READ-EXACT-{φ1==I}>
C   <A(φ1)-WRITE-EXACT-{φ1==I}>
C   <B(φ1,φ2)-READ-EXACT-{1<=φ1<=N, K-1<=φ2<=K}>
C   <B(φ1,φ2)-WRITE-EXACT-{1<=φ1<=N, φ2==K}>
      DO J = 1,N
C   <B(φ1,φ2)-WRITE-EXACT-{φ1==J, φ2==K}>
        B(J,K) = J*J - K*K
C   <A(φ1)-READ-EXACT-{φ1==I}>
C   <A(φ1)-WRITE-EXACT-{φ1==I}>
C   <B(φ1,φ2)-READ-EXACT-{φ1==J, K-1<=φ2<=K}>
        A(I) = A(I) + B(J,K) + B(J,K-1)
      ENDDO
    ENDDO
    END
```

Fig. 2. READ and WRITE Regions of the Running Example

```
    SUBROUTINE SUB(A,B,M,N)
    REAL A(N), B(N,M)
C   <A(φ1)-IN-EXACT-{1<=φ1<=N}>
    K = FOO()
    DO I = 1,N
      DO J = 1,N
        B(J,K) = J + K
      ENDDO
C   <B(φ1,φ2)-IN-EXACT-{1<=φ1<=N, φ2==K}>
C   <A(φ1)-IN-EXACT-{φ1==I}>
      K = K + 1
C   <B(φ1,φ2)-IN-EXACT-{1<=φ1<=N, φ2==K-1}>
C   <A(φ1)-IN-EXACT-{φ1==I}>
      DO J = 1,N
C   <B(φ1,φ2)-IN-EXACT-{φ1==J, φ2==K-1}>
C   <A(φ1)-IN-EXACT-{φ1==I}>
        B(J,K) = J*J - K*K
        A(I) = A(I) + B(J,K) + B(J,K-1)
      ENDDO
    ENDDO
    END
```

Fig. 3. IN Regions of the Running Example

III. Array Bound Checking

Array bound checking refers to determining whether all array references in a program are within their declared ranges. Such checking is desirable for any program, regardless of the programming language used, since bound violations are among the most common programming errors. Subscripting arrays beyond their declared sizes may produce unexpected results, security holes or failures. But on the other hand, bound checking can be very expensive because every array access must be guarded by two bound checks per dimension to assert the legality of the access. This increases the size of the executable file, the compilation time and the execution time. In addition, other code transformations or optimizations are prevented by these bound checks.

Some compilers solve this problem by providing the user with a compile-time option to enable or disable the checking. The purpose of this option, as mentioned by [8], is to allow users to enable checking in the development and debugging processes, and then, once all the defects are supposedly found and fixed, to turn it off for the production version. However, bound checking is just as important for delivered versions of programs as for development versions because the production ones may have bugs that were not even observed during testing phases. Instead of providing a way to turn bound checking off, what is needed is to optimize it so that it has a minimal overall cost.

Our approach to array bound checking uses READ and WRITE array regions analyses to verify if no array bound violation will occur at run-time in some section of code, tests are unnecessary for this section. If it is proven that an access violation might occur, tests are generated as needed. The algorithm begins with the largest compound statement and if we have a global answer about array element accesses for this statement, we do not have to go down into its sub-statements. The analysis can stop here, and bound checks can be inserted at the very beginning of the module entry and outside loops if we have sufficient information. So at a compound statement, the READ and WRITE regions of each array are used to test the feasibility of the corresponding array bound checks.

1) If the region is a MAY or EXACT region included in the declared dimensions of the array, no bound check is needed for the compound statement and we stop the process for the array here;
2) If the region is a MUST or EXACT region that contains elements which are outside the declared dimensions of the array, there is certainly a bound violation. An error is detected at compile-time;
3) If the region is an EXACT region and it is possible to project all pseudo ϕ variables from the systems builded from the region and the bound checks ($\phi_i < lower_i$ or $\phi_i > upper_i$), we have unavoidable tests to insert before the compound statement. The projection of one variable from a system of constraints is a linear programming operation, performed by using the Fourier-Motzkin method. Each bound check is accompanied with a stop message that tells the user in which array and on which dimension the subscript is out of range. The process stops here for the array;
4) Otherwise, we go down to the sub-statements of the current compound statement, take the regions of the concerning array and repeat the above steps. We have global vari-

ables to store information that the lower or upper bound of a dimension of an array has already been checked or not. If so, we do not have to check for this bound when going down to the sub-statements.

The algorithm terminates because we can always generate bound checks directly for array references of elementary statements in the control flow graph. Figure 4 shows inserted unavoidable tests for the running example. At the module entry, the regions of array A are in the declared range (1<=ϕ_1<=N) so no bound check is needed for this array. Only the first dimension of B is verified to be in the declared dimension, we have to go down after the assignment K = FOO() where the regions of B are EXACT. Lower and upper bound checks are generated for the second dimension, which are outside loops.

```
SUBROUTINE SUB(A,B,M,N)
REAL A(N), B(N,M)
K = FOO()
IF (M.LE.K) STOP 'Violation:array B,upper 2'
IF (K.LE.0) STOP 'Violation:array B,lower 2'
DO I = 1,N
  DO J = 1,N
    B(J,K) = J + K
  ENDDO
  K = K + 1
  DO J = 1,N
    B(J,K) = J*J - K*K
    A(I) = A(I) + B(J,K) + B(J,K-1)
  ENDDO
ENDDO
END
```

Fig. 4. Array Bound Checking for the Running Example

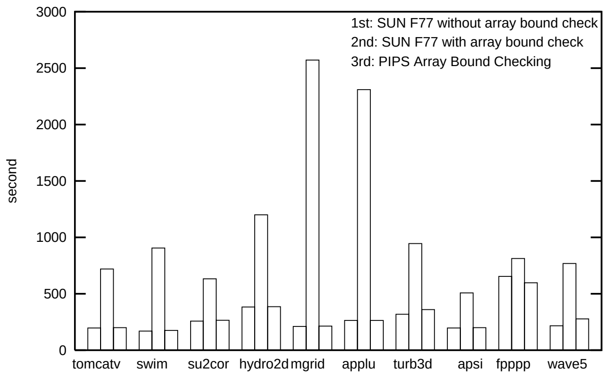

Fig. 5. Execution time: SUN F77 and PIPS

We also applied the method to the SPEC95 CFP benchmarks [5] and had very good results in comparison with other commercial compilers and related work. The average percentage of checks removed at run-time is 97%, the compilation is faster then the SUN Workshop F77 compiler version 5.0 with its bound checking option. The execution time compared with this compiler is shown in Figure 5. The execution times of programs with range checking added by PIPS are slightly longer than that of the unsafe programs without bound checks, about 5.33%. The speed-up is 4.37 times when comparing the execution time of the codes with our bound cheks and that of SUN.

The purpose of our method is to generate a minimum number of bound checks using the available information from array regions. Bound checks are inserted outside loops and at the beginning of the program. The other advantage of this algorithm is that it detects the sure bound violations or indicates that there is certainly no bound violation as early as possible, thanks to the context given by the top-down analysis of insertion of tests.

IV. Uninitialized variable checking

Used-before-set refers to the error occurring when a program uses a variable which has not been assigned a value. This uninitialized variable, once used in a calculation, can be quickly propagated throughout the entire program and anything may happen. The program may produce different results each time it runs, or may crash for no apparent reason, or may behave unpredictably. This is also a known problem for embedded software.

To detect this kind of error, two approaches can be used: compile-time analysis and run-time checking. However, compile-time analysis is far from perfect because complicated data and control flows result in a very large imprecision. Furthermore, the use of global variables and arrays with non-linear, indirection subscripts, etc sometimes makes static checking completely ineffective, leading to many spurious warnings. On the other hand, pure dynamic checking is costly due to heavy code instrumentation while information available at compile-time is not taken into account. The slowdown between instrumented and uninstrumented codes has been measured by up to 130 times [7]. In addition, run-time checking only validates the code for a specific input. Dynamic checking is not so effective that, as shown in a report comparing Fortran compilers [2], only some Lahey/Fujitsu and Salford compilers offer run-time checking for all kinds of variables. The other compilers do not have this option, or if so, they only check for local and formal scalar variables. Array and global variables are omitted.

With the growth of hardware performance - processor speed, memory bandwidth - software systems have becomed more and more complicated to solve better real application problems. Debugging several million lines of code becomes more difficult and time-consuming. Execution time overheads of dynamic checking is not highly appreciated. Automatic compile-time analysis to prove the safety of programs, to detect statically program errors, or to reduce the number of run-time checks is necessary. The question is, by using advanced array analyses, can static analysis be an adequate answer to the used-before-set problem for the scientific codes? If not, can a combination of static and dynamic analyses reduce the cost of uninitialized variable checking? The goal of our research is to use the imported array region analysis to detect uninitialized variables, and if sufficient information is not available, run-time checks are added to guarantee the program correctness.

The uninitialized variable checking is directly based on IN array regions. These regions are computed for arrays, but scalar variables also carry the same kind of information which is cheaper to compute. In fact, the region of a scalar has an empty predicate, i.e <V−IN−EXACT−{}>. Our algorithm traverses the program call graph in the invocation order, the callers are analyzed firstly. The list of IN regions at the module entry statement gives us the set of all possibly undefined variables of the module, and vice-versa, only variables in this list may be used before set. So at this statement, if the list of IN regions is empty, there is no used-before-set error. Otherwise, each variable in

the list is checked. Depending on the variable type (local, formal or global) and the current module, we have two cases:
Case 1: Local variable or global variable in the main program. Depending on the region approximation, we have two sub-cases:

- If the IN region has the approximation MUST or EXACT, the variable must be used somewhere in the module before being defined and an error is detected.
- Otherwise, the region is MAY, we instrument the code by inserting an initialization function before the module statement and go down to the sub-statements where the IN region is propagated from. Before each statement where we know with certainty that the variable must be imported, a verification function is inserted. We continue to go down for each statement with MAY regions. If this statement is a procedure call, information is added to mark that the corresponding formal parameters of the actual variable, or the corresponding global variables must be checked at the callee's level. To help the debugging process, information about the call path is added to locate the run-time error.

Case 2: Formal parameter or global variable in a called procedure. If this variable is marked as must be checked, we repeat the process as for local variables, but no initialization is needed since it has been performed earlier in one of the callers.

The efficiency of our analysis depends on the accuracy of array region analyses. The more precise the imported array regions are, the smaller the number of variables to be checked is. By using IN regions, we limit the number of variables to be checked, and code instrumentation is only used when we do not have enough information. Only array elements in the MAY IN regions are initialized and checked. PIPS initialization and verification statements are inserted in the source code and the program is then compiled and executed normally. The executable code appears to the user to operate as the original, but if a used-before-set error is detected, the program is stopped with a message to indicate the name of the variable, the module and the call path where this error occurred. Figure 6 shows the result of used-before-set analysis on the running example. There is only the IN region of array `A` at module entry. If we have the calling context the know that `A` is already initialized, there is no test. Otherwise, tests are inserted as shown in the comment lines. It is also noted that our experiments on the SPEC95 CFP becnhmarks show several used-before-set errors in the *wave5* benchmark but the uninitialized values are not propagated so they have no effect on the benchmark result.

V. CONCLUSION

The paper demonstrates that program analysis can be a powerful tool for improving software security. Static analysis is profitably combined with code instrumentation to reduce execution overhead. With available information provided by static analysis, only some code fragments requiring checking are instrumented. This method also makes it easy to perform error checking at the optimal time, during the development process. In comparison to other program verification techniques, our approach is completely automatic and do not need human interactions such as user annotations and assertions. Static analysis

```
   SUBROUTINE SUB(A,B,M,N)
   REAL A(N), B(N,M)
C  DO PHI = 1, N
C    IF is_nan(A(PHI)) STOP 'A is undefined'
C  ENDDO
   K = FOO()
   DO I = 1,N
     DO J = 1,N
       B(J,K) = J + K
     ENDDO
     K = K + 1
     DO J = 1,N
       B(J,K) = J*J - K*K
       A(I) = A(I) + B(J,K) + B(J,K-1)
     ENDDO
   ENDDO
   END
```

Fig. 6. Uninitialized Variable Checking for the Running Example

can be used to build an effective tool for detecting errors in large-scale systems at reasonable cost.

Our approach to array bound checking inserts unavoidable checks directly by using array region analyses. The number of generated tests is limited; range checks are put only where there might be bound violations. The main advantage of this top-down analysis is that it detects the sure bound violations or indicates that there is certainly no bound violation as soon as possible. By using imported array regions, only a small number of maybe uninitialized variables pointed by our compile-time analysis. This analysis can help the testing and validation process to save debugging time. Used-before-set errors are caught at run-time to avoid memory violation problems such as incorrect results, unpredictable behavior or system crashes.

Related work in the domain is discussed and compared in a more detailed work [9]. Our approaches to program verification could also be applied to other imperative languages for scientific applications that require software verification such as Ada, Java, etc. The PIPS software and documentation as well as the implementations are available on *http://www.cri.ensmp.fr/pips*.

REFERENCES

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers Principles, Techniques, and Tools*. Addison-Wesley, Massachusetts, 1986.

[2] J. Appleyard. Comparing Fortran compilers. *ACM SIGPLAN - Fortran Forum*, 20(1):6–10, 2001.

[3] B. Creusillet. *Analyses de régions de tableaux et applications*. PhD thesis, Centre de Recherche en Informatique, Ecole des Mines de Paris, December 1996.

[4] B. Creusillet and F. Irigoin. Interprocedural array region analyses. In *International Workshop on Languages and Compilers for Parallel Computing*, volume 1033 of *Lecture Notes in Computer Science*, pages 46–60. Springer-Verlag, 1995.

[5] J. J. Dujmovic and I. Dujmovic. Evolution and evaluation of SPEC benchmarks. *ACM SIGMETRICS*, 26(3):2–9, 1998.

[6] F. Irigoin, P. Jouvelot, and R. Triolet. Semantical interprocedural parallelization: an overview of the PIPS project. In *International Conference on Supercomputing*, pages 144–151, June 1991.

[7] A. Loginov, S. H. Yong, S. Horwitz, and T. W. Reps. Debugging via run-time type checking. In *Fundamental Approaches to Software Engineering*, pages 217–232, April 2001.

[8] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, San Francisco, 1997.

[9] T. V. N. Nguyen. *Efficient and Effective Software Verifications for Scientific Applications using Static Analyses and Code Instrumentation*. PhD thesis, Ecole des Mines de Paris, 2002.

[10] A. Schrijver. *Theory of Linear and Integer Programming*. John Wiley & Sons, Chichester, 1986.