Functional Equivalence Checking for Verification of Transformations on Array-Intensive Programs

K.C. Shashidhar

Departement Computerwetenschappen, Katholieke Universiteit Leuven, Belgium

Email: kodambal@cs.kuleuven.ac.be; Expected graduation: September 2005.

Advisors: Prof. Maurice Bruynooghe and Prof. Francky Catthoor

In this work, we have developed an efficient, fully automatic method for formal verification of source code transformations that is based on checking the functional equivalence of the original and the transformed programs.

MOTIVATION

Development of energy and performance-efficient embedded software, for mobile computing and communicating systems, is increasingly relying on application of complex transformations on the critical parts of the source code. Typically, these transformations are applied on the algorithmic kernels in the software, for example, signal processing algorithms, numerical methods etc. The transformations mainly affect the data-flow by restructuring the control loops and modifying the index expressions of array variables in the program. Designers applying such transformations, mostly manually or using ad-hoc tools, are often faced with the problem of ensuring the *correctness of application* of transformations. Currently, this problem is addressed by incomplete and time-consuming simulation based testing. Clearly, it is instead desirable to address the problem by formal verification of the transformed code against the original.

FOCUS

A pragmatic approach to the motivated verification problem is to check the functional equivalence of the original and the transformed source codes. In fact, the associated problem in hardware synthesis (RTL-level and below) has been very successfully addressed by equivalence checking methods. But to emulate this success in software synthesis, given that program equivalence problem is in general undecidable, it is necessary to focus the problem to a decidable set of programs and transformations, that are still relevant to the application area of interest.

Allowed Programs. In signal processing applications, structured, imperative, array-intensive pro-

grams are common. The critical functions requiring rigorous optimization in these programs are often restricted to a program class with certain properties, viz., (a) single-assignment form, (b) affine expressions in the subscripts of arrays and loop bounds, (c) only static allocation of data; and (d) simpler data dependent control-flow. On the one hand, these properties make it easier to analyze and transform the programs, and on the other, there are automatic methods that can convert most programs to have these properties. Our method assumes that the original and the transformed programs have these properties.

Allowed Transformations. When programs are in the mentioned class, it is decidable to check their equivalence under important program transformations like (1) loop transformations, (2) expression propagations; and (3) algebraic transformations. These transformations are commonly applied in order to achieve dramatic reductions in accesses to the data memory, and hence, the running time and energy consumption of the programs. In practice, they are also the ones that are most error prone to apply as they require manipulation of complex index expressions of array variables and the expressions in the bounds of the iterative loops. Our method is targeted to these transformations.

SOLUTION

For the class of programs and transformations that lie in our focus, we have developed an equivalence checking method [4, 3, 2, 1] that takes the original and the transformed programs, as input and checks whether the two programs are functionally input-output equivalent.

Method in Brief. The equivalence checking method is based on reasoning on the data dependencies in the two programs. It uses a program representation called an array data dependency graph (ADDG) that captures all the data dependencies in a program at the level of operators in its statements.

The information about data dependencies is represented in terms of polyhedral domains and mappings in closed form. Based on a synchronized traversal on the ADDGs of the original and the transformed programs, the method checks that they both have (I) identical sequences of operators and (II) identical data dependencies, for each corresponding paths in the computation between the observable variables. This is a sufficient condition for equivalence and the ADDGs can be suitably normalized under the considered transformations to check this condition. The traversal starts from each pair of the corresponding outputs and proceeds to the inputs by accounting for identical operators for corresponding paths and updating the data dependency relationship from the output. To address algebraic transformations, when operators with known algebraic properties (e.g., associativity, commutativity, ...) are reached, the checking normalizes the ADDGs and establishes corresponding paths to proceed with the traversal. Whenever the traversal reaches inputs on both sides, the data dependency relationship between the output and the input arrays for the corresponding paths are checked for identity. When the checking fails, the method suitably locates the errors in the text of the transformed program.

Error Diagnostics. The checker may fail to prove equivalence under two situations, corresponding to the two parts of the sufficient condition. In either case, the checker provides diagnostics for failure.

(1) The first situation arises when the operator nodes on the two corresponding sides do not match. In this case, the checker stops and points to the exact location of the mismatching statements in the two programs.

(2) The second situation arises when the number of elements, that should correspond on either side, do not match. In this case, the checker stops and reports the assignment statements on both sides that should collectively account for all the elements and the difference in the range of elements that causes the mismatch.

Features. The key features of our method are: (1) it is oblivious of the type, number and order of particular transformations that have been applied on the original program; (2) the checking is done in a single pass of the traversal of the ADDGs, therefore, equivalences are established only once for points of correspondence and reused; and (3) the checking can be restricted, if desired, to only certain parts of interest in the programs.

Experience. We have implemented the method in a prototype tool that accepts programs in ANSI C

Figure 1: The verification and debugging scheme.

language. Its use case is as shown in Figure 1. With this tool, verification of programs with 1000 lines of C code of control complexity comparable to what is seen in kernels of real-life program codes has been possible in less than 100 seconds on a standard desktop. The tool has also been able to help debug and gain confidence in the correctness of in-house source code transformation tools.

NOVELTY OF OUR SOLUTION

When compared to the related work discussed in our publications, the key differences of our solution are: (1) it targets source-to-source transformations that no other single method handles; (2) it does not unroll loops; (3) it does not require any hints from the user; and (4) it pays attention to generation of error diagnostics.

References

- [1] K. C. Shashidhar, M. Bruynooghe, F. Catthoor, and G. Janssens. An automatic verification technique for loop and data reuse transformations based on geometric modeling of programs. *Journal of Universal Computer Science*, 9(3):248–269, 2003.
- [2] —. Automatic functional verification of memory oriented global source code transformations. In *8th Intl. Workshop on High Level Design Validation and Test (HLDVT)*, pages 31–36. IEEE, 2003.
- [3] —. Functional equivalence checking for verification of algebraic transformations on array-intensive source code. In *Design, Automation and Test in Europe (DATE)*. IEEE, 2005.
- [4] —. Verification of source code transformations by program equivalence checking. In *Intl. Conf. on Compiler Construction (CC)*. LNCS Springer, 2005.

Publications are available at:

<http://www.cs.kuleuven.ac.be/~kodambal/pubs.html>