An Enhanced Equivalence Checking Method to Handle Bugs in Programs with Recurrences

Sudakshina Dutta and Dipankar Sarkar
Indian Institute of Technology Kharagpur, Kolkata, India

Keywords: Equivalence Checking, Recurrence, Dependence, Narrowing.

Abstract: Software designers often apply automatic or manual transformations on the array-handling source programs to improve performance of the target programs. Verdoolaege et al. (Verdoolaege et al., 2012) have proposed a method to automatically prove equivalence of the output arrays of the source and the generated transformed programs. Unlike the other approaches, the method of (Verdoolaege et al., 2012) provides the most sophisticated techniques to validate programs with non-uniform recurrences besides programs with uniform recurrences. However, if the recurrence expressions of the source and the transformed programs refer to more than one base cases of which some are non-equivalent and also if the domain of the output arrays partition based on dependences on different base cases, then some imprecision in the equivalence checking results is observed. The equivalence checker reports that the entire index spaces of the output arrays of the source program to be non-equivalent with that of the transformed program instead of the portion of the output arrays which depend on the non-equivalent base cases of the programs. In the current work, we have enhanced the method of equivalence checking of (Verdoolaege et al., 2012) so that it can precisely indicate the equivalent and non-equivalent portions of the output arrays.

1 INTRODUCTION

Due to resource constraints of embedded processors, developers apply aggressive loop and data transformations on the source program and generate the target program. With the increase of usages of embedded processors in high performance computing system, there is a growing need to verify the correctness of such transformations which are primarily applied by compilers. For validation of such transformations, dependence graph (DG) oriented mechanisms are more suitable than the control graph oriented ones because the domain of applications of such transformations involve arrays and the dependence computations of some array elements on other elements. Unlike the previously reported methods (Shashidhar et al., 2005), (Karfa et al., 2011), the DG oriented mechanism reported in (Verdoolaege et al., 2012) has been found to be sophisticated enough to handle many loop transformations with recurrences and it is applied for the case of programs with static control flow and piecewise affine expressions for all loop bounds, conditions and array index expressions.

However, if the recurrence expressions of the source and the transformed programs refer to more than one base cases some of which are non-equivalent and also if the domain of the output arrays partition based on dependences on different base cases, then equivalence checking mechanism fails to report non-equivalent portion of the output arrays precisely. The equivalence checker instead reports that the entire index spaces of the output arrays of the source program to be non-equivalent with that of the transformed program. Although the problem of formally verifying programs is undecidable, we have enhanced the method of equivalence checking of (Verdoolaege et al., 2012) so that it can precisely indicate the equivalent and non-equivalent portions of the output arrays for the programs with linear recurrences. From the program, the recurrence expression is extracted and it is solved for both the equivalent and non-equivalent base cases to form generalized conjectures of "proved" (equivalent) and "not proved" (not equivalent) cases. Later the conjectures are proved following the equivalence checking mechanism reported in (Verdoolaege et al., 2012).

Dutta, S. and Sarkar, D.
An Enhanced Equivalence Checking Method to Handle Bugs in Programs with Recurrences.
In *Proceedings of the 11th International Conference on Evaluation of Novel Software Approaches to Software Engineering (ENASE 2016)*, pages 254-259
ISBN: 978-989-758-189-2

2 MOTIVATION

Consider the example in Fig. 1. In this example, two non-equivalent programs with recurrences are shown where the output arrays in the programs are D. The only difference in the two programs are in the assignment statements s_2 and s_2'. The elements of the output array D are non-equivalent if the iterator value is even, else they are equivalent. However, the tool (isa-0.12) available with (Verdoolaege et al., 2012) reports the following:

Equivalence proved: $\{D[3]\}$

Equivalence NOT proved: $\{D[i] : 4 \leq i \leq N\}$

Although the tool reports above equivalence results, the algorithm of (Verdoolaege et al., 2012) theroticaly returns the following results:

Equivalence NOT proved: $\{D[i] : 3 \leq i \leq N\}$

The above line indicates that equivalence cannot be proved for the entire range of iterator values $3 \leq i \leq N$. To remove this imprecision in the output of the equivalence checker, we have enhanced the method of (Verdoolaege et al., 2012) in the present work. The output of the enhanced equivalence checker is the following:

Equivalence proved: $\{D[i] : 3 \leq i \leq N$ *and* $\exists \alpha$ *s.t.* $i = 2\alpha + 1\}$

Equivalence NOT proved: $\{D[i] : 3 \leq i \leq N$ *and* $\exists \alpha$ *s.t.* $i = 2\alpha\}$

```
s1 : A[1] = 2                    s1' : A[1] = 2
s2 : A[2] = 3                    s2' : A[2] = 9
do i = 3,N                       do i = 3,N
  s3 : A[i] = 2 * A[i-2]           s3' : A[i] = 2 * A[i-2]
  s4 : D[i] = A[i]                 s4' : D[i] = A[i]
end do                           end do
```

(a) (b)

Figure 1: Two non-equivalent code snippets.

3 MODEL OF EQUIVALENCE CHECKING

A *dependence graph* is a connected labeled directed graph $G = \langle V, E, I, V_o \rangle$ with vertices V, each of which involves a single arithmetic operation f of a statement s of the program, and edges E, called dependences among the vertices (or more precisely, their operations). There is a set of vertices $V_o \in V$ with in-degree 0 corresponding to an output array and a set $I \subset V$ of vertices corresponding to input arrays, each with out-degree 0. A vertex v is associated with the iteration domain D_v of the surrounding loop (if any) of the corresponding statement of the program. An edge $e = \langle v_1, v_2 \rangle$ starts from a vertex v_1 and ends in a vertex v_2 of the DG and is associated with a mapping M_e from some subset of domain D_{v_1} to the subset of the domain D_{v_2} capturing the dependence of the operation f of v_1 on the value of the operation f of v_2.

In Fig. 2, the DGs of the the code snippets of Fig. 1 are shown. In Fig. 2(a), v_1, v_2, v_3, $\cdots$, v_8 depict the vertices of the DG and $\langle v_1, v_1' \rangle$, $\langle v_2, v_3 \rangle$, etc. represent some of the edges of the DG. The output vertex v_1 indicates the output array D of the program. The vertices v_4, v_7 and v_8 represent the input vertices and they represent the reference to the constant values of the array $\mathbb{Z}$ in statements s_3, s_1 and s_2, respectively. The *id* operations in vertices v_2, v_5, v_6 represent *copy* operations in statements s_4, s_1 and s_2, respectively. The domains of the vertices are indicated beside the vertices. As the assignment s_4 is executed inside the loop with iteration domain $3 \leq i \leq N$, the domain of the vertex v_2 is indicated as $[3, N]$. The edges represent the read after write dependence among the domain elements of the vertices of the DG and it is represented by the mapping of the edges. The edge $\langle v_2, v_3 \rangle$ represents that the i^{th} iteration of the assignment statement s_4 reads the value assigned to the i^{th} iteration of the statement s_3 if $3 \leq i \leq N$. The self-loop $\langle v_3, v_3 \rangle$ represents the recurrence present in the statement s_3; consequently, the mapping $\{v_3(i) \rightarrow v_3(i-2) | 5 \leq i \leq N\}$ depicts the dependence of the i^{th} iteration of the second argument of the multiplication operation on the value generated by the $(i-2)^{th}$ iteration of the left hand side of the same operation. The edges $\langle v_3, v_5 \rangle$ and $\langle v_3, v_6 \rangle$ depict the dependence on the 3^{rd} and 4^{th} iterations of the multiplication operation of s_3 on the assignment operations of the statements s_1 and s_2. The edge $\langle v_3, v_4 \rangle$ depicts the dependence of the first argument of the multiplication operation on the constant C_2 for the domain $[3, N]$.

4 OVERVIEW OF THE EXISTING VALIDATION METHOD

The method of checking equivalence of two programs (Verdoolaege et al., 2012) takes two DGs as inputs. It starts by pairing up the output array vertices of the two DGs and associating with the pair a goal which asserts that the elements of the output arrays have to be pairwise computed identically in both the programs. These correspondences are maintained in an equivalence tree (ET) which is dynamically constructed as the equivalence proof proceeds. The goal propagates as a predicate R^{want} in a lockstep fashion along the pair of edges of DGs until the input vertices of the DGs are encountered or the pair of vertices recur in

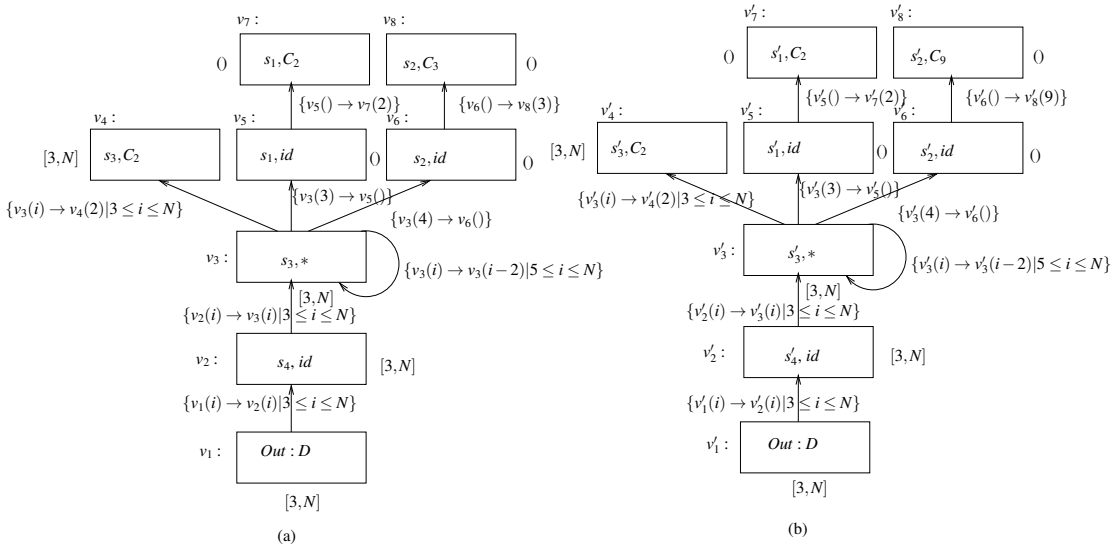

Figure 2: (a)DG of the program in Fig. 1(a), (b)DG of the program in Fig. 1(b).

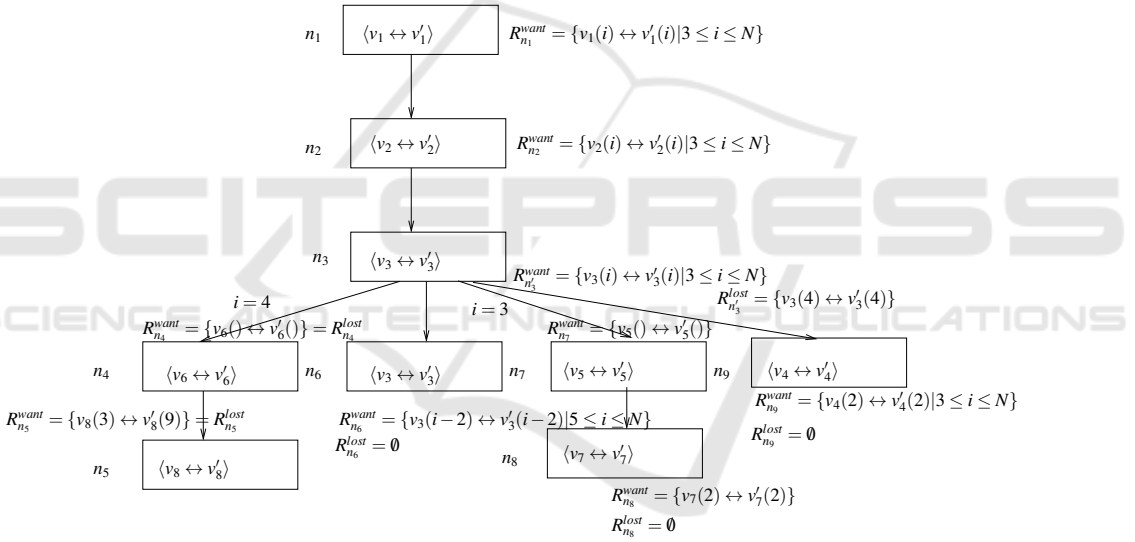

Figure 3: ET of the DGs shown in Fig. 2.

the ET. A subgoal is discharged as proved when reduced to an equivalence between the same elements of the input arrays. As forward propagation of goals stops, what cannot be proved as a predicate R^{lost} is propagated backwards and finally the subpart of what cannot be proved is shown with respect to the domains of the output arrays.

The ET construction process takes the DGs corresponding to Fig. 2 as inputs and starts with a root node $n_1 = \langle v_1, v_1' \rangle$ in Fig. 3. The node is associated with the predicate $R^{want}_{n_1} = \{v_1(i) \leftrightarrow v_1'(i) | 3 \leq i \leq N\}$ which represents the goal that the entire index spaces of the output arrays are pairwise identical. This goal propagates from vertices along the edges of the DGs. In this example, v_1 has the only outgoing edge $\langle v_1, v_2 \rangle$ and the entire domain $v_1(i)$ maps to $v_2(i)$, $3 \leq i \leq N$. Also, v_1' has the only outgoing edge $\langle v_1', v_2' \rangle$ the entire domain $v_1'(i)$ maps to $v_2'(i)$, $3 \leq i \leq N$. Hence, the goal propagates to the newly constructed node $n_2 = \langle v_2, v_2' \rangle$ with the goal $R^{want}_{n_2} = \{v_2(i) \leftrightarrow v_2'(i) | 3 \leq i \leq N\}$.

The goal, $R^{want}_{n_1}$, forward propagates following the mapping of the DG-edges as stated above. For proving equivalence of the output arrays $D[i]$, $3 \leq i \leq N$, the equivalence of $A[i]$, $3 \leq i \leq N$ has to be established due to the presence of the assignment statements s_4 and s_4'. The values of $A[i]$, $3 \leq i \leq N$, are determined by the multiplication operations of the statements s_3 and s_3'. Hence, the goal of proving equivalence of $D[i]$, $3 \leq i \leq N$ boils down to the goal of proving the

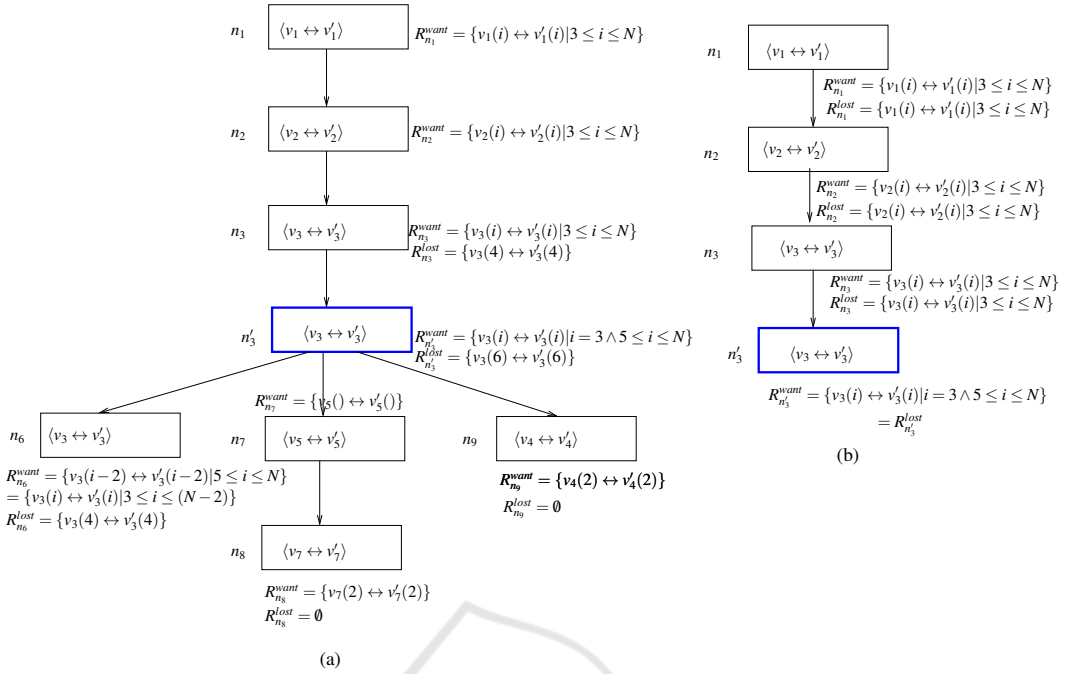

Figure 4: (a) ET of the DGs shown in Fig. 2 after the first narrowing, (b) ET of the DGs shown in Fig. 2 after the second narrowing (narrowed nodes in both the figures are shown in *blue* color).

result of the these multiplication operations for the range $3 \le i \le N$ and it is depicted by the ET-node $n_3 = \langle v_3, v'_3 \rangle$ with $R^{want}_{n_3} = \{v_3(i) \leftrightarrow v'_3(i) | 3 \le i \le N\}$. The equivalence of $A[i]$, $3 \le i \le N$ in statements s_3 and s'_3 can only be established if the equivalence of $A[i-2]$, $3 \le i \le N$ can be proved. For $i = 3$, the computation of $A[i]$ in the source and the target programs depend on the base cases denoted by the assignment operations of s_1 in the source program and s'_1 in the target program; this is depicted by the equivalence of the constant expressions C_2 in the ET-node n_8 with $R^{want}_{n_8} = \{v_7(2) \leftrightarrow v'_7(2)\}$. Similarly, for $i = 4$, the computation of $A[i]$ depends on the base case denoted by the assignment operation depicted by s_2 in the source program and s'_2 in the target program. This is depicted by the inequivalence of the assignment of constant expressions C_3 and C_9 in the statements s_2 and s'_2, respectively in the ET-node n_5 with $R^{want}_{n_5} = \{v_8(3) \leftrightarrow v'_8(9)\}$ and $R^{lost}_{n_5} = R^{want}_{n_5}$. For $5 \le i \le N$, the equivalence of $A[i]$ is dependent on the computation $A[i-2]$ evaluated in the same statements by multiplication operations due to the presence of recurrences in statements s_3 and s'_3; this is depicted by the node n_6. However, the equivalence of the entire range $A[i-2]$, $5 \le i \le N$ cannot be established as the inequivalent assignments $A[2] = 3$ in s_2 and $A[2] = 9$ in s'_2 are referred for computation of array elements $A[4]$ in the source and target programs, respectively.

As the array element $A[4]$ is not equivalent in the source and the transformed program, from the entire range of $A[i]$, $3 \le i \le N$, the element $i = 4$ is separated (referred to as first narrowing operation (Cousot and Cousot, 1992) in Fig. 4(a)) and the inequivalence of the elements $A[4]$ is depicted as $R^{lost}_{n_3} = \{v_3(4) \leftrightarrow v'_3(4)\}$. The process of establishing equivalence starts for the revised range $i = 3 \wedge 5 \le i \le N$. However, the equivalence of the entire range of the revised goal cannot also be established as for the range $5 \le i \le N$ the computation of $A[6]$ depends on the computation of inequivalent array elements $A[4]$ in the source and target programs. Hence, the entire range of $A[i]$, $3 \le i \le N$ is reported to be inequivalently computed and it results in the inequivalence of $D[i]$, $3 \le i \le N$ (referred to as second narrowing operation in Fig. 4(b)). Note that the first narrowing node n'_3 with revised goal $R^{want}_{n'_3} = \{v_3(i) \leftrightarrow v'_3(i) | i = 3 \wedge 5 \le i \le N\}$ is installed in Fig. 4(a) from which the forward propagation starts once again. After the second narrowing operation, the entire range of $R^{want}_{n'_3}$ is set to $R^{lost}_{n'_3}$. In the next step, $R^{lost}_{n_3} \cup R^{lost}_{n'_3} = \{v_3(i) \leftrightarrow v'_3(i) | 3 \le i \le N\}$ propagates back to n_1 and $R^{lost}_{n_1}$ is set to $\{v_1(i) \leftrightarrow v'_1(i) | 3 \le i \le N\}$.

5 SUGGESTED ENHANCEMENT

For proving equivalence of the output arrays $D[i]$, $3 \le i \le N$, the equivalence of $A[i]$, $3 \le i \le N$, has to be es-

tablished. For $i = 3$, the array elements $A[i]$ depend on the base case denoted by the equivalent assgiments of constant expressions C_2 on array elements $A[1]$ in the satements s_1 and s'_1. Hence, the array elements $A[3]$ are equivalently computed. For the $i = 4$, the array elements $A[i]$ depend on the base case denoted by the inequivalent assgiments of constant expressions C_3 and C_9 on array elements $A[2]$ in the statements s_2 and s'_2, respectively. Hence, the array elements $A[4]$ are inequivalently computed. For the rest of the domain $5 \leq i \leq N$, the computation of $A[i]$ depends on the computation of $A[i-2]$ due to recurrence. Instead of declaring the entire range of $A[i]$, $3 \leq i \leq N$, to be not proved to be equivalent in the source and transformed programs as (Verdoolaege et al., 2012), the recurrence relation depicted by the index expressions $A[i]$ and $A[i-2]$ (i.e., $i_m = i_{m-1} + 2$) are solved for the base cases $i = 3$ and $i = 4$ in our proposed method. The solution of the recurrence relation for $i = 3$ mapped to the range $3 \leq i \leq N$ ($3 \leq i \leq N \wedge \exists m \geq 0\ s.t.\ i = 3+2m$ i.e., the odd iterator values) is declared as the index range for which the array elements $A[i]$ are equivalently computed and the solution for $i = 4$ mapped to the range ($3 \leq i \leq N \wedge \exists m \geq 0\ s.t.\ i = 4+2m$ i.e., precisely the even iterator values) is declared as the index range for which the array elements $A[i]$ are inequivalently computed.

The ET-node n_3 represents the goal of proving equivalence of $A[i]$, $3 \leq i \leq N$, for proving equivalence of $D[i]$, $3 \leq i \leq N$ in Fig. 3. The ET-node n_6 represents the fact that for proving equivalence of $A[i]$, $5 \leq i \leq N$, the equivalence of $A[i-2]$ has to be established. The recurrence relations are formed using the method given below. In Fig. 3, we refer to the suffix expression i associated with the ancestor node n_3 as i_m and the suffix expression $i-2$ associated with its recurring descendant node n_6 as i_{m-1} thereby producing the recurrence $i_m = i_{m-1} + 2$; the inequivalence of the array element $A[4]$ contributes to the base case $i_0 = 4$ of the above recurrence. Thus, the recurrence equation $\{i_0 = 4,\ i_m = i_{m-1} + 2\}$ depicts the non-equivalent elements that would occur if we permit the ET-node n_6 to reduce further. The solution of the above-mentioned non-homogeneous recurrence equation is $i_m = 4 + 2m$; hence, the "not proved" elements of the ET-node n_3 is $\{v_3(i) \leftrightarrow v'_3(i) | 3 \leq i \leq N \wedge \exists m \geq 0\ s.t.\ i = 4+2m\}$. To represent such elements of n_3, a child $n_{3,2}$ is created with $R^{want}_{n_{3,2}} = \{v_3(i) \leftrightarrow v'_3(i) | 3 \leq i \leq N \wedge \exists m \geq 0\ s.t.\ i = 4+2m\}$ thereby setting the goal of proving equality of all the elements of $R^{want}_{n_{3,2}}$. The ET-node $n_{3,2}$ is then explored further to disprove the proof goal $R^{want}_{n_{3,2}}$ following the existing method.

The proposed method tries to generalize the proved cases next from Fig. 3 in a similar way as has been described above. The recurrence equation $\{i_0 = 3,\ i_m = i_{m-1} + 2\}$ is found using the same steps as described in the previous paragraph for the generalization of the proof goal for the equivalent base case for array elements $A[3]$. It depicts the "proved" elements that would occur if the ET-node n_6 is permitted to reduce further. The solution of the above-mentioned recurrence equation is $i_m = 3 + 2m$. Hence, the generalized "proved" elements at the ET-node n_3 is conjectured as $\{v_3(i) \leftrightarrow v'_3(i) | 3 \leq i \leq N \wedge \exists m\ s.t.\ i = 3 + 2m\}$. A child node $n_{3,1}$ of n_3 is created with the goal of proving the conjecture, i.e., $R^{want}_{n_{3,1}}$ is set as $\{v_3(i) \leftrightarrow v'_3(i) | 3 \leq i \leq N \wedge \exists m\ s.t.\ i = 3 + 2m\}$ as shown in Fig. 5. It is explored further to establish the proof goal $R^{want}_{n_{3,1}}$ using the existing method and $R^{lost}_{n_{3,1}} = \emptyset$. In the next step, $R^{lost}_{n_{3,2}} = \{v_3(i) \leftrightarrow v'_3(i) | 3 \leq i \leq N \wedge \exists m \geq 0\ s.t.\ i = 4 + 2m\}$ propagates back to root node n_1 resulting in $\{v_1(i) \leftrightarrow v'_1(i) | 3 \leq i \leq N \wedge \exists m \geq 0\ s.t.\ i = 4 + 2m\}$. This enhancement can also be used to precisely identify the equivalent and non-equivalent portions of the output arrays with one or more than one dimensions for the programs with linear recurrences and one or multiple base cases.

6 RELATED WORK

We have enhanced a novel, fully automated approach to the equivalence checking problem of static affine programs which uses abstract interpetation operator instead of using transitive closure operators. The two most closely related approaches are those of (Shashidhar et al., 2005), and (Karfa et al., 2011). In (Shashidhar et al., 2005), the authors proposed an ADDG based equivalence checking method to validate the loop transformations. The authors of (Karfa et al., 2011) redefine the equivalence of ADDGs to verify loop transformations along with a wide range of arithmetic transformations. The equivalence checking method relies on a normalization technique proposed in (Karfa et al., 2011) and some simplification rules to handle arithmetic transformations over arrays. As explained before, all of these approaches are based on transitive closures and therefore require uniform recurrences, unlike the abstract interpretaion based approach proposed in (Verdoolaege et al., 2012). We have enhanced the method for the programs with recurrences and some non-equivalent base cases.

7 CONCLUSION

We have enhanced the most sophisticated method of

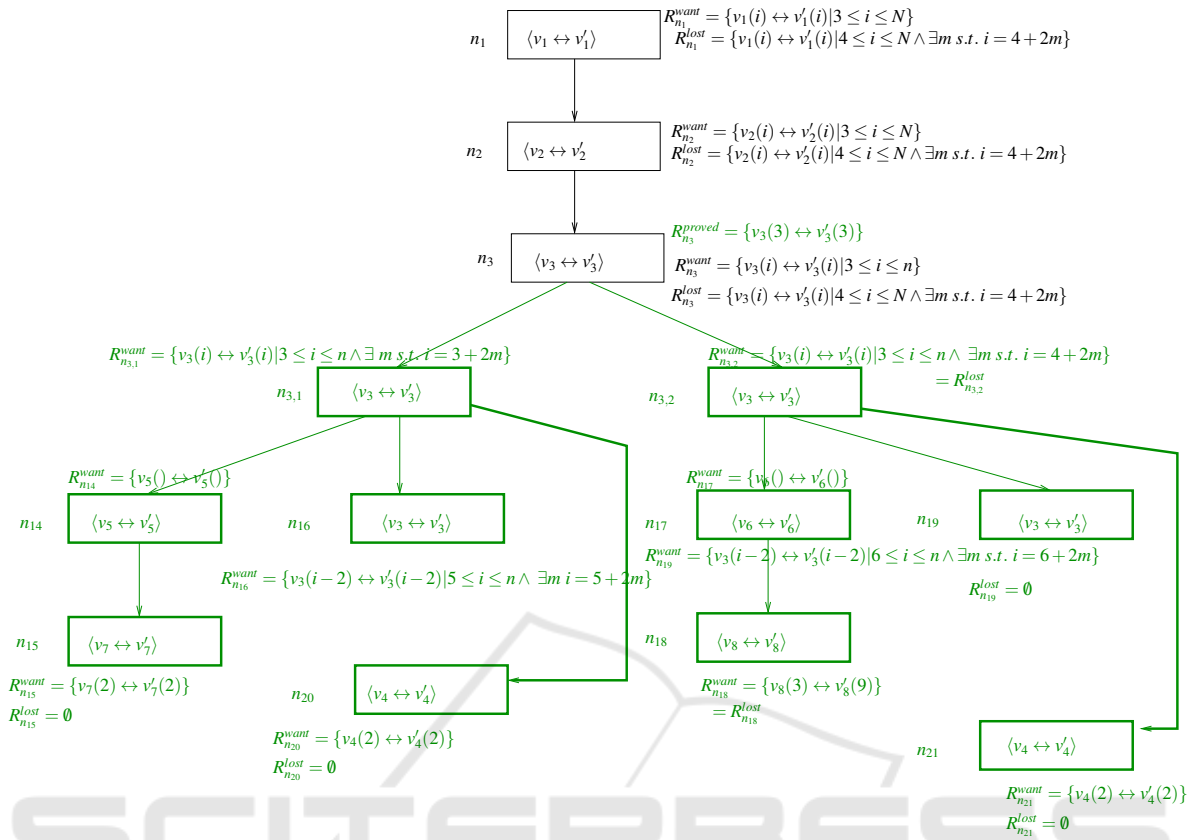

Figure 5: After enahancement, the final ET of the DGs shown in Fig. 2 after the forward and backward propagations are over.

checking equivalence of array-handling programs with recurrences. We are currently incorporating the method in the source code available with (Verdoolaege et al., 2012). The proposed method works on homogeneous or non-homogeneous linear recurrence eqtions. Also, we can precisely partition the domain of the output array to be "proved" and "not proved" using the method and are able to prove the generalized conjectures correctly. Unlike the method of (Verdoolaege et al., 2012), we can prove the "not proved to be equivalent" elements of the output arrays to be inequivalently computed. The proposed method has wide applications in signal processing domains where application programs are implemented with programs having recurrences.

REFERENCES

Cousot, P. and Cousot, R. (1992). Comparing the galois connection and widening/narrowing approaches to abstract interpretation. In *Proceedings of the 4th International Symposium on Programming Language Implementation and Logic Programming*, PLILP '92, pages 269–295, London, UK, UK. Springer-Verlag.

Karfa, C., Banerjee, K., Sarkar, D., and Mandal, C. (2011). Equivalence checking of array-intensive programs. In *VLSI (ISVLSI), 2011 IEEE Computer Society Annual Symposium on*, pages 156–161.

Shashidhar, K. C., Bruynooghe, M., Catthoor, F., and Janssens, G. (2005). Functional equivalence checking for verification of algebraic transformations on array-intensive source code. In *Design, Automation and Test in Europe, 2005. Proceedings*, pages 1310–1315 Vol. 2.

Verdoolaege, S., Janssens, G., and Bruynooghe, M. (2012). Equivalence checking of static affine programs using widening to handle recurrences. *ACM Trans. Program. Lang. Syst.*, 34(3):11:1–11:35.