Automatic \textit{A Posteriori} Verification of Certain Global Source-code Restructuring Transformations

K.C. Shashidhar*
IMEC vzw, Kapeldreef 75, B-3001, Heverlee, Belgium
kodambal@imec.be

Program optimization process is sometimes justifiably referred to as a \textit{necessary evil}, for it raises the question of correctness of the optimized program, which is often at stake. In order to lend credence to the process, it is required that effective verification and testing methods and tools are well in place. Many solutions exist which address the various facets of this requirement. But for many practical situations, the solutions on offer are clearly less than satisfactory; in our work we address one such situation.

Our endeavour is to provide a fully automatic technique to verify certain state-of-the-art optimizing program transformations applied on the multi-dimensional data dominated kernels of signal processing algorithms during the design of multimedia applications for embedded systems. We have made some progress and developed and implemented a technique that provides a posteriori proof of functional equivalence of the initial and transformed programs for some important transformations [3]. The technique is based on extracting and reasoning on the geometric domains representing the dependencies between the output and the input variables, which are preserved under the transformations considered. It also provides useful error diagnostics to identify and debug the program.

In what follows, we discuss the intuitive idea behind our method.

1 Problem Statement and an Example

The problem is to provide a technique to automatically verify the functional equivalence of the initial and the transformed sequential programs. We suppose that the latter has been obtained by only applying one or more of the following transformations:

1. Structure preserving and/or modifying loop transformations
2. Introduction and/or elimination of intermediate variables
3. Transformation of data-structures of intermediate variables and
4. Data-flow transformations involving expression propagation.

Given the application context of such a verification technique, we are allowed to assume that the programs are in dynamic single assignment form (see Vanbroeckhoven’s article in this proceedings), implemented in a subset of a C-like imperative language, devoid of pointers and non-linear loop iterator and array index expressions.

The following toy example is an input-output equivalent program fragment pair which is representative of the class of real-life programs whose equivalence we want to check. The example shows some of the transformations listed above applied in an intermixed fashion. But, both the programs will assign the same values to the output variable \texttt{out[0]} at the end of execution for any given data to the \texttt{in[0]} variable. Now the question is: how can we automatically verify this?

```c
{ for(i=0; i<128; i++)
    tmp[i] = f(in[i]);   /* n1 */
    for(i=128; i>127; i--) 
    out[i] = g(f(in[i])); /* n2 */
    out[i] = g(tmp[i]);   /* n3 */
} 
```

```c
{ for(i=0; i<64; i++)
    buf1[i] = in[i];     /* t1 */
    for(i=0; i<256; i++)
        if((i>=0) && (i<64))
            buf2[i] = f(buf1[i]); /* t2 */
            out[i] = g(buf2[i]); /* t3 */
        else
            out[i] = g(f(in[i])); /* t4 */
    }
```
2 Motivation and Related Work

Application-driven and architecture-driven code transformations for program cost-optimization, more often than not, refers to transformations on the source-code. The term source-to-source is more commonly used to stress this abstraction level. There are many advantages in working at the source code level and the motivation therein is beyond the scope of this article. Naturally, many methodological design frameworks, targeted to certain specific application domains, advocate transformations of high level executable specification written in languages like C, SystemC, etc. Our own motivation comes from the Data Transfer and Storage Exploration (DTSE) design framework [1].

Complete automation of application and architecture specific transformations in a high end compiler is prohibitively complex and has more or less been understood to be infeasible. As a result, application of global source-to-source transformations on programs is essentially an interactive activity with many apply 'n' check cycles; eventually converging on an acceptable design meeting the cost constraints. Employing a tool for the application of the transformations in such an activity is an a priori solution (proposed by many) to the transformation correctness problem, but it is not practicable for various reasons. Firstly, a customized transformation tool is rarely available for the given context. Even if it is available, it is usually inflexible to work with, as allowing only a predefined set of transformations is very restrictive. The designers desire the flexibility of applying a transformation that is not in the set when they see a clear gain in applying it. Secondly, when it is available and also extendable, requirement of skill comes in the way. Designers usually lack the skill required to add new transformations to the existing set with the required formal proofs. Finally, the correctness of the tool at hand itself is questionable. Although a predefined set of transformations may have been proven to be formally correct, often there is only a prototype implementation and there is no guarantee that the prototype is a correct implementation. As a result, in practice, designers, supported by some analysis tools, manually apply complex transformations, relying mainly on a combination of application know-how, experience and ingenuity. A proof of equivalence by a separate tool can substantially increase the confidence that the functionality is preserved, and this warrants the need for a posteriori methods.

Equivalence checking, in general, is well-known to be undecidable. Therefore, an automated check has to be based on a decidable condition that is sufficient for equivalence between initial and transformed programs. If the condition holds, the transformation is safe, ensuring the equivalence; otherwise, not much can be concluded. In the latter case, to be useful, the check should be able to pinpoint a reasonably small program fragment that is at the origin of the failure to prove equivalence.

The recent work on translation validation [4], addresses a very related problem of a posteriori validating whether the target code produced by a compiler is a correct translation of the source program, providing an alternative to the verification of translators/compilers. But in this technique, a trade-off exists between the class of transformations that can be checked and the extent of compiler instrumentation that is required to provide enough information to the validator about the transformations applied. Our attempt is to provide a source-to-source transformation verification infrastructure and is complementary to translation validation.

3 Transformation Verification Framework

The technique is an implementation of the scheme shown in Figure 1. Given the initial and transformed program pair, the geometrical models are extracted using static analysis from the two programs. The models are formulas that encode affine constraints on integer variables, logical connectives and quantifiers, also called Presburger formulas, symbolically representing the dependency mapping between the defined variable and the operand variables. Verification involves forming the appropriate necessary and sufficient equivalence conditions and checking them for each mapping in the corresponding assignment statements in the two programs.

To be able to find correspondence between the assignment statements in the two programs, in the presence of non-matching intermediate variables and distributed expressions, it is necessary to canonicalize the statements defining the output variables by eliminating the intermediate variables. Since we forbid algebraic data-flow transformations, once the statements are in canonical form, there has to be a corresponding match between the two programs in order to have equivalence of computation. After identifying the correspondences, we use OMEGA calculator [2] to evaluate our checks. If a condition does not hold, the transformed program is debugged with the error diagnostics generated as a result of invalidity of the condition. The equivalence checking itself is done completely oblivious of any information about either the particular transformations that have been applied or the order in which they have been applied. As a result, the check provides a proof of equivalence that is independent of the agent applying the transformations.
Owing to lack of space, we provide only a very brief illustration of the technique on our toy example: the initial program has `tmp[]` as an intermediate variable and the transformed program has `buf1[]` and `buf2[]` as intermediate variables. Once we have extracted the mappings for each of the statements in the initial program, we calculate back the dependency of the output variable `out[]` on the input variable `in[]` by taking composition of the mappings across statements, essentially eliminating the intermediate variables. The resulting canonical signature of the computation defining `out[]` on `in[]` along with the associated dependency mapping between them should have an exact corresponding match in an assignment statement extracted similarly from the transformed program. There is only one canonical signature of computation for the initial program i.e., \( \text{out}[] = g(f(\text{in}[])) \) with the dependency mapping \( \mathcal{M} := \{ [i] \rightarrow [i] \mid 0 \leq i \leq 255 \land i \in \mathbb{Z} \} \) between `out[]` and `in[]`. The transformed program provides a match for this signature with the mapping \( \mathcal{M}' \). A check on the equivalence of the two mappings i.e., \( \mathcal{M} = \mathcal{M}' \), holds, proving that the two programs assign exactly same values to `out[]`.

4 Implementation and Present Status

We have implemented our technique in **GMV - Geometric Model Verifier**, a prototype tool which integrates calls to the geometrical model extractor and the **OMEGA** and coordinates the constructed checks and provides error location information to the user. **GMV** has successfully verified transformations applied on some real life examples with many complex loops and multi-dimensional arrays, like implementations of signal processing application cores like Durbin, MPEG-4 motion estimation kernel and updating singular value decomposition (USVD) algorithm. The verification was possible in a push-button style and took time only in the order of few seconds. In the USVD case, it detected a bug in the transformed USVD code (400 lines of code), which was traced to a bug in the constant propagation unit of the code generator that a prototype loop transformation tool used.

Presently we are investigating the particular case of a cyclic recursive dependency of an array variable on itself in the presence of computation. Currently we do not allow transformation of statements in such a cyclic dependency. A solution to this case, consistent with the rest of the analysis, will render the technique complete for the listed set of transformations. In future work we will investigate extension of the framework to handle algebraic data-flow transformations and develop a tight approximation to handle programs with non-linear loop iterator and array index expressions.

References


