Temporal Precondition Verification of Design Transformations

Ranga Vemuri and **Anuradha Sridhar**

Laboratory for Digital Design Environments
Department of Electrical and Computer Engineering
University of Cincinnati, ML 30
Cincinnati, Ohio 45221-0030, USA

Abstract: Design transformations are ubiquitous in design derivation systems. Many such transformations have elaborate conditions of applicability known as *preconditions*. Usually, preconditions have both spatial and temporal components. The temporal (components of the) preconditions are usually specified by associating a dynamic interpretation with the design description at hand. Such dynamic interpretations have a semantic content which is based on interpreting the design description over the domain of natural numbers. Thus the problem of precondition verification is just as difficult as the problem of design verification itself.

This paper is an informal exposition to the techniques we have used in verifying preconditions of the design transformations in a transformational exploration system for register-level hardware designs. These techniques are based on a purely syntactic interpretation of the design description and avoid the difficulties associated with the theorem proving techniques one could employ when a semantic interpretation is associated. While not as powerful as theorem proving methods, we found these techniques to be adequate for most cases of precondition verification of useful design transformations, at least within the design domain, namely register-level hardware, considered.

1 Introduction

Transformational design methods have been proposed for both program derivation (eg. [4]) and hardware design derivation (eg. [3, 5, 9]). In such systems, a design transformation can be applied to the current design to derive a functionally equivalent design. Many design transformations have elaborate preconditions which must be satisfied by the design before the transformation can be applied. For example, two values can be assigned to the same memory location, only if the *life-spans* of the values do not overlap; two ALU's can be folded into one only if they are never used at the same time and if the wires to connect the sources and destinations to the ALU's already exist. Clearly, preconditions of transformations have a *spatial* component and a *temporal* component. The spatial preconditions are predicated upon the static composition and connectivity among the various modules in the design description and are relatively easy to verify. The temporal preconditions are predicated upon a dynamic interpretation of the design representation. Such dynamic interpretation is usually based on the input data typically encoded as (streams of) natural numbers (or an equivalent domain of values) and on interpreting the design entities (such as the functional units) as functions over the domain of interpretation. Then, in general, automated verification of the temporal preconditions is as difficult as verifying design (against its functional specifications) itself.

While building a register-level design-space exploration system [7], we have developed several techniques for precondition verification of design transformations. These techniques do not use conventional theorem proving techniques directly and are not based on any semantic interpretation of the design representation. Rather, they exploit any information available about the set of possible

traces (execution paths) to effectively verify the preconditions. *Partial evaluation* is the main technique we used for this purpose. When combined with two other methods, namely using *symbolic equivalence* rules and *comparison rules*, we found that partial evaluation is a very effective technique for precondition verification. The preconditions themselves are specified in terms of what we call *t-expressions*. A *t-expression* essentially encapsulates all the time-steps at which the various *control points* in the design can be executed given different traces. Verification of temporal preconditions then amounts to determining the truth value of various predicates on one or more t-expressions. Specification of preconditions using t-expressions was the subject of an earlier work [7, 8]. In this paper, after providing an introduction to t-expressions, we concentrate on the evaluation of t-expressions while partial or no trace information is available.

In this paper, we use *flow chart programs* as the designs being manipulated in the design derivation system. Flow chart programs prescribe a sequencing of assignment statements and no-ops. An assignment statement can be executed in one time-step; like-wise a no-op can be executed in one time-step. The sequencing operators themselves do not consume any time-steps. [1] We do not explicitly deal with the internals of the assignment statement in the flow chart. It is enough to say that during an assignment statement several resources (such as ALU's, memory locations, wires etc.) are used. Some of these resources (such as ALU's) are occupied or live only during execution of the assignment statement. Other resources (such as memory locations) are occupied over many time steps (as long as the values stored in them are live). Assignment statements abstract register transfers in hardware design and program statements in software design. Similarly, no-ops represent control delays. (Manipulating the occurrences of no-ops is the primary way to schedule the design operations over the resources available.)

In Section 2 we introduce the flow charts. We introduce t-expressions which describe the time-step assignment to the control points in the flow charts. We describe a notation for describing execution traces. In Section 3 we describe the evaluation process of a t-expression at a given execution trace. In Section 4 we briefly discuss the symbolic equivalence of t-expressions. In Section 5, we introduce partial evaluation of t-expressions as a mechanism to verify the preconditions by utilizing any partial trace information that might be available to the designer. Often, in temporal preconditions we are only interested in relative scheduling of the resources rather than the absolute time-steps at which they are used. Verification of such preconditions involves comparing two t-expressions. We introduce several *comparison rules* to compare t-expressions through relation operators without evaluating the expressions. Section 7 contains a very brief sketch of our precondition verification system and two small examples to explain how these techniques fit together. The verification system itself is discussed in [6]; in this paper we concentrate on the underlying techniques.

2 Flow Charts, T-Expressions and Traces

Our *flow chart programs* (or, simply, flow charts) contain the following types of statements: *assignment*, *noop*, *parallel*, *case*, and *while*. A flow chart has exactly one *enter* node and one *exit* node. We use the icons shown in Figure 1 to write flow chart programs. Figure 1 also defines the syntax rules for the flow chart programs. Figure 2 shows an example flow chart program. Informally, an *assignment* statement is used to assign to one or more *variables* values computed using certain resources and the values currently stored in the variables. For the purposes of this paper, we are not concerned with the semantics or the internals of the assignment statement any further.

[1] It is straight forward to extend our techniques to incorporate the case where sequencing operators as well as assignments take multiple time-steps.

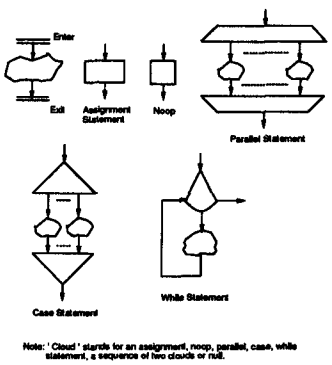

Figure 1: **Statements in Flow Chart Programs**

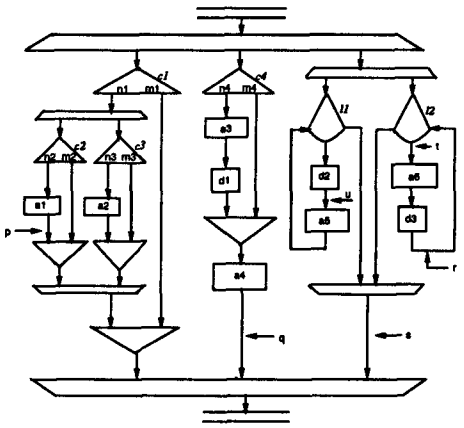

Figure 2: **An Example Flow Chart Program**

A *control point* is an edge in a flow chart. A *t-expression* is associated with each control point, as per the derivation rules shown in Figure 3. For the purposes of the t-expressions, we associate a unique symbol, called the *c-variable*, with each *case* statement and a unique symbol, called the *l-variable* with each *while* statement in the flow chart. (C-variables and l-variables are used to bind the trace information to the preconditions without making references to the execution paths themselves explicitly.) We also label each branch of each *case* statement with a unique symbol called the *branch id.* Using the derivation rules, the following t-expressions can be derived at the control points p, q, r and s of the flow chart shown in Figure 2:

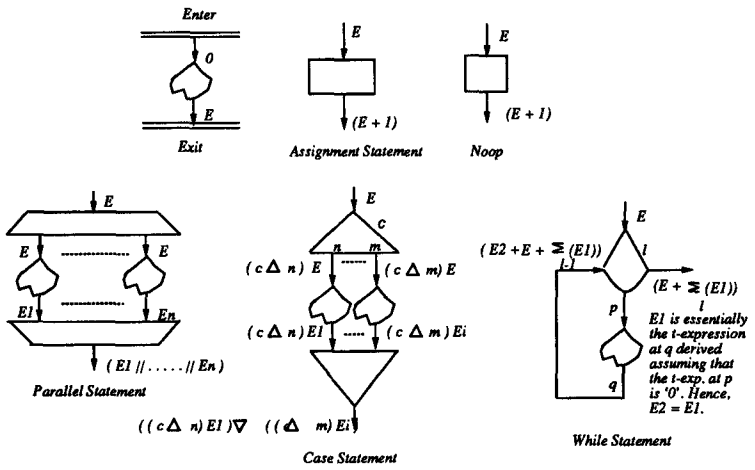

Figure 3: **T-Expression Derivation Rules**

$$E_p = (c2\Delta n2)((c1\Delta n1)0 + 1) \quad (1)$$

$$E_q = (((c4\Delta n4)(0 + 1 + 1))\nabla((c4\Delta m4)0)) + 1 \quad (2)$$

$$E_r = 0 + 1 + 1 + \sum_{l2-1}(0 + 1 + 1) \quad (3)$$

$$E_s = ((0 + \sum_{l1}(0 + 1 + 1)))\|((0 + \sum_{l2}(0 + 1 + 1))) \quad (4)$$

Informally, a *trace* of a flow chart is an execution path through the flow chart beginning at the *enter* node and ending at the *exit* node. [2] For example, `{a1,a4,d2,a6},{a5,d3},{d2,a6},{a5,d3},{a6},{d3}` is a trace of the flow chart shown in Figure 2. In this trace, *a1, a4, d2*, and *a6* are executed in the first step, *a5* and *d3* are executed in the second step and so on. We use a visual notation, called *trace diagrams*, to denote traces. Intuitively, the trace diagrams denote an assignment of integer values to l-variables and branch ids to c-variables. [3] For example, Figure 4 is the trace diagram corresponding to the above trace. The trace diagram indicates that the c-variable *c1* diverged on the branch id *n1* and in that *context* the c-variables *c2* and *c3* diverged on *n2* and *m3* respectively. It also shows that the l-variables *l1* and *l2* took the values 2 and 3 respectively indicating the number of iterations of the corresponding *while* statements.

[2]Note that a trace cannot be defined as a *path* in the graph theoretic sense because of the parallel statements whose branches are executed simultaneously.

[3]Only well-formed trace diagrams denote valid traces. To be well-formed, a trace diagram must assign an integer value to each l-variable for each context of invocation of the corresponding *while* statement and a *branch id* to each c-variable for each context of invocation of the corresponding *case* statement. In this paper we consider only well-formed trace diagrams. When in doubt, the reader should try to 'walk through' the trace denoted by the trace diagram.

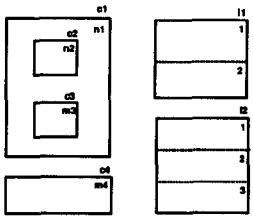

Figure 4: **Example Trace Diagram**

3 Evaluation of T-Expressions

A t-expression of a control point in a flow chart can be evaluated at any given trace of the flow chart to yield an integer. We use the following rules when evaluating the t-expressions:

1. '+' and '-' denote integer addition and subtraction respectively.
2. $\|$ denotes integer *'maximum-of'* operation.
3. $\sum$ denotes multiple instantiations.
4. Δ denotes selective instantiation:
   $(x \Delta y) = x \; if \; x = y$; $undefined \; otherwise.$
5. ∇ denotes mutual exclusion. For any given trace, all but one of the operands of the ∇ operator evaluate to 'undefined'. The value of the ∇ then is same as the value of that one operand.

For example, given the trace shown in Figure 4, the t-expressions (Eqs. 1-4) can be evaluated to, $E_p = 1$, $E_q = 1$, $E_r = 6$, $E_s = 6$. The evaluation can be better understood by viewing the snapshots of the decorated parse-trees of the t-expression E_q and E_s shown in Figure 5. The evaluation process involves iteratively substituting, at the leaves of the parse tree, the values of l-variables and c-variables and propagating these values up the tree using the evaluation rules given above until the root of the tree is assigned with an integer (or the special 'undefined' value). Note that new leaves are created during the reduction process due to the multiple instantiations resulting from evaluating the $\sum$ operator. These new leaves get appropriate values from the trace diagram. For any valid trace of the flow chart, any t-expression in the flow chart can be so evaluated [7, 8].

Informally, the integer to which a t-expression is reduced indicates the time-step (relative to the *enter* node) at which the corresponding control point is reached by a controller executing the flow chart. There is one complication however. The control points inside *while* statements may be traversed several times during the execution of the flow chart. This leads to the notion of *series evaluation* of the t-expressions for control points inside *while* statements.

A *series evaluation* of a t-expression of a control point inside a *while* statement yields a series of integers. If the *while* statement is not nested within any other *while* statement then the series evaluation procedure involves repeating, with $l = 0, 1, 2, \cdots lval$ where *lval* is the value assigned to the l-variable l by the given trace, the procedure illustrated in Figure 5. If the *while* statement is

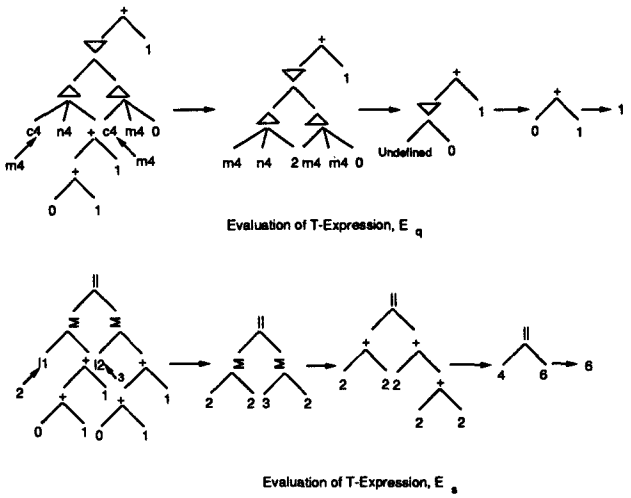

Figure 5: **Evaluation of T-Expressions**

nested within another *while* then this procedure is recursively applied. Each such evaluation of the t-expression yields an integer (or 'undefined' value) in the series. For example, given the trace in Figure 4, the t-expression E_r shown in the previous section evaluates to the series `[2,4,6]`.

Hence forth, when we say 'evaluation' of t-expressions we mean series evaluation for expressions of control points within a *while* statement the usual integer evaluation for other expressions.

4 Symbolic Equivalence

In general, evaluation of a t-expression requires a complete trace. However, during the process of design derivation no information about trace (or the set of possible traces) is available. Even so, it is sometimes possible to reduce the t-expressions using certain equivalence rules. Two t-expressions (or subexpressions) are equivalent if and only if, for any t-trace, both expressions evaluate to the same value. Following are some of the equivalence rules often used in our system: [4]

1. $(E \| E) = E$
2. $((c\Delta n)E\nabla(c\Delta m)E) = E$
3. $(((c\Delta n)i_1\nabla(c\Delta m)i_2))\|i_3 = i_3$, where i_1, i_2, and i_3 are integers and if $max(i_1, i_2) \leq i_3$

At any time during the evaluation process, an equivalence rule can be applied to reduce a subexpression of a t-expression into a simpler form. Examples illustrating the use of these rules appear in Section 7.

[4]Actually, these rules are used in some what more general forms where associativity and commutativity of some of the t-expression operators is implicitly taken into account.

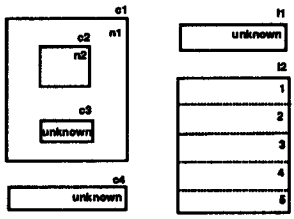

Figure 6: **Example Partial Trace Diagram**

5 Partial Evaluation

If partial trace information is available, it can be effectively used to reduce t-expressions through a technique known as *partial evaluation* [1]. Such partial trace information may be available in the design description or, more often in interactive design systems, through the designer. For example, for fixed iteration-depth loops (of the type *for i = 1, 100 do ()*) the values of the corresponding l-variables are known. In a flow chart with two successive *case* statements with m and n branches respectively, of the $m*n$ possible traces, the designer, knowing the values the variables on which the *case* statements are predicated assume, may be able exclude several. In general, partial evaluation of a t-expression yields a *residual* t-expression. The residual expression is subject to symbolic reduction as well as the other techniques discussed in this paper.

A partial trace is specified by a trace diagram in which a special *unknown* symbol is used to specify one's ignorance about the assignments of values to specific instances of the l-variables and c-variables. For example, Figure 6 shows a partial trace of the flow chart given in Figure 2.

The procedure for partial evaluation is similar to the total evaluation procedure discussed in Section 3 except that, in general, by the end of the evaluation process the root may not receive any value. The available l-values and c-values are substituted at the leaves and propagated through the internal nodes by applying as many t-expression evaluation rules as possible. At the end of this process, a part of the parse tree remains intact where as the other part is decorated and reduced. For example, the t-expressions 1-4 can be partially evaluated at the partial trace shown in Figure 6 to yield the following residual t-expressions:

$$
\begin{aligned}
E_p &= 1 \\
E_q &= (((c4\Delta n4)2)\nabla((c4\Delta m4)0)) + 1 \\
E_r &= 10 \\
E_s &= \sum_{l1}(2)\|(10)
\end{aligned}
$$

6 Comparison of T-Expressions

Most of the time, in verifying the preconditions, evaluation of one t-expression in itself is of little use. More important is to evaluate two t-expressions and compare the results. For example, to substitute

a resource r_1 used in an assignment statement s by another resource r_2 it is necessary to verify, for each execution of s, whether r_2 is being used elsewhere. This can be done by comparing the t-expression E associated with s and the t-expressions E_i associated with the assignment statements in which r_2 is used; the objective is to verify whether E and any E_i evaluate to the same value at some trace of the flow chart. Resource folding, where a resource can be a memory location, an ALU, a wire, a register etc., is the most important class of optimizing transformations found in design derivation systems.

In our precondition verification framework, the following operators are provided to compare t-expressions: $E_1\, op\, E_2\, iff\, eval(E_1)\, op\, eval(E_2)$ for all the c-traces (of interest) of the flow chart, where *eval* stands for the result of evaluating the expression and *op* is any of the following relational operators: $<, >, =, \leq, \geq, and\ \neq$.

For example, consider the t-expressions at the control points t and r in the flow chart shown in Figure 2:

$$E_t = \sum_{l2-1}(0+1+1)+0$$
$$E_r = \sum_{l2-1}(0+1+1)+0+1+1$$

For any trace where E_t and E_r are defined it can be shown that $E_t < E_r$. Similarly, for control points t and u it can be shown that $E_t \neq E_u$ for any trace where both expressions are defined. Hence, for example, certain types of resource folding transformations can be applied to assignment statements $a5$ and $a6$ (say, using the same ALU for the computations in the assignment statements).

In addition to the techniques discussed in the previous sections, we use several rules, called *comparison rules* to compare t-expressions when the entire trace information is unavailable. Some of the comparison rules often used in our system are given below:

1. $E < k + E$ where k is a positive integer and E is a t-expression.
2. $\sum_{l1} i \neq \sum_{l2}(i+j)$ where i, j are positive integers and *l1* and *l2* are l-variables.
3. $(c\Delta m)i_1\nabla(c\Delta n)i_2) \neq i_3$, where i_1, i_2 and i_3 are integers and $i_3 \neq i_1$ and $i_3 \neq i_2$

The next section has an example to show the application of these rules.

7 Examples and Implementation

This section has two more examples to illustrate the applications of the techniques discussed in the previous sections. Figure 7 shows a flow chart in which we are interested in finding the time step at which control point x is reached. The t-expression at x is,

$E_x = \sum_l((((c2\Delta2)((c1\Delta1)1\nabla(c1\Delta2)1) + 1)\nabla(c2\Delta3)((c1\Delta1)1\nabla(c2\Delta2)1))\|(((c1\Delta1)1\nabla(c2\Delta2)1) + 1 + 1))$

Using the second symbolic equivalence rule given in Section 4 this expression can be reduced to,

$E_x = \sum_l((((c2\Delta2)(1) + 1)\nabla(c2\Delta3)(1))\|((1) + 1 + 1))$

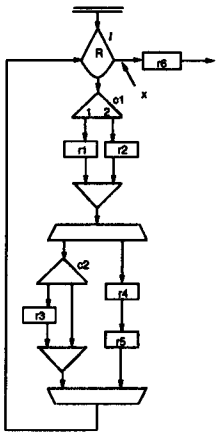

Figure 7: **An Example Flow Chart Program**

By partial evaluation, this is evaluated to,

$E_x = \sum_l((((c2\Delta2)(2)\nabla(c2\Delta3)(1))\|(3))$

Using the third symbolic equivalence rule given in Section 4 this can be reduced to,

$E_x = \sum_l(3)$

This cannot be reduced any further without some information on the trace. Specifically, a partial trace which assigns a value to the l-variable, *l*, is sufficient to evaluate this expression.

As another example, consider the flow chart shown in Figure 8. Suppose we are interested in resource folding involving assignment statements *r11* and *r12*, such as using the same ALU in both statements. The t-expressions at control points y and z are given below:

$E_y = (c3\Delta2)(((c1\Delta1)1\nabla(c1\Delta2)2) + 1)\nabla(c3\Delta3)(((c1\Delta1)1\nabla(c1\Delta2)2) + 1 + 1 + 1)$

$E_z = (c2\Delta1)3\nabla(c2\Delta2)1$

In the absence of any knowledge of the set of possible traces, resource folding is not possible, because we cannot verify that $E_y \neq E_z$ for all possible traces. (In fact, $E_y = E_z$ at the trace where *c1* diverges on *2*, *c3* on *2* and *c2* diverges on *1*). Suppose that the designer can assert that (say, for example, because the same register is tested to take the branching decision involving both *c1* and *c2* *case* statements) whenever *c1* diverges on *1*, *c2* also diverges on *1* and whenever *c1* diverges on *2*, *c2* also diverges on *2*. However, no information is available about *c3*. Even with partial information, the above expressions can be reduced using partial evaluation to (when *c1* diverges on *1*),

$E_y = (c3\Delta2)(2)\nabla(c3\Delta3)(4)$

$E_z = 3$

Using the third comparison rule given in Section 6 one can assert that $E_y \neq E_z$ at this partial trace. Similarly the two expressions can be reduced to, (when *c1* diverges on *2*),

$E_y = (c3\Delta2)(3)\nabla(c3\Delta3)(5)$

$E_z = 1$

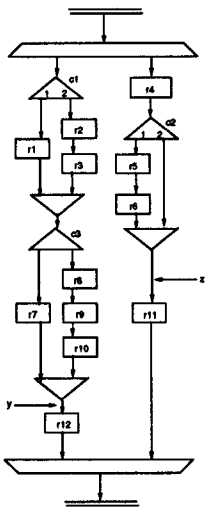

Figure 8: **An Example Flow Chart Program**

Again, using the third comparison rule we can assert that $E_y \neq E_z$ at this partial trace. Therefore, without any further knowledge of the trace information, we can positively verify the resource folding precondition.

We have implemented all the techniques discussed in this paper in a precondition verification system, called PV. PV is implemented in Prolog on a Unix workstation and is part of a transformational design system for register-level hardware designs. PV is described in [6].

8 Discussion

We have successfully used the techniques presented here in a precondition verification system which can verify the preconditions of a set of 18 design transformations used in a register-level design system. In our experience, designers (or programmers) can in most occasions supply partial information about the possible traces. By using the partial evaluation technique in conjunction with symbolic equivalence and comparison rules we could successfully exploit that information leading to better optimization of the design.

Various extensions to the t-expressions are possible: assignments taking multiple time steps, control operators taking one or more time-steps etc. can be easily incorporated.

Temporal logic can also be used to specify temporal preconditions. However, verification of temporal logic expressions is more time consuming. We have abstracted away, by using c- and l-variables, references to the data values which decide the trace of execution. Use of temporal logic would perhaps force one to make references to the data values; we believe that the mechanism based on c- and l-variables and trace diagrams is cleaner and the associated verification techniques are faster as opposed to those using temporal logic based theorem provers.

Finally, having argued the case against the use of general purpose theorem proving for precondition verification, we *are* currently exploring ways of using, in a limited way, such theorem proving techniques, to enhance the power of the methods discussed in this paper. It seems that after applying several rounds of partial evaluation and symbolic equivalence, even if certain precondition could not be verified, it can almost certainly be reduced to a form where theorem proving can be applied in a computationally efficient manner; On the other hand, theorem proving, specifically those based on temporal logic, would have been very expensive (and, preconditions are to be verified on the fly during design transformation) if it were to be used on the unreduced preconditions.

References

[1] D. Bjorner, A. P. Ershov, and N. D. Jones (eds.), "Partial Evaluation and Mixed Computation", North-Holland, 1988.

[2] R. S. Boyer and J. S. Moore, "The Correctness Problem in Computer Science", Academic Press, 1981.

[3] Steven Johnson, "Digital Design from Recursion Equations" MIT Press, 1982.

[4] L. Meertens (ed.), "Program Specification and Transformations, North-Holland, 1987.

[5] W. Rosentiel, "Optimizations in High-level Synthesis", Microprocessing and Microprogramming, pp. 347-352, 1986.

[6] A. Sridhar and R. Vemuri, "Automatic Precondition Verification for High-Level Design Transformations", Proc. International Symposium on Circuits and Systems, 1990.

[7] R. R. Vemuri, "A Transformational Approach to Register Transfer Level Design-Space Exploration", Ph.D. Thesis, Case Western Reserve University, January 1989.

[8] R. Vemuri and C.A. Papachristou, "On the Control-Step Assignment in A Transformational Synthesis System" in G. Saucier and P. McLellan (ed.), Logic and Architecture Synthesis for Silicon Compilers, Elsevier, 1988.

[9] R. A. Walker and D. E. Thomas, "Design Transformation for Algorithmic Level IC Design", IEEE Trans. on Computer-Aided Design, pp. 1115-1128, October 1989.