An optimizing compiler applies a sequence of transformations on an input program to obtain a semantically equivalent output program that uses fewer resources; semantic equivalence is preserved between the source and the transformed programs at each step. Recent advancement of multi-core and multi-processor systems has enabled incorporation of concurrent applications in software systems for better performance and resource utilization. In general, writing a parallel program is harder than writing the corresponding sequential programs. Hence, parallelizing compilers are generally used for this purpose to generate appropriate code for given parallel architectures from sequential programs. Till date, fully automated parallelization steps used in a compiler do not guarantee correctness by construction. Moreover, they are not optimal. Hence, such transformations are often done in a semi-automated manner with programmer assistance underscoring the importance of verifying such steps. With the increasing relevance of the parallelization process in the prevalent high performance computing systems, the need to verify the parallelization step is felt even more keenly. One approach towards achieving this goal could be to formally verify a compiler so that each of its runs is guaranteed to yield an output (transformed) program which is equivalent to the input program. Proving correctness of a complex program like a compiler, however, is immensely difficult, if not impossible; hence, the approach of transformation validation is followed whereupon for every run of the compiler its input source program is proved to be equivalent to its output transformed program.

The present work describes a transformation validation approach to establish equivalence of the source program and the generated parallelized program to validate parallelizing transformations. To achieve this goal it seeks to leverage as its back end a method of checking equivalence of sequential programs on arrays which is the most sophisticated one among those reported in the literature which are capable of handling non-uniform recurrences. However, if the recurrence expressions of the source and the transformed programs refer to more than one non-equivalent base case consequently partitioning the domains of the output arrays of the source and the transformed programs, then the equivalence checking mechanism fails to report non-equivalent portion of the output arrays precisely. In the present work, a mechanism is proposed to precisely indicate the equivalent and non-equivalent portions of the output arrays for the programs with linear recurrences. Furthermore, to use this method, the present work devises a mechanism for constructing dependence graphs for loop parallelized programs (with or without $wait-signal$ statements) and vectorized programs. Finally, a method to indicate the most likely location of the fault in the source program is proposed which works if the source and transformed programs are identified to be inequivalent by the back end method. The method is also able to indicate the most likely sequence of enabling transformations which may have been applied (wrongly) to obtain the transformed program.

Keywords: Verification, Dependence Graph, Conflict Access Graph, Equivalence Tree, Equivalence Checking.