Verification of Code Motion Techniques Using Value Propagation

Kunal Banerjee, Chandan Karfa, Dipankar Sarkar, and Chittaranjan Mandal

***Abstract*—An equivalence checking method of finite state machines with datapath based on value propagation over model paths is presented here for validation of code motion transformations commonly applied during the scheduling phase of high-level synthesis. Unlike many other reported techniques, the method is able to handle code motions across loop bodies. It consists in propagating the variable values over a path to the subsequent paths on discovery of mismatch in the values for some live variable, until the values match or the final path segments are accounted for without finding a match. Checking loop invariance of the values being propagated beyond the loops has been identified to play an important role. Along with uniform and nonuniform code motions, the method is capable of handling control structure modifications as well. The complexity analysis depicts identical worst case performance as that of a related earlier method of path extension which fails to handle code motion across loops. The method has been implemented and satisfactorily tested on the outputs of a basic block-based scheduler, a path-based scheduler, and the high-level synthesis tool SPARK for some benchmark examples.**

***Index Terms*—Code motion validation, equivalence checking, finite state machines with datapath, value propagation.**

I. Introduction

CODE motion-based transformations, whereby operations are moved across BB boundaries, are used widely in the high-level synthesis tools to improve synthesis results [1]–[3]. The objectives of code motions are (i) reduction of number of computations performed at run-time and (ii) minimization of lifetimes of the temporary variables to avoid unnecessary register usage. Based on the above objectives, code motions can be classified into three categories, namely, busy, lazy, and sparse code motions [4], [5]. Busy code motions (BCMs) advance the code segments as early as possible. Lazy code motions (LCMs) place the code segments as late as possible to facilitate code optimization. LCM is an advanced optimization technique to remove redundant computations. It also involves common subexpression elimination and loop-invariant code motions. In addition, it can also remove partially redundant computations (i.e., computations that occur multiple times on some execution paths, but not even once on other alternate paths). BCMs may reduce the number of computations performed but it increases the life time of temporary variables. LCMs optimally cover both the goals. However, code size is not taken into account in either of these two approaches as both these methods lead to code replication. Sparse code motions additionally try to avoid code replication.

The bisimulation-based method presented in [6] and [7] has been applied successfully to verify structure preserving code motions. However, this method cannot be applied when the control structure gets modified by path-based schedulers [8], [9]. To alleviate this limitation, path-based equivalence checkers for programs represented using the finite state machines with datapath (FSMD) model were proposed [7], [10]–[12]. A path, however, cannot be extended across a loop by definition of path cover [13], [14]. Therefore, all these methods fail in the case of code motions across loops, whereupon some code segment before a loop body is placed after the loop body, or vice-versa. The translation validation for LCMs proposed in [15] is capable of validating code motions across loops but it requires additional information from the synthesis tool which is difficult to obtain in general.

The objective of this paper is to develop a unified verification approach for code motion techniques, including code motions across loop, and control structure modifications, without requiring any information from the transformation engine. This combination of features had not been previously achieved by any single verification technique. A preliminary version of this paper appears in [16] which has since been modified considerably to handle speculative code motions and dynamic loop scheduling (DLS) [9]. In addition to uniform and nonuniform code motion techniques, this paper aims at verifying code motions across loops by propagating the variable values through all the subsequent path segments if mismatch in the values of some live variables is detected. Repeated propagation of values is possible until an equivalent path or a final path segment ending in the reset state is reached. In the latter case, any prevailing discrepancy in values indicates that the original and the transformed behaviors are not equivalent; otherwise they are. The variables whose values are propagated beyond a loop must be invariant to that loop for valid code motions across loops. The loop invariance of such values can be ascertained by comparing the propagated values that are obtained while entering the loop and after one traversal of the loop.

Specifically, the contributions of this paper are as follows. (i) A new concept of value propagation-based equivalence checking has been presented for verification of code motions across loops without compromising the capability of handling uniform and non-uniform code motions and control

Manuscript received August 21, 2013; revised December 9, 2013; accepted March 11, 2014. Date of current version July 15 2014. The work of K. Banerjee was supported by TCS Research Fellowship. This paper was recommended by Associate Editor S. Seshia.

K. Banerjee, D. Sarkar, and C. Mandal are with the Department of Computer Science and Engineering, Indian Institute of Technology, Kharagpur 721302, India (e-mail: kunalb@cse.iitkgp.ernet.in; ds@cse.iitkgp.ernet.in; chitta@cse.iitkgp.ernet.in).

C. Karfa is with Synopsys India Pvt. Ltd., Bangalore 560016, India (e-mail: ckarfa@yahoo.co.in).

Digital Object Identifier 10.1109/TCAD.2014.2314392

structure modifications achieved in earlier methods [10], [12]. (ii) The computational complexity of the presented method has been analyzed and found to be no worse than that for [12]. (iii) Experimental results for several non-trivial benchmarks have been presented to empirically establish the effectiveness of the presented method. (iv) Correctness of value propagation as a method of equivalence checking, and correctness and complexity of the equivalence checking procedure are treated formally.

The rest of the paper is organized as follows. Section II contains a summary of the related work. In Section III, the FSMD model and related concepts are described. Section IV illustrates the basic concepts of the method of value propagation with the help of an example. Some intricacies of the method–detection of loop invariance of subexpressions and subsumption of conditions of execution of the paths–are also highlighted in this section. The correctness of value propagation as a method of equivalence checking is given in Appendix A. The overall verification process is presented in Section V along with some illustrative examples. Its correctness and complexity are formally treated in Appendix B. Experimental results are provided in Section VI. The paper is concluded in Section VII.

II. Related Work

Some recent works [7], [10]–[12], [17], [18] target verification of code motion techniques. Specifically, an FSMD-based equivalence checking has been reported in [17] that can handle code transformations confined within BB boundaries. An improvement based on path recomposition is suggested in [18] to verify speculative code motions where the correctness conditions are formulated in higher-order logic and verified using the Prototype Verification System (PVS) theorem prover. In [6] and [7], a translation validation approach is proposed for high-level synthesis. Their method establishes a bisimulation relation between the set of points in the initial behavior and those in the scheduled behavior under the assumption that the control structure of the behavior does not change during the synthesis process. Recently, the bisimulation-based method presented in [6] is further enhanced in [19] by adding the capability of handling path-based scheduling given in [8]. However, those test cases that involve extensive modification of the control structure of the input programs on application of DLS [9] cannot be handled by [19]. An equivalence checking method for scheduling verification was proposed in [10]. This method is applicable even when the control structure of the input behavior has been modified by the scheduler [8], [9]. It has been shown that this method can verify many code motion techniques. The work reported in [11] has identified some false-negative cases of the algorithm in [10] and proposed an algorithm to overcome those limitations. The method of [12] develops over that of [10] to additionally handle non-uniform code motion techniques. The methods proposed in [10]–[12], [18] are mostly path cover-based approaches where each behavior is decomposed into a finite set of finite paths and equivalence of behaviors are established by showing path level equivalence between two behaviors captured as FSMDs. Changes in structures resulting from optimizing transformations are accounted for by extending path segments in a particular behavior. However, as mentioned in the Introduction, a path cannot be extended across a loop by definition of path cover and hence, all these methods fail in the case of code motions across loops. The translation validation for LCMs proposed in [15] is capable of validating code motions across loops including the cases of infinite loops and also for computations which may fail due to certain exceptions. This method, however, assumes that there exists an injective function from the nodes of the original code to the nodes of the transformed code. The function can be easily obtained and exploited by their methodology because they can precisely detect the code snippet where LCM has been applied, if at all, since the validator is a built-in component within the CompCert compiler [20]. However, such a mapping may not hold for practical synthesis tools like SPARK [21]; even if it holds, it is hard to obtain such a mapping from the synthesis tool. It would be desirable to have a method which can verify code motions across loops without taking any information from the synthesis tools. Although code optimized using verified compilers render verification at a later stage (such as the one presented in this paper) unnecessary, such compilers rarely exist and often deal with input languages with considerable restrictions. As a result, these compilers are not yet popular with the electronic design automation industry, thus necessitating behavioral verification as a post-synthesis process. Moreover, often a *verified* compiler is built out of unverified code transformers; in such cases, it is augmented with a proof-assistant to ensure the correctness of the applied transformations in terms of equivalence between the input program and the program obtained by these code transformers, such as [15]; the methodology presented in the current paper can be used to build such proof-assistants as well.

III. FSMD Model and Related Concepts

A brief formal description of the FSMD model is given in this section; a detailed description of FSMD models can be found in [10]. The FSMD model [22], used in this paper to model the initial behavior and the transformed behavior, is formally defined as an ordered tuple $\langle Q, q_0, I, V, O, f{:}Q \times 2^S \rightarrow Q, h{:}Q \times 2^S \rightarrow U\rangle$, where Q is the finite set of control states, q_0 is the reset (initial) state, I is the set of input variables which are never changed in course of a computation of the model, V is the set of storage variables, O is the set of output variables, f is the state transition function, h is the update function of the output and the storage variables, U represents a set of storage and output assignments and S represents a set of relations over arithmetic expressions and Boolean literals.

A *path* α in an FSMD model is a finite sequence of states where at most the first and the last states may be non-distinct and any two consecutive states in the sequence are in f. The initial (start) and the final states of a path α are denoted as α^s and α^f, respectively. The *condition of execution* R_α *of the path* α is a logical expression over the variables in V and the inputs I such that R_α is satisfied by the (initial) data state of the path iff the path α is traversed. The *data transformation* r_α *of a path* α over V is the tuple $\langle s_\alpha, \theta_\alpha\rangle$; the first member s_α is an ordered tuple $\langle e_i\rangle$ of algebraic expressions over the variables in V and the inputs in I such that the expression e_i represents the value of the variable v_i after the execution of the path in terms of the initial data state of the path; the second member θ_α, which represents the output list along the path

α, is typically of the form $[OUT(P_{i_1}, e_1), OUT(P_{i_2}, e_2), \ldots]$. More specifically, for every expression e output to port P along the path α, there is a member $OUT(P, e)$ in the list appearing in the order in which the outputs occur in α. The condition of execution and the data transformation of a path are computed using the method of symbolic execution.

A *computation* of an FSMD is a finite walk from the reset state back to itself without having any intermediary occurrence of the reset state. Two computations μ_1 and μ_2 of an FSMD are said to be equivalent if $R_{\mu_1} \equiv R_{\mu_2}$, $r_{\mu_1} = r_{\mu_2}$, where R_{μ_1} and R_{μ_2} represent the conditions of execution (over the variables in I) of μ_1 and μ_2, respectively and r_{μ_1} and r_{μ_2} are the data transformations (of the program variables V in terms of the input variables I at the end of the computations) of μ_1 and μ_2, respectively. Computational equivalence of two paths p_1 and p_2, denoted as $p_1 \simeq p_2$, can be defined in a similar manner in terms of $R_{p_1}, R_{p_2}, r_{p_1}$ and r_{p_2}. To determine the equivalence of arithmetic expressions under associative, commutative, distributive transformations, expression simplification, constant folding, etc., we rely on the normalization technique presented in [23] which supports integers only and assumes that no overflow or underflow occurs. The method in [15], in contrast, can handle floating point expressions as well since it treats LCM to be a purely syntactical redundancy elimination transformation.

It is worth noting that if two behaviors are to be the same, then their outputs must match. So, when some variable is output, its counterpart in the other FSMD must attain the same value. In other words, equivalence of θ_α hinges upon the equivalence of s_α. Hence, the rest of the paper focuses on computation of s_α; the computation of θ_α has deliberately been omitted for the sake of brevity.

Let the input behavior be represented by the FSMD $M_0 = \langle Q_0, q_{0,0}, I, V_0, O, f_0, h_0\rangle$ and the scheduled behavior be represented by the FSMD $M_1 = \langle Q_1, q_{1,0}, I, V_1, O, f_1, h_1\rangle$. An FSMD M_0 is said to be contained in an FSMD M_1, symbolically $M_0 \sqsubseteq M_1$, if for any computation μ_0 of M_0, there exists a computation μ_1 of M_1 such that $\mu_0 \simeq \mu_1$. Two FSMDs M_0 and M_1 are said to be computationally equivalent, if $M_0 \sqsubseteq M_1$ and $M_1 \sqsubseteq M_0$.

An FSMD may consist of an infinite number of computations. However, any computation μ of an FSMD M can be looked upon as a computation along some concatenated path $[\alpha_1\alpha_2\alpha_3...\alpha_k]$ of M such that, for $1 \leq i < k$, α_i terminates in the initial state of the path α_{i+1}, the path α_1 emanates from the reset state q_0 and the path α_k terminates in q_0 of M; α_i's may not all be distinct. Hence, we have the following definition.

Definition 1 (Path cover of an FSMD): A finite set of paths $P = \{p_0, p_1, p_2, \ldots, p_k\}$ is said to be a path cover of an FSMD M if any computation μ of M can be looked upon as a concatenation of paths from P.

In order to obtain a path cover for an FSMD each loop is to be cut in at least one cut-point. The set of all paths from a cut-point to another cut-point without having any intermediary cut-point is a path cover of the FSMD [13]. In this paper, a path cover is obtained by setting the reset state and the branching states (i.e., states with more than one outward transition) of the FSMD as cut-points. In addition to the notion of path covers, we need a notion of correspondence between states from two FSMDs as defined below.

Definition 2 (Corresponding States): Let $M_0 = \langle Q_0, q_{0,0}, I, V_0, O, f_0, h_0\rangle$ and $M_1 = \langle Q_1, q_{1,0}, I, V_1, O, f_1, h_1\rangle$ be two FSMDs having identical input and output sets, I and O, respectively, and $q_{0,i}, q_{0,k} \in Q_0$ and $q_{1,j}, q_{1,l} \in Q_1$.

1) The respective reset states $q_{0,0}$ and $q_{1,0}$ are corresponding states.
2) If $q_{0,i} \in Q_0$ and $q_{1,j} \in Q_1$ are corresponding states and there exist $q_{0,k} \in Q_0$ and $q_{1,l} \in Q_1$ such that, for some path β from $q_{0,i}$ to $q_{0,k}$ in M_0, there exists a path α from $q_{1,j}$ to $q_{1,l}$ in M_1 such that $\beta \simeq \alpha$, then $q_{0,k}$ and $q_{1,l}$ are corresponding states.

Finally, the following theorem can be concluded from the above discussion; a detailed proof is available in [10].

Theorem 1: An FSMD M_0 is contained in another FSMD M_1 ($M_0 \sqsubseteq M_1$), if there exists a finite path cover $P_0 = \{p_{0,0}, p_{0,1}, \ldots, p_{0,l}\}$ of M_0 for which there exists a set $P_1 = \{p_{1,0}, p_{1,1}, \ldots, p_{1,l}\}$ of paths of M_1 such that the initial states of $p_{0,i}$ and $p_{1,i}$ are corresponding states, so are their final states and $p_{0,i} \simeq p_{1,i}$, $0 \leq i \leq l$.

It is important to note that the choice of cut-points is nonunique and it is not guaranteed that a path cover of one FSMD obtained from any choice of cut-points in itself will have the corresponding set of equivalent paths for the other FSMD. The path-based approaches [10]–[12] do modify the initial set of path covers to new sets so that the resulting path covers satisfy the property characterized in theorem 1; they, however, cannot validate code motions across loops. In the subsequent section, we devise a method whereby an established path-based approach is molded suitably to verify such transformations.

IV. Method of Value Propagation

A. Basic Concepts

The value propagation method consists in propagating values of variables over the corresponding paths of the two FSMDs on discovery of mismatch in the values of some live variable; the mismatched values are marked to distinguish them from the matched values. A variable v is said to be *live* at a state s if there is an execution sequence starting at s along which its value may be used before v is redefined [24]. Propagation of values from a path β_1 to the next path β_2 is accomplished by associating a *propagated vector* at the end state of the path β_1 (or equivalently, the start state of the path β_2). A *propagated vector* $\bar{\vartheta}$ through a path β_1 is an ordered pair of the form $\langle \mathcal{C}, \langle e_1, e_2, \ldots, e_k\rangle\rangle$, where $k = |V_0 \cup V_1|$. The first element $\mathcal{C}$ of the pair represents the condition that has to be satisfied at the start state of β_1 to traverse the path and reach its end state with the upgraded propagated vector. The second element, referred to as *value-vector*, comprises of e_i, $1 \leq i \leq k$, which represents the symbolic value attained at the end state of β_1 by the variable $v_i \in V_0 \cup V_1$. To start with, for the reset state, the propagated vector is $\langle \top, \langle v_1, v_2, \ldots, v_k\rangle\rangle$ (also represented as $\bar{\rho}$), where $\top$ stands for *true* and $e_i = v_i, 1 \leq i \leq k$, indicates that the variables are yet to be defined. For brevity, we represent the second component as $\bar{v}$ to mean $\langle v_1, v_2, \ldots, v_k\rangle$ and as $\bar{e}$ to mean $\langle e_1, e_2, \ldots, e_k\rangle$, where e_i's are the symbolic expressions (values) involving variables $v_i, 1 \leq i \leq k$, in general. It is important to note that an uncommon variable, i.e., a variable

that is defined in either of the FSMDs but not both, from the set $V_1 - V_0$ ($V_0 - V_1$) retains its symbolic value in the propagated vectors of M_0 (M_1) throughout a computation of M_0 (M_1). Let $R_\beta(\bar{v})$ and $s_\beta(\bar{v})$ represent, respectively, the condition of execution and the data transformation of a path β when there is no propagated vector at the start state β^s of β. In the presence of a propagated vector, $\bar{\vartheta}_{\beta^s} = \langle c_1, \bar{e}\rangle$ say, at β^s, its condition of execution becomes $c_1(\bar{v}) \wedge R_\beta(\bar{v})\{\bar{e}/\bar{v}\}$ and the data transformation becomes $s_\beta(\bar{v})\{\bar{e}/\bar{v}\}$, where $\{\bar{e}/\bar{v}\}$ is called a substitution; the expression $\tau\{\bar{e}/\bar{v}\}$ represents that each variable $v_j \in \bar{v}$ that occurs in τ is replaced by the corresponding expression $e_j \in \bar{e}$ simultaneously with other variables. The propagated vector $\bar{\vartheta}_\beta$ associated with a path $\beta{:}\beta^s \Longrightarrow \beta^f$ will synonymously be referred to as $\bar{\vartheta}_{\beta^f}$ associated with the final state β^f of β. Also, we use the symbol $\bar{\vartheta}_{i,j}$ to represent a propagated vector corresponding to the state $q_{i,j}$. We follow a similar convention (of suffixing) for the members of the propagated vector, namely the condition and the value-vector. The above discussion provides the foundation for comparing two paths, and hence the following definition is in order.

Definition 3 (Equivalent paths for a pair of propagated vectors): A path $\beta{:}q_{0,s} \Longrightarrow q_{0,f}$ of FSMD M_0 with a propagated vector $\langle \mathcal{C}_\beta, \bar{v}_{0,s}\rangle$ is said to be (unconditionally) equivalent (denoted using $\simeq$) to a path $\alpha{:}q_{1,s} \Longrightarrow q_{1,f}$ of FSMD M_1 with a propagated vector $\langle \mathcal{C}_\alpha, \bar{v}_{1,s}\rangle$ if $\mathcal{C}_\beta \wedge R_\beta(\bar{v}_{0,s}/\bar{v}) \equiv \mathcal{C}_\alpha \wedge R_\alpha(\bar{v}_{1,s}/\bar{v})$ and $s_\beta(\bar{v}_{0,s}/\bar{v}) = s_\alpha(\bar{v}_{1,s}/\bar{v})$. Otherwise, the path β with the above propagated vector is said to be conditionally equivalent (denoted using $\simeq_c$) to the path α with the corresponding propagated vector if $q_{0,f} \neq q_{0,0}$, the reset state of M_0, $q_{1,f} \neq q_{1,0}$, the reset state of M_1 and $\forall\beta'$ emanating from the state $q_{0,f}$ with the propagated vector $\langle \mathcal{C}_\beta \wedge R_\beta(\bar{v}_{0,s}/\bar{v}), s_\beta(\bar{v}_{0,s}/\bar{v})\rangle$, $\exists\alpha'$ from $q_{1,f}$ with the propagated vector $\langle \mathcal{C}_\alpha \wedge R_\alpha(\bar{v}_{1,s}/\bar{v}), s_\alpha(\bar{v}_{1,s}/\bar{v})\rangle$, such that $\beta' \simeq \alpha'$ or $\beta' \simeq_c \alpha'$.

In definition 3, the conditions $q_{0,f} \neq q_{0,0}$ and $q_{1,f} \neq q_{1,0}$ prevent value propagation beyond the reset states. It also implies that if β' terminates in $q_{0,0}$ such that it has an equivalent path α', then $\beta' \simeq \alpha'$ must hold; more specifically, paths terminating in reset states cannot have any conditionally equivalent paths. To distinguish explicitly between *conditionally equivalent* paths and *unconditionally equivalent* paths, the terms *C-equivalent* and *U-equivalent* are henceforth used. The final states of two C-equivalent paths are called *conditionally corresponding (C-corresponding) states*.

Example 1: Fig. 1 illustrates the method of value propagation. Let the variable ordering be $\langle u, v, w, x, y, z\rangle$. The states $q_{0,0}$ and $q_{1,0}$ are the reset states of the FSMDs that are being checked for equivalence. The propagated vector corresponding to each reset state is $\bar{\rho}$. The propagated vectors at the end states $q_{0,1}$ and $q_{1,1}$ of the paths β_1 and α_1 respectively are found to be $\bar{\vartheta}_{0,1} = \langle \top, \langle u, v, w, \mathbf{f_1}(\mathbf{u}, \mathbf{v}), \mathbf{y}, z\rangle\rangle$ and $\bar{\vartheta}_{1,1} = \langle \top, \langle u, v, w, \mathbf{x}, \mathbf{f_2}(\mathbf{u}), z\rangle\rangle$, respectively. The variables u, v, w and z did not have any propagated value at the beginning and have not changed their values along the paths. The values of x and y are given in bold face to denote that they mismatch; when the value of some variables is propagated in spite of a match (this is done to resolve transformations such as copy propagation), they are given in normal face. The mismatched values are demarcated because they require special treatment during analysis of code motion across loop, as explained shortly in Section IV-B. Next we use the characteristics of the paths β_2 and α_2 to obtain the respective conditions of execution and data transformations for these propagated vectors. Let Π_i^n represent the i-th projection function of arity n; we keep the arity understood when it is clear from the context. For the path β_2, note that the condition of execution R_{β_2} is $p(x)$ and the data transformation s_{β_2} is $\{y \Leftarrow f_2(u), z \Leftarrow f_3(v)\}$. The modified condition of execution, R'_{β_2} is calculated as $\Pi_1(\bar{\vartheta}_{0,1}) \wedge R_{\beta_2}(\bar{v})\{\Pi_2(\bar{\vartheta}_{0,1})/\bar{v}\} \equiv \top \wedge p(x)\{f_1(u, v)/x\} \equiv p(f_1(u, v))$ and the modified data transformation s'_{β_2} as $s_{\beta_2}(\bar{v})\{\Pi_2(\bar{\vartheta}_{0,1})/\bar{v}\} = \{x \Leftarrow f_1(u, v), y \Leftarrow f_2(u), z \Leftarrow f_3(v)\}$. The symbol $s'_{\beta_2}\Big|_x$ represents the value corresponding to the variable x as transformed after application of s_{β_2} on the propagated vector $\Pi_2(\bar{\vartheta}_{0,1})$, i.e., $f_1(u, v)$. The characteristic tuple for path α_2 comprises $R_{\alpha_2}(\bar{v}) \equiv p(f_1(u, v))$ and $s_{\alpha_2}(\bar{v}) = \{z \Leftarrow f_3(v), x \Leftarrow f_1(u, v)\}$. The condition R'_{α_2} of the path α_2 and the data transformation s'_{α_2} of the propagated vector are calculated similarly as $R'_{\alpha_2} \equiv \Pi_1(\bar{\vartheta}_{1,1}) \wedge R_{\alpha_2}(\bar{v})\{\Pi_2(\bar{\vartheta}_{1,1})/\bar{v}\} \equiv \top \wedge p(f_1(u, v)) \equiv p(f_1(u, v))$ and $s'_{\alpha_2} = s_{\alpha_2}(\bar{v})\{\Pi_2(\bar{\vartheta}_{1,1})/\bar{v}\} = \{x \Leftarrow f_1(u, v), y \Leftarrow f_2(u), z \Leftarrow f_3(v)\}$. We find that $R'_{\beta_2} \equiv R'_{\alpha_2}$ and $s'_{\beta_2} = s'_{\alpha_2}$. Since a match in the characteristic tuples of the respective paths implies value propagation is no longer required in this case, there is no need to store the vectors; the paths β_2 and α_2 are declared as U-equivalent.

For the paths β_3 and α_3, having the conditions of execution $R_{\beta_3} = \neg p(x)$ and $R_{\alpha_3} = \neg p(f_1(u, v))$ respectively, the respective conditions R'_{β_3} and R'_{α_3} with respect to $\bar{\vartheta}_{\beta_3^s}$ and $\bar{\vartheta}_{\alpha_3^s}$ are found to be $\neg p(f_1(u, v))$. The data transformations are $s_{\beta_3} = \{y \Leftarrow f_4(u, v), z \Leftarrow f_5(w)\}$ and $s_{\alpha_3} = \{x \Leftarrow f_6(u, v, w), z \Leftarrow f_5(w)\}$. The respective modified data transformations are $s'_{\beta_3} = \{x \Leftarrow f_1(u, v), y \Leftarrow f_4(u, v), z \Leftarrow f_5(w)\}$ and $s'_{\alpha_3} = \{x \Leftarrow f_6(u, v, w), y \Leftarrow f_2(u), z \Leftarrow f_5(w)\}$. There is a mismatch between s'_{β_3} and s'_{α_3} for the values of x and y; hence the propagated vectors that are stored at $q_{0,2}$ and $q_{1,2}$ (via β_3 and α_3) are $\bar{\vartheta}_{0,2} = \langle \neg p(f_1(u, v)), \langle u, v, w, \mathbf{f_1}(\mathbf{u}, \mathbf{v}), \mathbf{f_4}(\mathbf{u}, \mathbf{v}), f_5(w)\rangle\rangle$ and $\bar{\vartheta}_{1,2} = \langle \neg p(f_1(u, v)), \langle u, v, w, \mathbf{f_6}(\mathbf{u}, \mathbf{v}, \mathbf{w}), \mathbf{f_2}(\mathbf{u}), f_5(w)\rangle\rangle$ respectively. In this example, the propagated vectors $\bar{\vartheta}_{0,1}$ at $q_{0,1}$ and $\bar{\vartheta}_{1,1}$ at $q_{1,1}$ got identically transformed over paths β_2 and α_2, respectively. Had they not matched we would have to store the corresponding propagated vectors along these paths also. This indicates that we may have to store more than one propagated vector in any state. This, however, will not be

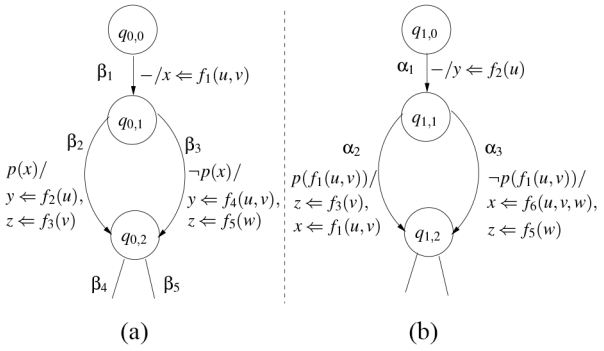

Fig. 1. Example of value propagation. (a) Original FSMD. (b) Transformed FSMD.

Algorithm 1 *valuePropagation* (β, $\langle \mathcal{C}_{\beta^s}, \bar{v}_{\beta^s} \rangle$, α, $\langle \mathcal{C}_{\alpha^s}, \bar{v}_{\alpha^s} \rangle$)

Inputs: Two paths β and α, and two propagated vectors $\langle \mathcal{C}_{\beta^s}, \bar{v}_{\beta^s} \rangle$ at β^s and $\langle \mathcal{C}_{\alpha^s}, \bar{v}_{\alpha^s} \rangle$ at α^s.
Outputs: Propagated vector $\langle \mathcal{C}_{\beta^f}, \bar{v}_{\beta^f} \rangle$ for β^f and the propagated vector $\langle \mathcal{C}_{\alpha f}, \bar{v}_{\alpha f} \rangle$ for α^f.

1: $\mathcal{C}_{\beta^f} \leftarrow \mathcal{C}_{\beta^s} \wedge R_\beta(\bar{v})\{\bar{v}_{\beta^s}/\bar{v}\}$; $\mathcal{C}_{\alpha f} \leftarrow \mathcal{C}_{\alpha^s} \wedge R_\alpha(\bar{v})\{\bar{v}_{\alpha^s}/\bar{v}\}$;
2: **if** $\exists$ variable $v_i \in (V_0 \cup V_1)$ which is *live* at β^f or α^f and $s_\beta(\bar{v})\{\bar{v}_{\beta^s}/\bar{v}\}\Big|_{v_i} \neq s_\alpha(\bar{v})\{\bar{v}_{\alpha^s}/\bar{v}\}\Big|_{v_i}$ **then**
3: $\forall\, v_j \in (V_0 \cup V_1)$, $\Pi_j(\bar{v}_{\beta^f}) \leftarrow s_\beta(\bar{v})\{\bar{v}_{\beta^s}/\bar{v}\}\Big|_{v_j}$;
4: $\forall\, v_j \in (V_0 \cup V_1)$, $\Pi_j(\bar{v}_{\alpha^f}) \leftarrow s_\alpha(\bar{v})\{\bar{v}_{\alpha^s}/\bar{v}\}\Big|_{v_j}$;
5: Mark each variable, x say, which exhibits mismatch at β^f and α^f, and also mark all those variables on which x depends;
6: **end if**
7: **return** $\langle \langle \mathcal{C}_{\beta^f}, \bar{v}_{\beta^f} \rangle, \langle \mathcal{C}_{\alpha f}, \bar{v}_{\alpha f} \rangle \rangle$;

the case. Owing to the depth-first traversal approach achieved through recursive invocations, it is sufficient to work with a single propagated vector at a time for each state. Suppose we first reach the state $q_{0,2}$ via the path β_2 with the mismatched propagated vector $\bar{\vartheta}_{\beta_2^f}$. Only after the U-equivalent or C-equivalent of each of the paths β_4 and β_5 emanating from $q_{0,2}$ using $\bar{\vartheta}_{\beta_2^f}$ are found, does the procedure again visit the state $q_{0,2}$ corresponding to β_3. Finally, when β_3 is accounted for, we store simply $\bar{\vartheta}_{\beta_3^f}$ in $q_{0,2}$.

It has been stated above that only the values of those live variables which exhibit mismatch are marked in the propagated vectors; however, those variables on which these mismatched variables depend need to be marked as well to detect valid code motion across loop as borne out by the example in Section IV-B. The function *valuePropagation* (Algorithm 1) formalizes the above steps of computation of the propagated vectors in case of a mismatch.

Using value propagation as a pivotal concept, we need to formalize our main task of equivalence checking of two given FSMDs. However, certain intricacies are interlaced with value propagation which need to be resolved first. The following subsections illustrate these intricacies and furnishes the mechanisms for addressing them. The overall verification method is then presented in a subsequent section.

B. Need for Detecting Loop Invariance of Subexpressions

In order to verify behaviors in the presence of code motions beyond loops, one needs to ascertain whether certain subexpressions remain invariants in a loop or not. Basically, suppose some variable, w say, gets defined before a loop L in the original behavior, and *somehow* it can be confirmed that the value of w remains invariant in that loop. Now, if an equivalent definition for w is found in the other behavior after exiting the loop L' (which corresponds to L), then such behaviors will indeed be equivalent. In this paper, we target to establish the loop invariance of propagated vectors which, in turn, suffices to prove the loop invariance of the involved variables. The following example is used to reveal these facts more vividly.

Example 2: In Fig. 2, the operation $y \Leftarrow t_1 - t_2$, which is originally placed before the loop body in M_0 [Fig. 2(a)], is moved after the loop in the transformed FSMD M_1 [Fig. 2(b)].

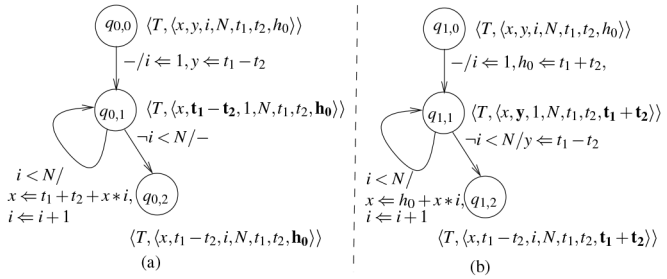

Fig. 2. Example of propagation of values across loop. (a) M_0. (b) M_1.

This transformation reduces the register lifetime for y. In addition, the subexpression $t_1 + t_2$, which is originally computed in every iteration of the loop, is now computed only once before the start of the loop and the result is stored in an uncommon variable h_0 which, in turn, is used in every loop iteration. These two transformations are possible because the variables t_1, t_2 are not updated and y is neither used NOR updated within the loop body.

Now consider the equivalence checking between these two behaviors. The path $q_{0,0} \Longrightarrow q_{0,1}$ of M_0 is said to be candidate C-equivalent to the path $q_{1,0} \Longrightarrow q_{1,1}$ of M_1 since the values of y and h_0 mismatch. Accordingly, we have the propagated value $t_1 - t_2$ for y at $q_{0,1}$ and $t_1 + t_2$ for h_0 at $q_{1,1}$. Now, the path $q_{0,1} \Longrightarrow q_{0,1}$ (which represents a loop) is declared to be candidate C-equivalent of $q_{1,1} \Longrightarrow q_{1,1}$ with the propagated values of h_0 and y. The variables y and h_0 are not updated in either of the loop bodies; NOR are the variables t_1, t_2 participating in the expression values $t_1 - t_2$ and $t_1 + t_2$ of y and h_0, respectively. Hence, we have the same propagated values at $q_{0,1}$ and $q_{1,1}$ after executing the loops indicating that code motion across loop may have taken place. Since h_0 is an uncommon variable (in $V_1 - V_0$), its value can never match over the two FSMDs. Therefore, if a definition of the common variable y is detected after the loop in M_1 which renders the values of y in the two FSMDs equivalent, then such code motions across the loop would indeed be valid. In the current example, this occurs in the path $q_{1,1} \Longrightarrow q_{1,2}$ and hence finally, $q_{0,1} \Longrightarrow q_{0,2}$ and $q_{1,1} \Longrightarrow q_{1,2}$ are designated as U-equivalent paths with matches in all the variables except in the uncommon variable, and the previously declared candidate C-equivalent path pairs are asserted to be actually C-equivalent.

The above example highlights the intricacy involved in verification of code motions across loops, that is, detection of loop invariant subexpressions. Example 2 exhibits only simple loops. However, a loop may comprise of several paths because of branchings contained in a loop. The conventional approach of using back edges[1] to detect loops has been used in [16]. This approach however, fails in the presence of DLS [9]. In Section V-B, an example of DLS is presented and its equivalence is established by our verification procedure with the help of a different loop detection mechanism as described below.

To identify loops, a *LIST* of (candidate) C-equivalent pairs of paths is maintained. Initially, the *LIST* is empty. Whenever

[1]A back edge (u, v) connects u to an ancestor v in a depth-first tree of a directed graph [25].

a pair of (candidate) C-equivalent paths is detected, they are added to the *LIST*. Thus, if a path occurs in the *LIST*, then it indicates that the final state of the path has an associated propagated vector with some unresolved mismatch(es). If at any point of value propagation a path with the start state q is encountered which is also the final state of some path appearing in *LIST*, it means that a loop has been encountered with the mismatch that was identified during the last visit of q (stored as a propagated vector) not yet resolved.[2] Once pairs of U-equivalent paths are subsequently detected for every path emanating from the states β^f and α^f of a candidate C-equivalent path pair $\langle \beta, \alpha \rangle$ present in the *LIST*, the paths β and α are declared as *actually* C-equivalent (by definition 3).

If a loop is crossed over with some mismatches persisting, it means that the last constituent path of the loop is not yet U-equivalent. Let q be both the entry and exit state of such a loop; the propagated vector already stored at q (during entry) and the one computed after traversal of the final path of the loop in M leading to q need to be compared. This is explained with the help of example 2. Let us first consider a variable, such as y in the example, whose values mismatched while entering the loop. Now, if this variable had got updated within the loop, then surely it was not an invariant in the loop, a fact which would have easily been detected by the presence of a mismatch for y between the initial and the final vectors of $q_{0,1} \Longrightarrow q_{0,1}$ or $q_{1,1} \Longrightarrow q_{1,1}$. Secondly, consider those variables on which the mismatched variables depend such as, t_1 and t_2. If any of them were to get modified within the loop (even if identically for both the FSMDs), then moving the operation $y \Leftarrow t_1 - t_2$ from the path $q_{0,0} \Longrightarrow q_{0,1}$ in M_0 to the path $q_{1,1} \Longrightarrow q_{1,2}$ in M_1 would not have been a valid case of code motion across loop. Such modifications (in t_1 or t_2) also get detected by comparing the vectors since we mark the value of those variables as well on which some mismatched variable depends (as was mentioned in Section IV) in the propagated vector. Lastly, all other variables should get defined identically in the two loops for their data transformations to match. If any difference in the value of some marked variable between entry and exit to a loop is detected, then we can ascertain that this variable is not an invariant in the loop. It further implies that the code motion across the loop need not be valid. Since it cannot be determined statically how many times a loop will iterate before exiting, it is not possible to determine what values will the unmarked variables take when a loop iterates. Moreover, the unmarked variables also do not participate in code motion across loop, hence they are reverted to their symbolic values at the exit of a loop. For example, the variable i in both the FSMDs of Fig. 2 has the respective values 1 and 2 during the entry and the exit of the loop. However, before the paths $q_{0,1} \Longrightarrow q_{0,2}$ and $q_{1,1} \Longrightarrow q_{1,2}$ are compared, the value of i is set to its symbolic value "i" (instead of 2) in the propagated vectors computed at $q_{0,1}$ and $q_{1,1}$ after the respective loops are exited. The function *loopInvariant* (Algorithm 2) accomplishes these tasks.

[2]While treating the example in Section V-B, we underline a situation where the loop detection mechanism based on back edge fails and the above method succeeds.

Algorithm 2 *loopInvariant* (γ, $\bar{\vartheta}'_{\gamma^f}$, $\bar{\vartheta}_{\gamma^f}$, $\bar{\vartheta}'_{\delta^f}$)

Inputs: A path γ, a propagated vector $\bar{\vartheta}'_{\gamma^f}$ which is computed after γ, the propagated vector $\bar{\vartheta}_{\gamma^f}$ stored in the cut-point γ^f, which is the entry/exit state of a loop, and the propagated vector $\bar{\vartheta}'_{\delta^f}$ where δ is the C-corresponding path of γ.
Outputs: A Boolean value.

1: **if** $\exists$ a marked variable x, $\bar{\vartheta}'_{\gamma^f}\Big|_x \neq \bar{\vartheta}_{\gamma^f}\Big|_x$ **then**
2: **return** *false*;
3: **else if** $\exists$ an unmarked variable x, $\bar{\vartheta}'_{\gamma^f}\Big|_x \neq \bar{\vartheta}'_{\delta^f}\Big|_x$ **then**
4: **return** *false*;
5: **end if**
6: Set each unmarked variable x to its symbolic value "x";
7: **return** *true*;

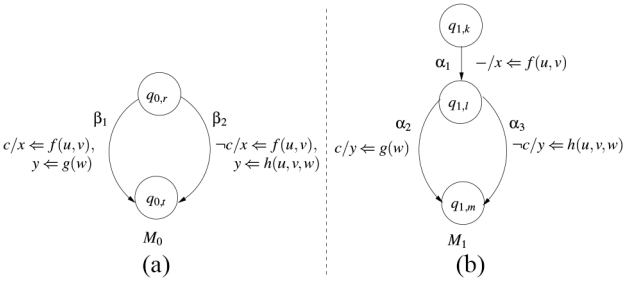

Fig. 3. Example to illustrate the cases $R_\beta \Rightarrow R_\alpha$ and $R_\alpha \Rightarrow R_\beta$. (a) Original FSMD. (b) Transformed FSMD.

C. Subsumption of Conditions of Execution of the Paths Being Compared

It is worth noting that the examples demonstrated in Figs. 1 and 2 illustrate the case $R_\beta \equiv R_\alpha$. Another intricacy arises when the condition $R_\beta \not\equiv R_\alpha$ but $R_\beta \Rightarrow R_\alpha$ or $R_\alpha \Rightarrow R_\beta$ is encountered. Specifically, to handle this situation, the notion of *null path* is introduced in an FSMD, to force value propagation along the path in the other FSMD. A null path (of length 0) from any state q to the same state q has the condition of execution $\top$ and a null (identity) data transformation. We refer to a null path emanating from a state q as η_q having the start state η_q^s same as the final state $\eta_q^f = q$. The example given below elucidates on how to handle these cases.

Example 3: In Fig. 3, let the states $q_{0,r}$ and $q_{1,k}$ be the corresponding states and the variable ordering be $\langle u, v, w, x, y \rangle$. When we search for a path starting from $q_{1,k}$ that is C-equivalent to β_1, we find $R_{\beta_1} \Rightarrow R_{\alpha_1}$ (i.e. $c \Rightarrow \top$). The path α_1 is compared with $\eta_{q_{0,r}}$; in the process, the vector propagated to $q_{1,l}$ becomes $\bar{\vartheta}_{1,l} = \langle \top, \langle u, v, w, \mathbf{f}(\mathbf{u}, \mathbf{v}), y \rangle \rangle$ (and $\eta_{q_{0,r}}$ is held C-equivalent to α_1). For the path α_2, $R_{\alpha_2} \equiv c$ and $s_{\alpha_2} = \{y \Leftarrow g(w)\}$. Based on $\bar{\vartheta}_{1,l}$, the path characteristics are then calculated to be $R'_{\alpha_2} \equiv c$ and $s'_{\alpha_2} = \{x \Leftarrow f(u, v), y \Leftarrow g(w)\}$ which are equivalent to that of β_1. Analogously, the path characteristics of β_2 and α_2 (with $\bar{\vartheta}_{1,l}$) are found to be equivalent.

When we have to show the converse, i.e., $M_1 \sqsubseteq M_0$, we are required to find a path in M_0 that is equivalent to α_1. We find two paths β_1 and β_2 such that $R_{\beta_1} \Rightarrow R_{\alpha_1}$ (i.e., $c \Rightarrow \top$) and also $R_{\beta_2} \Rightarrow R_{\alpha_1}$ (i.e., $\neg c \Rightarrow \top$). The vector $\bar{\vartheta}_{1,l}$ (same as above) is propagated to $q_{1,l}$ after comparing α_1 with $\eta_{q_{0,r}}$. Then we recursively try to find the

Algorithm 3 *findEquivalentPath* (β, $\bar{\vartheta}_{\beta^s}$, $q_{1,j}$, $\bar{\vartheta}_{\alpha^s}$, P_0, P_1)

Inputs: A path $\beta \in P_0$, the propagated vector at its start state $\bar{\vartheta}_{\beta^s}$, a state $q_{1,j} \in M_1$ which is the C-corresponding or U-corresponding state of β^s, the propagated vector $\bar{\vartheta}_{\alpha^s}$ associated with $q_{1,j}$, and a path cover P_0 of M_0, a path cover P_1 of M_1,
Outputs: Let $\beta_m = \beta$ or η_{β^s}. An ordered tuple $\langle \beta_m, \alpha, \bar{\vartheta}_{\beta_m^f}, \bar{\vartheta}_{\alpha^f} \rangle$ such that β_m and α are either U- or C-equivalent, and $\bar{\vartheta}_{\beta_m^f}$ and $\bar{\vartheta}_{\alpha^f}$ are the vectors that are to be propagated to β_m^f and α^f respectively.

Let $R'_\beta = R_\beta(\Pi_2(\bar{\vartheta}_{\beta^s})/\bar{\nu})$ and $s'_\beta = s_\beta(\Pi_2(\bar{\vartheta}_{\beta^s})/\bar{\nu})$;
for each path $\alpha \in P_1$ originating from $q_{1,j}$ **do**
  Let $R'_\alpha = R_\alpha(\Pi_2(\bar{\vartheta}_{\alpha^s})/\bar{\nu})$ and $s'_\alpha = s_\alpha(\Pi_2(\bar{\vartheta}_{\alpha^s})/\bar{\nu})$;
  if $R'_\beta \equiv R'_\alpha$ **then**
    if $s'_\beta = s'_\alpha$ **then**
      return $\langle \beta, \alpha, \bar{\rho}, \bar{\rho} \rangle$; **Case** 1.1
    else
      $\langle \bar{\vartheta}_{\beta^f}, \bar{\vartheta}_{\alpha^f} \rangle \leftarrow$ *valuePropagation* (β, $\bar{\vartheta}_{\beta^s}$, α, $\bar{\vartheta}_{\alpha^s}$); **Case** 1.2
      return $\langle \beta, \alpha, \bar{\vartheta}_{\beta^f}, \bar{\vartheta}_{\alpha^f} \rangle$;
    end if
  else if $R'_\beta \Rightarrow R'_\alpha$ **then**
    $\langle \bar{\vartheta}_{\eta_{\beta^s}}, \bar{\vartheta}_{\alpha^f} \rangle \leftarrow$ *valuePropagation* (η_{β^s}, $\bar{\vartheta}_{\beta^s}$, α, $\bar{\vartheta}_{\alpha^s}$); **Case** 2
    return $\langle \eta_{\beta^s}, \alpha, \bar{\vartheta}_{\eta_{\beta^s}}, \bar{\vartheta}_{\alpha^f} \rangle$;
  else if $R'_\alpha \Rightarrow R'_\beta$ **then**
    $\langle \bar{\vartheta}_{\beta^f}, \bar{\vartheta}_{\eta_{q_{1,j}}} \rangle \leftarrow$ *valuePropagation* (β, $\bar{\vartheta}_{\beta^s}$, $\eta_{q_{1,j}}$, $\bar{\vartheta}_{\alpha^s}$); **Case** 3
    return $\langle \beta, \eta_{q_{1,j}}, \bar{\vartheta}_{\beta^f}, \bar{\vartheta}_{\eta_{q_{1,j}}} \rangle$
  end if
end for
return $\langle \beta, \Omega, \bar{\rho}, \bar{\rho} \rangle$; **Case** 4

paths equivalent to α_2 and α_3, thereby obtaining the set of C-equivalent paths: $\{\langle \alpha_1, \eta_{q_{0,r}} \rangle\}$ and the set of U-equivalent paths: $\{\langle \alpha_2, \beta_1 \rangle, \langle \alpha_3, \beta_2 \rangle\}$.

To summarize, therefore, of the paths β and α, the one having a stronger condition will have a null path introduced at its initial state to force value propagation along the other path. The function *findEquivalentPath* (Algorithm 3) seeks to find a pair of U- or C-equivalent paths depending on how the conditions of execution of the paths being compared are related to each other. The propagated vectors for the end states of these paths also get updated accordingly.

Specifically, the function *findEquivalentPath* takes as inputs a path β of the FSMD M_0, a propagated vector $\bar{\vartheta}_{\beta^s}$, a state $q_{1,j}$ of the FSMD M_1, which has correspondence with the state β^s, a propagated vector $\bar{\vartheta}_{\alpha^s}$ at $q_{1,j}$, and the path covers P_0 and P_1 of the FSMDs M_0 and M_1, respectively. The function's objective is to find a U-equivalent or C-equivalent path for β (with respect to $\bar{\vartheta}_{\beta^s}$) in P_1 emanating from $q_{1,j}$. For the paths in P_1 emanating from $q_{1,j}$, the characteristic tuples are computed with respect to $\bar{\vartheta}_{\alpha^s}$. It returns a 4-tuple $\langle \beta_m, \alpha, \bar{\vartheta}_{\beta_m^f}, \bar{\vartheta}_{\alpha^f} \rangle$, which is defined depending upon the following cases. In all the cases, β_m is a path in P_0, α is a path in P_1, $\bar{\vartheta}_{\beta_m^f}$ is the propagated vector at the end state β_m^f of β_m, and $\bar{\vartheta}_{\alpha^f}$ is the propagated vector at the end state α^f of α. In the algorithm, the symbols R'_β and R'_α denote respectively the condition of execution of the path β with respect to $\bar{\vartheta}_{\beta^s}$ and that of the path α with respect to $\bar{\vartheta}_{\alpha^s}$; likewise for s'_β and s'_α. We have the following cases:

Case 1($R'_\beta \equiv R'_\alpha$): A path α in P_1 is found such that $R'_\alpha \equiv R'_\beta$—under this situation, we have the following two subcases:

Algorithm 4 *containmentChecker* (M_0, M_1)

Inputs: Two FSMDs M_0 and M_1.
Outputs: Whether M_0 is contained in M_1 or not, P_0, P_1: path covers of M_0 and M_1 respectively, ζ: the set of corresponding state pairs, E_u: ordered pairs $\langle \beta, \alpha \rangle$ of paths such that $\beta \in P_0$ and $\alpha \in P_1$, and $\beta \simeq \alpha$, E_c: ordered pairs $\langle \beta, \alpha \rangle$ of paths such that $\beta \in P_0$ and $\alpha \in P_1$, and $\beta \simeq_c \alpha$.

Incorporate cutpoints in M_0 and M_1; let P_0 (P_1) be the set of all paths of M_0 (M_1) from a cutpoint to a cutpoint having no intermediary cutpoint, with each path having its condition of execution R, data transformation s, and output list θ computed;
Let ζ be $\{\langle q_{0,0}, q_{1,0} \rangle\}$, and E_u and E_c be empty;
for each member $\langle q_{0,i}, q_{1,j} \rangle$ of ζ **do**
  if *correspondenceChecker*($q_{0,i}$, $q_{1,j}$, P_0, P_1, ζ, E_u, E_c, $LIST = \Phi$) returns "failure" **then**
    Report "*May be* $M_0 \not\sqsubseteq M_1$" and (exit);
  end if
end for
Report "$M_0 \sqsubseteq M_1$";

Case 1.1($s'_\beta = s'_\alpha$): Return $\langle \beta, \alpha, \bar{\rho}, \bar{\rho} \rangle$.

Case 1.2($s'_\beta \neq s'_\alpha$): Return $\langle \beta, \alpha, \bar{\vartheta}_{\beta^f}, \bar{\vartheta}_{\alpha^f} \rangle$. The propagated vectors are computed by *valuePropagation*(β, $\bar{\vartheta}_{\beta^s}$, α, $\bar{\vartheta}_{\alpha^s}$).

Case 2($R'_\beta \Rightarrow R'_\alpha$): Return $\langle \eta_{\beta^s}, \alpha, \bar{\vartheta}_{\eta_{\beta^s}}, \bar{\vartheta}_{\alpha^f} \rangle$. The null path η_{β^s} originating (and terminating) in the start state of β is returned as β_m along with α and the propagated vectors computed by *valuePropagation* invoked with the same parameters as above. Note that $\bar{\vartheta}_{\eta_{\beta^s}} = \bar{\vartheta}_{\beta^s}$.

Case 3($R'_\alpha \Rightarrow R'_\beta$): Return $\langle \beta, \eta_{q_{1,j}}, \bar{\vartheta}_{\beta^f}, \bar{\vartheta}_{\eta_{q_{1,j}}} \rangle$.

Case 4($R'_\beta \not\equiv R'_\alpha$ *and* $R'_\beta \not\Rightarrow R'_\alpha$ *and* $R'_\alpha \not\Rightarrow R'_\beta$): Return $\langle \beta, \Omega, \bar{\rho}, \bar{\rho} \rangle$. No path in P_1 can be found whose condition of execution is equal to or stronger/weaker than that of β; then *findEquivalentPath* returns a non-existent path Ω in place of α. In this case, the other three values in the four-tuple are not of any significance.

V. Overall Verification Method

To begin with the procedure of equivalence checking, we have to identify the cut-points in an FSMD followed by identification of paths and their corresponding characteristics involving the conditions and the data transformations. We also need to store the pairs of corresponding states in a structure, ζ say, and the U-equivalent and the C-equivalent pairs of paths in E_u and E_c, respectively.

The function *containmentChecker* (Algorithm 4) initializes all the above mentioned data structures, invokes *correspondenceChecker* (Algorithm 5) with the members of the set ζ of the corresponding state pairs one by one to check whether for every path emanating from a state in the pair, there is a U- or C-equivalent path from the other member of the pair; depending on the result returned by *correspondenceChecker*, it outputs the decision whether the original FSMD is contained in the transformed FSMD or not. A list, called *LIST*, is maintained to keep track of the (candidate) C-equivalent pairs of paths visited along the chain of recursive invocations of *correspondenceChecker* invoked by *containmentChecker*. In case the equivalence checking procedure fails for a chain of paths, this *LIST* is output as a possible counterexample. Hence, note that every time *correspondenceChecker* is called from *containmentChecker*, the former's last parameter *LIST* is set to *empty*.

The verification procedure starts with the two reset states declared as corresponding states and also ends with the reset

Algorithm 5 *correspondenceChecker* ($q_{0,i}$, $q_{1,j}$, P_0, P_1, ζ, E_u, E_c, *LIST*)

Inputs: Two states $q_{0,i} \in M_0$ and $q_{1,j} \in M_1$, two path covers P_0 of M_0 and P_1 of M_1, ζ: the set of corresponding state pairs, E_u and E_c for storing the pairs of U-equivalent and C-equivalent paths, respectively, and *LIST*: a list of paths maintained to keep track of C-equivalent paths.

Outputs: Returns "success" if for every path emanating from $q_{0,i}$ there is an equivalent path originating from $q_{1,j}$ in M_1, otherwise returns "failure." Also updates ζ, E_u, E_c and *LIST*.

1: **for** each path $\beta \in P_0$ emanating from $q_{0,i}$ **do**
2: **if** $\Pi_1(\bar{\vartheta}_{0,i}) \wedge R_\beta(\bar{v})\{\Pi_2(\bar{\vartheta}_{0,i})/\bar{v}\} \not\equiv false$ **then**
3: $\langle \beta_m, \alpha, \bar{\vartheta}'_{\beta_m^f}, \bar{\vartheta}'_{\alpha^f} \rangle \leftarrow$ *findEquivalentPath*(β, $\bar{\vartheta}_{0,i}$, $q_{1,j}$, $\bar{\vartheta}_{1,j}$, P_0, P_1);
4: **if** $\alpha = \Omega$ **then**
5: Report "*equivalent path of* β *may not be present in* M_1," display *LIST* and return (failure);
6: **else if** $\bar{\vartheta}_{\beta_m^f} \neq \bar{\vartheta}_{\alpha^f}$ /* satisfied by cases 1.2, 2, 3 of algorithm 3 */ **then**
7: **if** (β_m^f appears as the final state of some path already in *LIST* $\wedge$ *!loopInvariant*(β_m, $\bar{\vartheta}'_{\beta_m^f}$, $\bar{\vartheta}_{\beta_m^f}$, $\bar{\vartheta}'_{\alpha^f}$)) $\vee$ (α^f appears as the final state of some path already in *LIST* $\wedge$ *!loopInvariant*(α, $\bar{\vartheta}'_{\alpha^f}$, $\bar{\vartheta}_{\alpha^f}$, $\bar{\vartheta}'_{\beta_m^f}$)) **then**
8: Report "*propagated values are not loop invariant*," display *LIST* and return (failure);
9: **else if** $\beta_m^f = q_{0,0} \vee \alpha^f = q_{1,0}$ **then**
10: Report "*reset states reached with unresolved mismatch*," display *LIST* and return (failure);
11: **else**
12: $\bar{\vartheta}_{\beta_m^f} \leftarrow \bar{\vartheta}'_{\beta_m^f}$; $\bar{\vartheta}_{\alpha^f} \leftarrow \bar{\vartheta}'_{\alpha^f}$; /* candidate C-equivalence */
13: Append $\langle \beta_m, \alpha \rangle$ to *LIST*;
14: **return** *correspondenceChecker* (β_m^f, α^f, P_0, P_1, ζ, E_u, E_c, *LIST*);
15: **end if** /* *loopInvariant* */
16: **else**
17: $E_u \leftarrow E_u \bigcup \{\langle \beta_m, \alpha \rangle\}$; /* U-equivalence */
18: $\zeta \leftarrow \zeta \bigcup \{\langle \beta_m^f, \alpha_m^f \rangle\}$; /* $\beta_m = \beta$ in this case */
19: **end if** /* $\alpha = \Omega$ */
20: **end if**
21: **end for**
22: $E_c \leftarrow E_c \bigcup$ {last member of *LIST*};
23: *LIST* $\leftarrow$ *LIST*$\setminus$ {last member of *LIST*};
24: return (success);

states as a corresponding state pair in case the two FSMDs are indeed equivalent. Otherwise, it terminates on encountering any of the following "failure" conditions: (i) given a path in one FSMD, it fails to discover its U- or C-equivalent path in the other one: (ii) it discovers that a propagated vector depicting some mismatches at a loop state is not a loop invariant and some of them do not get resolved in the loop. Note that while failure to find a U-equivalent path occurs in one step, failure to find a C-equivalent path may be detected only when the reset state is reached through several steps without finding a match in the variable values. A chain of C-equivalence may be obtained in the form $\beta_1 \simeq_c \alpha_1$ if $\beta_2 \simeq_c$ if $\beta_3 \simeq_c \alpha_3 \ldots$ if $\beta_k \simeq \alpha_k$, where β_k^f and α_k^f are the reset states; when $\beta_k \simeq \alpha_k$ is identified, the state pair $\langle \beta_k, \alpha_k \rangle$ is to be put in E_u and the pairs $\langle \beta_i, \alpha_i \rangle, 1 \leq i \leq k-1$, in E_c (after carrying out some further checks). Whereas, $\beta_k \not\simeq \alpha_k$ would result in a failure in the equivalence checking procedure, and the chain, kept track of by *LIST*, is output as a possible counterexample.

The central function is *correspondenceChecker* given in Algorithm 5. It takes as inputs two states $q_{0,i}$ and $q_{1,j}$ belonging to the FSMDs M_0 and M_1 respectively, and are in correspondence to each other; the path covers P_0 and P_1, the set of corresponding state pairs ζ, the set of U-equivalent path pairs E_u, the set of C-equivalent path pairs E_c, and the *LIST*. It returns "success" if for every path emanating from $q_{0,i}$, an equivalent path originating from $q_{1,j}$ is found; otherwise it returns "failure." The behavior of the function is as follows. Dynamically, for any path β originating from the state $q_{0,i}$ of the original FSMD M_0, it invokes the function *findEquivalentPath* to find a U- or C-equivalent path α originating from $q_{1,j}$ of the transformed FSMD M_1 where $\langle q_{0,i}, q_{1,j} \rangle$ is a corresponding or C-corresponding state pair. Recall that the function *findEquivalentPath* returns a 4-tuple $\langle \beta_m, \alpha, \bar{\vartheta}_{\beta_m^f}, \bar{\vartheta}_{\alpha^f} \rangle$ depending on the following cases: (i) if it fails to find a path α such that $\beta \simeq \alpha$ or $\beta \simeq_c \alpha$, then it returns $\alpha = \Omega$, where Ω represents a non-existent path, causing *correspondenceChecker* to return "failure" as shown in its step 5; (ii) if it finds a path α such that $R_\beta \equiv R_\alpha$ or $R_\alpha \Rightarrow R_\beta$, then β_m is β itself; and (iii) if it finds a path α such that $R_\beta \Rightarrow R_\alpha$ and $R_\beta \not\equiv R_\alpha$, then β_m is returned as a null path from $q_{0,i}$. In the last two cases, the function *correspondenceChecker* next examines whether the propagated vectors $\bar{\vartheta}_{\beta_m^f}$ and $\bar{\vartheta}_{\alpha^f}$ computed after β_m and α are equal or not.

$\bar{\vartheta}_{\beta_m^f} \neq \bar{\vartheta}_{\alpha^f}$: Unequal propagated vectors imply that U-equivalence could not be established, and hence further value propagation is required. However, the following checks are carried out first.

Loop-invariance: Whether a loop has been crossed over is checked in step 7, and if so, a check for loop invariance of the propagated vectors $\bar{\vartheta}_{\beta_m^f}$ and $\bar{\vartheta}_{\alpha^f}$ is carried out with the aid of the function *loopInvariant*. In case a failure in loop invariance is detected for either of the propagated vectors, *correspondenceChecker* returns "failure."

Extendability: Next, checks are made to ensure that neither of the end states β_m^f and α^f is a reset state. Since a computation does not extend beyond the reset state, reaching a reset state with C-equivalence (and not U-equivalence) results in returning failure by the *correspondenceChecker* as shown in step 10.

If $\bar{\vartheta}_{\beta_m^f} \neq \bar{\vartheta}_{\alpha^f}$ and both the checks for loop invariance and end states being a reset state resolve in success, then $\langle \beta_m, \alpha \rangle$ is appended to *LIST*, and *correspondenceChecker* calls itself recursively with the arguments β_m^f and α^f (step 14) to continue searching for equivalent paths.

$\bar{\vartheta}_{\beta_m^f} = \bar{\vartheta}_{\alpha^f}$: Attainment of equal propagated vectors signify discovery of U-equivalent paths. Consequently, the data structures ζ and E_u get updated. Notice that these steps are executed only after the recursive calls to *correspondenceChecker* terminate, i.e., a U-equivalent pair has been found. It is to be noted that a state pair qualifies to be a member of ζ if the propagated vectors computed for these states match totally, i.e., when $\bar{\vartheta}_{\beta_m^f} = \bar{\vartheta}_{\alpha^f}$.

When the control reaches step 22 of *correspondenceChecker*, it implies that for every chain of paths that emanates from the state $q_{0,i}$, there exists a corresponding chain of paths emanating from $q_{1,j}$ such that their final paths are U-equivalent. Note that $q_{0,i}$ and $q_{1,j}$ are the respective final states of the last member of *LIST*. Hence, the last member of *LIST* gets added to the set E_c in accordance with definition 3 and is removed from *LIST*. The remaining (preceding) members are

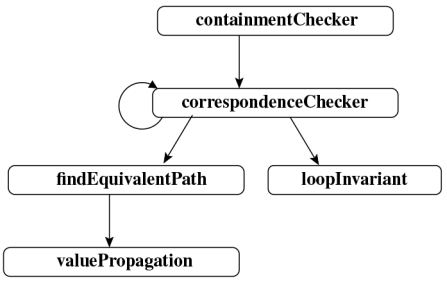

Fig. 4. Call graph of the proposed verification method.

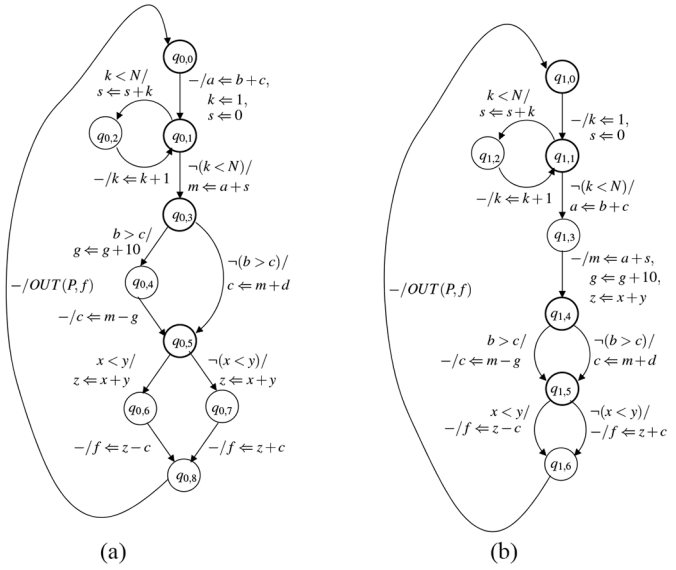

Fig. 5. FSMDs before and after scheduling. (a) M_0. (b) M_1.

yet to be declared C-equivalent because all the paths emanating from the final state of the last path of the updated *LIST* have not yet been examined. The members of *LIST* are also displayed as a part of the report generated whenever one of the "failure" conditions mentioned above is encountered to aid in the process of debugging.

A call graph of the proposed verification method is given in Fig. 4. The correctness and the complexity of the method have been treated in Appendix B.

A. Illustrative Example

In this section, the working of our verification procedure is explained with the example given in Fig. 5. Fig. 5(a) and (b) show two FSMDs before and after scheduling. The definition of a in Fig. 5(a) occurs before the loop ($q_{0,1} \rightarrow q_{0,2} \rightarrow q_{0,1}$), whereas in Fig. 5(b), it has been moved after the loop ($q_{1,1} \rightarrow q_{1,2} \rightarrow q_{1,1}$)–an instance of code motion across loop. Such code motions reduce register life times (as in this case) and may also minimize the overall register usage. Moreover, the definition of z has been moved from both the (*true* and *false*) branches to the predecessor block which is an instance of boosting up–a uniform code motion technique, and the definition of g has been moved from one of the branches to the predecessor block which is an instance of speculation–a non-uniform code motion technique.

Example 4: The example given in Fig. 5(a) represents the original behavior M_0 and Fig. 5(b) represents its transformed behavior M_1. The following steps summarize the progress of the verification method. To begin with, $q_{0,0}$ and $q_{1,0}$ are considered to be corresponding states.

1) The path characteristics of $q_{0,0} \Longrightarrow q_{0,1}$ and $q_{1,0} \Longrightarrow q_{1,1}$ are found to match in all aspects other than the definition of a. Hence, they are declared to be candidate C-equivalent and the path pair is stored in the *LIST* (step 13 of *correspondenceChecker*).
2) Next, the loops $q_{0,1} \Longrightarrow q_{0,1}$ and $q_{1,1} \Longrightarrow q_{1,1}$ are compared. The variables s and k are found to be updated identically in both the loops while the values for a, viz., $b + c$ in M_0 and the symbolic value "a" in M_1, along with those of b and c remain unmodified in the loops; hence, it is concluded that code motion across loop may have taken place and the paths representing the loops are appended to the *LIST*.
3) The paths $q_{0,1} \Longrightarrow q_{0,3}$ and $q_{1,1} \Longrightarrow q_{1,4}$ are analyzed next. The definition of a in the latter path is found to be equivalent to that in M_0. The values of g and z, however, mismatch in the paths being compared and consequently, the paths are also put in the *LIST*.
4) When the paths $q_{0,3} \overset{b>c}{\Longrightarrow} q_{0,5}$ and $q_{1,4} \overset{b>c}{\Longrightarrow} q_{1,5}$ are compared, the values of the variables g and c match but the mismatch for z still persists; therefore, these paths are put in the *LIST* as well.
5) Finally, on comparing the paths $q_{0,5} \overset{x<y}{\Longrightarrow} q_{0,0}$ and $q_{1,5} \overset{x<y}{\Longrightarrow} q_{1,0}$, all values are found to match. Consequently, these pair of paths are declared to be U-equivalent. Note that as per definition 3, a pair of candidate C-equivalent pair of paths can be declared to be actually C-equivalent when all the paths emanating from that pair are found to be U-equivalent or C-equivalent. Hence, the paths $q_{0,3} \overset{b>c}{\Longrightarrow} q_{0,5}$ and $q_{1,4} \overset{b>c}{\Longrightarrow} q_{1,5}$ cannot be declared as C-equivalent yet.
6) Owing to the depth-first traversal of the FSMDs (achieved through recursion in step 14 of *correspondenceChecker*), the paths $q_{0,5} \overset{\neg(x<y)}{\Longrightarrow} q_{0,0}$ and $q_{1,5} \overset{\neg(x<y)}{\Longrightarrow} q_{1,0}$ are examined next. Note that the *LIST* available in this step is the same as that of step 5. Again these paths are found to be U-equivalent and now the paths $q_{0,3} \overset{b>c}{\Longrightarrow} q_{0,5}$ and $q_{1,4} \overset{b>c}{\Longrightarrow} q_{1,5}$ are declared to be C-equivalent.
7) Now the paths $q_{0,3} \overset{\neg(b>c)}{\Longrightarrow} q_{0,5}$ and $q_{1,4} \overset{\neg(b>c)}{\Longrightarrow} q_{1,5}$ are compared. It is found that they differ in the values of the variables g and z; however, g is no longer a live variable at $q_{0,5}$ and $q_{1,5}$–hence its value can be ignored.
8–9) With the propagated value of z, the pairs of paths $q_{0,5} \overset{x<y}{\Longrightarrow} q_{0,0}$ and $q_{1,5} \overset{x<y}{\Longrightarrow} q_{1,0}$, and $q_{0,5} \overset{\neg(x<y)}{\Longrightarrow} q_{0,0}$ and $q_{1,5} \overset{\neg(x<y)}{\Longrightarrow} q_{1,0}$ are declared to be U-equivalent and all the path pairs present in the LIST are declared to be C-equivalent.

B. Example of Dynamic Loop Scheduling

The DLS [9] transformation modifies the control structure of a behavior by introducing new branches while keeping all the loop feedback edges intact. As already mentioned in Section IV-B, the *LIST* has been introduced to detect crossing

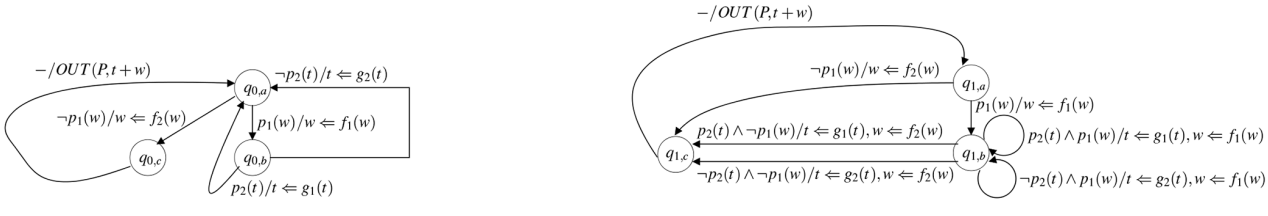

Fig. 6. Example of dynamic loop scheduling (DLS). (a) Original behavior M_0. (b) After DLS M_1.

over of loops. This is a deviation from [16] where back edges were used for detection of loops. The new method of loop detection aids in verifying DLS transformations [9] which [16] cannot handle. The example in Fig. 6 is used to illustrate verification of DLS transformations.

Example 5: The example given in Fig. 6(a) shows the original behavior and Fig. 6(b) shows its transformed version after application of DLS. Note that the states $q_{0,a}$, $q_{0,b}$ of FSMD M_0 and the states $q_{1,a}$, $q_{1,b}$ of FSMD M_1 qualify as cut-points. Initially, $q_{0,a}$ and $q_{1,a}$ are considered as corresponding states and the verification procedure then proceeds in the following sequence.

1) The path pair $\langle q_{0,a} \Longrightarrow q_{0,b},\ q_{1,a} \Longrightarrow q_{1,b}\rangle$ is declared as U-equivalent.
2) The path pair $\langle q_{0,a} \overset{\neg p_1}{\Longrightarrow} q_{0,a},\ q_{1,a} \overset{\neg p_1}{\Longrightarrow} q_{1,a}\rangle$ is declared as U-equivalent.
   Next, we consider the paths emanating from the corresponding states $q_{0,b}$ and $q_{1,b}$.
3) For the path $q_{0,b} \overset{p_2}{\Rightarrow} q_{0,a}$ in M_0, it is found that there are two paths in M_1, namely $q_{1,b} \overset{p_2 \wedge p_1}{\Longrightarrow} q_{1,b}$ and $q_{1,b} \overset{p_2 \wedge \neg p_1}{\Longrightarrow} q_{1,a}$, emanating from $q_{1,b}$ in M_1 whose conditions of execution are stronger than the path of M_0. This results in declaring the paths $q_{0,b} \overset{p_2}{\Rightarrow} q_{0,a}$ and $\eta_{q_{1,b}}$ as candidate C-equivalent and the path pair is stored in the *LIST*.
4) The paths $q_{0,a} \Longrightarrow q_{0,b}$ (with respect to the propagated vector $\langle p_2(t), \langle \mathbf{g_1(t)}, w\rangle\rangle$) and $q_{1,b} \overset{p_2 \wedge p_1}{\Longrightarrow} q_{1,b}$ (w.r.t. $\langle \top, \langle \mathbf{t}, w\rangle\rangle$) are now found to be U-equivalent since the condition of execution and the data transformation of the former path match with those of the latter. It is to be noted that after considering the latter path in M_1, one ends up in $q_{1,b}$ which is the final state of a path (specifically, $\eta_{q_{1,b}}$) that is already present in the *LIST*. However, since U-equivalence was established by the time this state is (re)visited, the equivalence of the paths can be ascertained, and the call to *loopInvariant* is no longer required.
5) First of all, note that when this step is executed, the invoked call to this instance of *correspondenceChecker* has $\{\langle q_{0,b} \overset{p_2}{\Rightarrow} q_{0,a}, \eta_{q_{1,b}}\rangle\}$ in the *LIST* as well. A similar U-equivalence is now established between the paths $q_{0,a} \overset{\neg p_1}{\Longrightarrow} q_{0,a}$ of M_0 and $q_{1,b} \overset{p_2 \wedge \neg p_1}{\Longrightarrow} q_{1,a}$ of M_1.
   Since eventually for all paths emanating from the final state of $q_{0,b} \overset{p_2}{\Rightarrow} q_{0,a}$, a U-equivalent path is discovered that originates from the final state of $\eta_{q_{1,b}}$, the path pair $\langle q_{0,b} \overset{p_2}{\Rightarrow} q_{0,a}, \eta_{q_{1,b}}\rangle$ is declared as *actually* C-equivalent, i.e., put in E_c and the *LIST* is rendered empty by removing its last (and only) entry. Note that unless both the paths from $q_{0,a}$, i.e., $q_{0,a} \overset{p_1}{\Rightarrow} q_{0,b}$ and $q_{0,a} \overset{\neg p_1}{\Longrightarrow} q_{0,a}$ are treated, *LIST* is not updated–as borne out by the fact that *LIST* update takes place after the loop 1 - 21 in *correspondenceChecker*.
6) The paths $q_{0,b} \overset{\neg p_2}{\Longrightarrow} q_{0,a}$ and $\eta_{q_{1,b}}$ are considered as candidate C-equivalent, similar to step 3.

7–8) The path pairs $\langle q_{0,a} \Longrightarrow q_{0,b}, q_{1,b} \overset{\neg p_2 \wedge p_1}{\Longrightarrow} q_{1,b}\rangle$ and $\langle q_{0,a} \overset{\neg p_1}{\Longrightarrow} q_{0,a}, q_{1,b} \overset{\neg p_2 \wedge \neg p_1}{\Longrightarrow} q_{1,a}\rangle$ are declared as U-equivalent, and $\langle q_{0,b} \overset{\neg p_2}{\Longrightarrow} q_{0,a}, \eta_{q_{1,b}}\rangle$ is declared as actually C-equivalent.

Now let us consider what would have gone wrong if back edges were used to detect loops instead of *LIST*. The first two steps of the method described in [16] would have been identical to the ones given above, and the states $q_{0,b}$ and $q_{1,b}$ would have been declared as corresponding states. In step 3, the method in [16] would find the paths $q_{0,b} \overset{p_2}{\Rightarrow} q_{0,a}$ and $\eta_{q_{1,b}}$ as candidate C-equivalent (similar to ours); however, it would also detect $q_{0,b} \overset{p_2}{\Rightarrow} q_{0,a}$ as a back edge to $q_{0,a}$. While entering the loop at $q_{0,a}$, the variable t had the symbolic value "t," but after step 3 the propagated vector computed for $q_{0,a}$ would have the value $g_1(t)$ for t. Clearly, there is a mismatch in the values for t, and therefore the method in [16] would terminate declaring the two FSMDs to be inequivalent. It is important to note that in case of code motions across loops, some paths will be detected to be C-equivalent while entering the loops. Hence, whenever the entry/exit state is revisited without resolving the mismatches, the state will definitely appear as a final state of some path in *LIST*, thereby leading to detection of loops (and without having any explicit knowledge about back edges).

The bisimulation-based method in [19] can handle the control structure modifications stated in the path-based scheduling method [8]. In general, during path-based scheduling, two consecutive paths get concatenated into one whose condition of execution is obtained by taking the conjunction of those of its constituent paths. However, unlike [8], DLS may introduce new branches by concatenating paths that lie within a loop with the "exit" path of that loop [such as the branches $q_{1,b} \xrightarrow{p_2 \wedge \neg p_1} q_{1,c}$ and $q_{1,b} \xrightarrow{\neg p_2 \wedge \neg p_1} q_{1,c}$ in Fig. 6(b)]. A primary requirement for the equivalence checking method of [19] is that the loop exit conditions must be identical in order to find whether two states are in a relation (a concept similar to our corresponding states); consequently, the method of [19] results in a *failure* on application of DLS. For a more sophisticated example of DLS, one may refer to [26],

TABLE I
VERIFICATION RESULTS BASED ON OUR SET OF BENCHMARKS

Benchmarks	Original FSMD		Transformed FSMD		#Variable		#across	Maximum	Time (ms)	
	#state	#path	#state	#path	com	uncom	loops	mismatch	PE	VP
BARCODE-1	33	54	25	56	17	0	0	3	20.1	16.2
DCT-1	16	1	8	1	41	6	0	6	6.3	3.6
DIFFEQ-1	15	3	9	3	19	3	0	4	5.0	2.6
EWF-1	34	1	26	1	40	1	0	1	4.2	3.6
LCM-1	8	11	4	8	7	2	1	4	–	2.5
IEEE754-1	55	59	44	50	32	3	4	3	–	17.7
LRU-1	33	39	32	38	19	0	2	2	–	4.0
MODN-1	8	9	8	9	10	2	0	3	5.6	2.5
PERFECT-1	6	7	4	6	8	2	2	2	–	0.9
QRS-1	53	35	24	35	25	15	3	19	–	15.9
TLC-1	13	20	7	16	13	1	0	2	9.1	4.1

which both the path extension method [10], [26] and the value propagation method described in this paper can handle.

VI. EXPERIMENTAL RESULTS

The verification procedure presented in this paper has been implemented in *C* on a 2.0 GHz Intel Core 2 Duo machine and satisfactorily tested on several benchmarks. The tool is available as the package annotated as Assorted formal equivalence checking programmes. The results are provided in Table I. Some of these benchmarks, such as BARCODE, LCM, QRS, and TLC, are control intensive; some are data intensive, such as DCT, DIFFEQ, and EWF, whereas some are both control and data intensive, such as IEEE754 and LRU. The transformed FSMD is obtained from the original one in multiple steps. First, we feed the original FSMD to the synthesis tool SPARK [21] to get an intermediate FSMD which is then converted into the (final) transformed FSMD manually according to a path-based scheduler [8] and accounting for induction variable elimination.[3] This multiple-step process helps us ascertain that our method can perform equivalence checking successfully when both control structure has been modified and code motions have occurred. It is to be noted that we prevent SPARK from applying loop shifting and loop unrolling transformations since they cannot be handled by our verifier presently. The column "#across loops" in Table I represents the number of code motions across loops; the "0" entries indicate that the transformed FSMDs contain no operation that has been moved across loop(s). However, the transformations do help in reducing the number of states (BARCODE, DCT, EWF, etc.) and (or) the number of paths (TLC). The column Maximum Mismatch displays the maximum number of mismatches found between two value-vectors for each benchmark. The run-times obtained by executing the benchmarks by our tool (VP) as well as by that of [10] (PE) show that the value propagation method takes less time. The dashes (–) in the column corresponding to PE represents that the tool exited with negative false results in those cases since it is unable to handle code motions across loops. Other than transformations like path merging/splitting and code motions across loops, the transformations that were applied to the original behaviors to produce the corresponding optimized behaviors include associative, commutative, distributive transformations, copy and constant propagation, common subexpression elimination, arithmetic expression simplification, partial evaluation, constant (un)folding, redundant computation elimination, etc. As demonstrated in Table I, the verification tool was able to establish equivalences in all the cases.

[3]It was mentioned in [16] that the benchmarks were first fed to the path-based scheduler and then to SPARK; it should be other way round as mentioned here.

TABLE II
VERIFICATION RESULTS BASED ON THE BENCHMARKS PRESENTED IN [10]

Benchmarks	PE (in ms)			VP (in ms)		
	BB-based	path based	SPARK	BB-based	path based	SPARK
BARCODE-2	12.1	15.4	19.8	10.1	13.1	12.0
DCT-2	3.3	3.1	3.3	2.3	2.6	2.7
DHRC-2	29.6	28.1	29.4	25.9	28.0	26.9
DIFFEQ-2	1.4	2.5	3.4	1.2	1.4	1.4
EWF-2	2.2	1.5	2.0	1.4	1.3	1.8
GCD-2	3.0	4.1	3.0	1.6	1.7	1.3
IEEE754-2	18.3	14.4	25.8	20.1	14.6	20.1
LRU-2	5.2	4.8	7.6	4.3	4.0	4.1
MODN-2	2.4	2.4	5.4	1.5	1.5	2.3
PERFECT-2	1.9	1.7	2.4	0.8	0.9	1.2
PRAWN-2	192.8	223.6	217.8	52.2	59.5	48.9
TLC-2	3.4	6.9	2.9	3.0	3.3	2.9

Table II gives a comparison of the execution times required by PE and VP for the benchmarks presented in [10] which involve no code motion across loops. Moreover, the transformed FSMDs considered in [10] have been obtained by subjecting the original FSMDs to a single compiler at a time. Although the source FSMDs for all the common benchmarks in Tables I and II are identical, the transformed FSMDs are not. Hence, to differentiate between the benchmark suites, the numerals 1 and 2 have been appended with the benchmark names. From the results it can be seen that for the benchmarks which are scheduled using a BB-based SAST [27] and the path-based scheduler, VP performs somewhat better than PE and more so for SPARK.

We further our investigation by subjecting the source codes to non-uniform code motions such as speculation, reverse speculation, safe speculation, etc. A recent work [12] addresses verification of such code motions by analyzing the data-flow of the programmes. This method resolves the decision of extending a path upon finding a mismatch for some variable, x say, by checking whether the mismatched value of x is used in some subsequent path or not before x is defined again. For the data-flow analysis, it constructs a Kripke structure from the given behavior, generates a Computation Tree Logic (CTL) formula to represent the def-use relation (for x) and employs the model checker NuSMV [28] to find whether that formula holds in the Kripke structure or not. Depending upon the output returned by the model checker, paths are extended

TABLE III
VERIFICATION RESULTS BASED ON THE BENCHMARKS PRESENTED IN [12]

Benchmarks	#BB	#Branch	#Path	#State		Maximum	Time (ms)	
				orig	trans	mismatch	PN	VP
BARCODE-3	28	25	55	32	29	5	3.00E+3	12.2
DHRC-3	7	14	31	62	47	5	1.62E+5	28.1
DIFFEQ-3	4	1	3	16	10	4	2.69E+2	4.0
FINDMIN8-3	14	7	15	8	9	7	1.00E+5	2.3
GCD-3	7	5	11	7	6	2	3.25E+2	2.1
IEEE754-3	40	28	59	55	42	6	3.40E+5	24.0
LRU-3	22	19	39	33	25	4	3.00E+3	4.7
MODN-3	7	4	9	6	5	4	5.41E+2	2.4
PARKER-3	14	6	13	12	10	6	6.00E+3	3.1
PERFECT-3	6	3	7	9	6	2	1.07E+2	1.1
PRAWN-3	85	53	154	122	114	8	5.81E+5	61.5
QRS-3	26	16	35	53	24	12	1.94E+5	15.7
TLC-3	17	6	20	13	13	3	2.45E+2	4.0
WAKA-3	6	2	5	9	12	6	1.00E+3	2.2

accordingly. However, in the presence of code motions across loop, the method of [12] fails due to the same reason as that of [10], i.e., prohibition of extension of paths beyond loops. For comparing the method of [12] with that of ours, another set of test cases have been constructed where the original behaviors have undergone both uniform and non-uniform code motions to produce the transformed behaviors. It is important to note that in none of these cases code motion across loop has been applied. The time taken by our tool and that of [12] (PN) are tabulated in Table III. The method of PN takes considerable amount of more time because it spends a large proportion of its execution time interacting with NuSMV through file handling. Although a comparative analysis with the method of [19] would have been relevant, we cannot furnish it since their tool is not available to us.

VII. CONCLUSION

Code optimization is a common process during the scheduling phase of high-level synthesis. This involves applications of transformations to the control structure of the code and often involves code motions across loops. It is important to determine the equivalence of the source code and the optimized code. In this paper, we have presented a value propagation-based equivalence checking method which not only handles control structure modifications but also code motions across loops apart from simpler uniform and non-uniform code motions without needing any supplementary information from the synthesis tools. Our method currently does not handle transformations such as loop unrolling, loop merging and loop shifting. Loop shifting is handled to some extent in [6] and [19] at the cost of termination; moreover, they cannot handle control structure transformations introduced by [9] which our method can. While there are several other techniques which determine equivalence in the presence of control structure modifications, there is only one technique [15] which handles code motions across loops but it requires additional information from the synthesis tools that is difficult to obtain in general. Note that while the method described in [15] captures infinite loops and computations that may fail, our method is capable of handling the former but needs some improvement to cover the latter. Correctness of value propagation as a method of equivalence checking and correctness of the overall equivalence checking procedure are proven formally. Complexity of the equivalence checking procedure has been analyzed mathematically. Our method has been tested successfully on several benchmarks, including those of [10] and [12]. The experimental results demonstrate that our mechanism can verify equivalence between two behaviors even when one of them has been subjected to successive code transformations using the SPARK high-level synthesis tool and a path-based scheduler. The results show that our method performs comparably with both these methods in terms of verification time required and outperforms them in terms of the type of transformations handled. Possible extensions of our work could be to enhance it for validation of several loop transformations such as loop unrolling, loop merging, loop shifting, and those aimed at optimizing memory hierarchy allocation [29].

APPENDIX A
CORRECTNESS OF VALUE PROPAGATION AS A METHOD OF EQUIVALENCE CHECKING

Theorem 2 (Correctness of the approach): An FSMD M_0 with no unreachable state[4] is contained in another FSMD M_1 ($M_0 \sqsubseteq M_1$), if
for a finite path cover $P_0 = \{p_{0,1}, \ldots, p_{0,l_0}\}$ of M_0, there exists a path cover $P_1 = \{p_{1,1}, \ldots, p_{1,l_1}\}$ of M_1, such that:

1) each path of P_0 is either conditionally or unconditionally equivalent to some path of P_1 satisfying the correspondence relation of the respective start states and final states of the paths; thus, symbolically, $\forall i, 1 \leq i \leq l_0$:
   a) $p_{0,i} \simeq p_{1,j}$ or $p_{0,i} \simeq_c p_{1,j}$, and $(p^s_{0,i}, p^s_{1,j})$ belongs to the set of corresponding (C-corresponding) state pairs, for some j, $1 \leq j \leq l_1$;
   b) $p_{0,i} \simeq_c \eta$, a null path of M_1, and $(p^s_{0,i}, \eta^s)$ belongs to the set of corresponding (C-corresponding) state pairs;
2) all paths of P_0 leading to the reset state will have unconditionally equivalent paths in P_1 leading to the reset state of M_1; thus, symbolically, $\forall p_{0,k_0} \in P_0$ such that p^f_{0,k_0} is the reset state $q_{0,0}$ of M_0, $\exists p_{1,k_1} \in P_1$ such that p^f_{1,k_1} is the reset state $q_{1,0}$ of M_1, and $p_{0,k_0} \simeq p_{1,k_1}$.

Proof: A path p_2 is said to be consecutive to p_1 if $p^f_1 = p^s_2$. Since P_0 is a path cover of M_0, a computation μ_0 of M_0 can be viewed as a concatenation of *consecutive* paths starting and ending at the reset state of M_0; symbolically, $\mu_0 = [p_{0,i_1}, p_{0,i_2}, \ldots, p_{0,i_n}]$, where $p_{0,i_j} \in P_0, 1 \leq j \leq n$, and $p^s_{0,i_1} = p^f_{0,i_n} = q_{0,0}$. From the hypotheses 1 and 2 of the theorem, it follows that there exists a sequence S of paths $[p_{1,k_1}, p_{1,k_2}, \ldots, p_{1,k_n}]$, where $p_{1,k_j} \in P_1$ or is a null path of M_1, $1 \leq j < n$ and $p_{1,k_n} \in P_1$, such that $p_{0,i_l} \simeq p_{1,k_l}$ or $p_{0,i_l} \simeq_c p_{1,k_l}$, $1 \leq l < n$ and $p_{0,i_n} \simeq p_{1,k_n}$. For S to represent a computation of M_1, it must be a concatenation of consecutive paths in M_1 starting from the reset state $q_{1,0}$ back to itself, which may not be the case because $p^f_{1,k_j} \neq p^s_{1,k_{j+1}}$ is possible when the path joining p^f_{1,k_j} and $p^s_{1,k_{j+1}}$ is C-equivalent to a null path in M_0 (and hence is not present in S). Introduction of null paths at appropriate places does not alter the computation μ_0

[4]For taking care of useless paths, it suffices to check whether or not some cut-point is reachable from the reset state or not (using dfs or bfs).

since null paths have conditions of execution *true* and null data transformations (by definition) and preserves consecutiveness property. Let μ_0' be a sequence obtained from μ_0 by introducing such null paths at the appropriate places which have C-corresponding paths in M_1. The sequence yields a computation equivalent to μ_0. The sequence S' of paths of M_1 corresponding to μ_0' can be obtained by introducing at appropriate places in S the paths of M_1 which are C-corresponding to the null paths of M_0 introduced in μ_0 to obtain μ_0'. This new sequence S' now indeed represents a concatenated path that starts and ends at the reset state $q_{1,0}$. Furthermore, hypothesis 2 of the theorem implies that whatever mismatches might be present in the respective last but one paths in μ_0' and S', they must get resolved when the respective last paths back to the reset states are traversed. Let μ_1 be the computation of M_1 represented by sequence S'. Thus, the computations μ_1 and μ_0', and hence μ_0, must be equivalent. ■

APPENDIX B
CORRECTNESS AND COMPLEXITY OF THE EQUIVALENCE CHECKING PROCEDURE

A. Correctness

Theorem 3 (Partial correctness): If the verification method terminates in step 8 of the function *containmentChecker*, then $M_0 \sqsubseteq M_1$.

Proof: The proof is tantamount to ascertaining that if the verification method terminates in step 8 of *containmentChecker*, then both hypotheses 1 and 2 of theorem 2 are satisfied by the path covers P_0 and P_1 for the FSMDs M_0 and M_1 respectively. The path covers P_0 and P_1 comprise of paths starting from and ending in cut-point(s) without having any intermediary cut-point. The set E_u of U-equivalent paths and E_c of C-equivalent paths are updated in steps 17 and 22, respectively, of *correspondenceChecker*. The fact that the pair of paths added to the set E_u and E_c are indeed U-equivalent and C-equivalent, respectively, is affirmed by the function *findEquivalentPath*.

Now, we need to prove that $E = E_u \bigcup E_c$ contains a member for each path in P_0. Suppose a path β exists in P_0 that does not have a corresponding member in E. Absence of β in E indicates that the path has not been considered at all during execution of the verification method. Since the state β^s is reachable (by definition) there must be some other path β' that leads to it. Let us consider the following cases.

$\beta' \in E$: Then E must contain some member of the form $\langle \beta', \alpha' \rangle$, where $\alpha' \in P_1$ and either (i) $\beta' \simeq_c \alpha'$ which means β would definitely have been considered in some subsequent recursive call of *correspondenceChecker*, or (ii) $\beta' \simeq \alpha'$ which means the end state of β', i.e., the start state of β, must be a member of ζ and β must have eventually been considered as given in step 4 of *containmentChecker* (contradiction).

$\beta' \notin E$: In such a case, one should consider the path β'' that leads to β'. Now, these two cases hold for β'' as well. A repetitive application of the argument lands up in the paths emanating from the reset state $q_{0,0}$, which is a member of ζ by step 2 of *containmentChecker*; here steps 3 and 4 of *containmentChecker* ensure invocation of *correspondenceChecker* with $q_{0,i} = q_{0,0}$ and step 1 of *correspondenceChecker* ensures that paths from $q_{0,0}$ must have been treated, thereby again leading to a contradiction.

What remains to be proved is that hypothesis 2 of theorem 2 holds when the verification method terminates in step 8 of *containmentChecker*. Suppose it does not. Then there exist paths $\beta \in P_0$ and $\alpha \in P_1$ such that $\beta^f = q_{0,0}$, $\alpha^f = q_{1,0}$ and $\beta \simeq_c \alpha$. It would result in trying to extend a path beyond the reset states, and thus, the function *correspondenceChecker* returns failure to *containmentChecker* as shown in step 10, whereupon the latter terminates in step 5 and not in step 8 (contradiction). ■

Theorem 4 (Termination): The verification method always terminates.

Proof: With respect to the call graph of Fig. 4, we prove the termination of the modules in a bottom-up manner. The functions *valuePropagation* and *loopInvariant* obviously execute in finite time since the former involves comparison of two path characteristics and two propagated vectors, whereas the latter involves comparison of two propagated vectors only. The *for*-loop in *findEquivalentPath* is executed only $\|P_1\|$ number of times which is finite. The outermost *for*-loop in the function *correspondenceChecker* can be executed $\|P_0\|$ times which is also finite. Whenever *correspondenceChecker* is invoked with the states $q_{0,i}$ and $q_{1,j}$ as arguments, there is a possibility that the function is recursively invoked with the end states of some path β originating from $q_{0,i}$ and some path α originating from $q_{1,j}$. Now, it remains to be shown that *correspondenceChecker* can invoke itself recursively only finite number of times. Let us associate the tuple $\langle n_0, n_1 \rangle \in \mathbb{N}^2$ to each invocation of *correspondenceChecker*($\beta_m^f, \alpha_m^f, \ldots$) (step 14), where n_0 and n_1 denote the number of paths that lie ahead of β_m^f and α_m^f in FSMD M_0 and FSMD M_1 respectively. (Note that one cannot go beyond the reset states.) Also, let $\langle n_0', n_1' \rangle < \langle n_0, n_1 \rangle$ if $n_0' < n_0$ or, $n_0' = n_0$ and $n_1' < n_1$. Note that in the 4-tuple returned by *findEquivalentPath* in step 3 of *correspondenceChecker*, β_m and α both cannot be null paths simultaneously. Hence, a sequence of recursive invocations of *correspondenceChecker* can be represented by a sequence of these tuples $\langle n_0, n_1 \rangle > \langle n_0', n_1' \rangle > \langle n_0'', n_1'' \rangle$, and so on; specifically, this is a strictly decreasing sequence. Since $\mathbb{N}^2$ is a well-founded set [14] of pairs of nonnegative numbers having no infinite decreasing sequence, *correspondenceChecker* cannot call itself recursively infinite times. The only loop in *containmentChecker* depends upon the size of ζ, the set of corresponding states. Note that *correspondenceChecker* is called in this loop for every member of ζ only once. Since the number of states in both the FSMDs is finite, the number of elements in ζ has to be finite. ■

B. Complexity

The complexity of the overall verification method is in the order of product of the following two terms: (i) the complexity of finding a U-equivalent or a C-equivalent path for a given path from a state, and (ii) the number of times we need to find such a path. The first term is the same as the complexity of *findEquivalentPath*($\beta, q_{1,j}, \cdots$) which tries to find a path α starting from $q_{1,j} \in M_1$ such that $\alpha \simeq \beta$ or $\alpha \simeq_c \beta$. Let the number of states in M_1 be n and the maximum number of parallel edges between any two states be k. Therefore, the maximum possible state transitions from a state are $k \cdot n$. The condition of execution associated with each transition emanating from a state is distinct. The function checks all transitions

from $q_{1,j}$ in the worst case. Note that the conditions of execution and the data transformations of the paths are stored as normalized formulas [23] by the function *containmentChecker*. If $\|F\|$ be the length of the normalized formula (in terms of the number of variables along with that of the operations in F), then the complexity of normalization of F is $O(2^{\|F\|})$ due to multiplication of normalized sums. As such, all the paths in the FSMDs will have their data transformations and conditions of execution computed during the initialization steps in the function *containmentChecker*. However, the function *findEquivalentPath* needs to substitute the propagated values in these entities necessitating multiplication. Hence, the complexity of finding a U-equivalent or a C-equivalent path is $O(k \cdot n \cdot 2^{\|F\|})$. On finding a C-equivalent path, value propagation is carried out in $O(2^{\|F\|} \cdot |V_0 \cup V_1|)$ time. So, the overall complexity is $O(2^{\|F\|} \cdot (k \cdot n + |V_0 \cup V_1|))$.

The second term is given by the product of (i) the number of times *correspondenceChecker* is called from *containmentChecker*, which is the same as the size of the set of corresponding states ζ, and (ii) the number of recursive calls made to *correspondenceChecker* (in the worst case). We notice that if the number of states for the original FSMD is n, then the number of states for the transformed FSMD is in $O(n)$ for both path-based scheduler [8] and SPARK [21]. So, for simplicity, let the number of states in M_1 be n as well. In the worst case, all the states of M_0 may be cut-points and the number of paths in M_0 is at most $k \cdot n \cdot (n-1)/2$. In this case the *correspondenceChecker* can recursively call itself as many as $(n-1)$ times leading to consideration of $k \cdot (n-1) + k^2 \cdot (n-1) \cdot (n-2) + \cdots + k^{(n-1)} \cdot (n-1) \cdot (n-2) \cdots 2 \cdot 1 \simeq k^{(n-1)} \cdot (n-1)^{(n-1)}$ number of paths. Also, $\|\zeta\| \leq n$. Therefore, the complexity of the overall method is $O((k \cdot n + |V_0 \cup V_1|) \cdot 2^{\|F\|} \cdot n \cdot k^{(n-1)} \cdot (n-1)^{(n-1)})$ in the worst case. It is important to note that in [12], the authors had neglected the time complexity of computing the path characteristics of the concatenated paths that result from path extensions. Upon considering the same, the worst case time complexity of the presented method is found to be identical to that of [12].

ACKNOWLEDGMENT

The authors would like to thank the anonymous reviewers' comments toward improvement of the paper.

REFERENCES

[1] S. Gupta, N. Dutt, R. Gupta, and A. Nicolau, "Using global code motions to improve the quality of results for high-level synthesis," *Trans. Computer-Aided Design Integr. Circuits Syst.,* vol. 23, no. 2, pp. 302–312, Feb. 2004.

[2] L. C. V. dos Santos and J. Jress, "A reordering technique for efficient code motion," in *Proc. DAC*, New Orleans, LA, USA, 1999, pp. 296–299.

[3] G. Lakshminarayana, A. Raghunathan, and N. Jha, "Incorporating speculative execution into scheduling of control-flow-intensive design," *Trans. Computer-Aided Design Integr. Circuits Syst.*, vol. 19, no. 3, pp. 308–324, Mar. 2000.

[4] O. Ruthing, J. Knoop, and B. Steffen, "Sparse code motion," in *Proc. IEEE POPL*, Boston, MA, USA, 2000, pp. 170–183.

[5] J. Knoop, O. Ruthing, and B. Steffen, "Lazy code motion," in *Proc. PLDI*, San Francisco, CA, USA, 1992, pp. 224–234.

[6] S. Kundu, S. Lerner, and R. Gupta, "Validating high-level synthesis," in *Proc. CAV*, Princeton, NJ, USA, 2008, pp. 459–472.

[7] S. Kundu, S. Lerner, and R. Gupta, "Translation validation of high-level synthesis," *Trans. Computer-Aided Design Integr. Circuits Syst.*, vol. 29, no. 4, pp. 566–579, Apr. 2010.

[8] R. Camposano, "Path-based scheduling for synthesis," *Trans. Computer-Aided Design Integr. Circuits Syst.*, vol. 10, no. 1, pp. 85–93, Jan. 1991.

[9] M. Rahmouni and A. A. Jerraya, "Formulation and evaluation of scheduling techniques for control flow graphs," in *Proc. EURO-DAC*, Brighton, U.K., 1995, pp. 386–391.

[10] C. Karfa, D. Sarkar, C. Mandal, and P. Kumar, "An equivalence-checking method for scheduling verification in high-level synthesis," *Trans. Computer-Aided Design Integr. Circuits Syst.*, vol. 27, no. 3, pp. 556–569, Mar. 2008.

[11] C.-H. Lee, C.-H. Shih, J.-D. Huang, and J.-Y. Jou, "Equivalence checking of scheduling with speculative code transformations in high-level synthesis," in *Proc. ASP-DAC*, Yokohama, Japan, 2011, pp. 497–502.

[12] C. Karfa, C. A. Mandal, and D. Sarkar, "Formal verification of code motion techniques using data-flow-driven equivalence checking," *ACM Trans. Design Autom. Electron. Syst.*, vol. 17, no. 3, Article 30, Jun. 2012.

[13] R. W. Floyd, "Assigning meaning to programs," in *Proc. 19th Symp. Appl. Math.*, 1967, pp. 19–32.

[14] Z. Manna, *Mathematical Theory of Computation*. New York, NY, USA: McGraw-Hill, 1974.

[15] J.-B. Tristan and X. Leroy, "Verified validation of lazy code motion," in *Proc. PLDI*, Dublin, Ireland, 2009, pp. 316–326.

[16] K. Banerjee, C. Karfa, D. Sarkar, and C. Mandal, "A value propagation based equivalence checking method for verification of code motion techniques," in *Proc. ISED*, Kolkata, India, 2012, pp. 67–71.

[17] Y. Kim, S. Kopuri, and N. Mansouri, "Automated formal verification of scheduling process using finite state machines with datapath (FSMD)," in *Proc. ISQED*, Washington, DC, USA, 2004, pp. 110–115.

[18] Y. Kim and N. Mansouri, "Automated formal verification of scheduling with speculative code motions," in *Proc. GLSVLSI*, Orlando, FL, USA, 2008, pp. 95–100.

[19] T. Li, Y. Guo, W. Liu, and M. Tang, "Translation validation of scheduling in high level synthesis," in *Proc. GLSVLSI*, Paris, France, 2013, pp. 101–106.

[20] X. Leroy *et al.* (2013, Aug. 20). *The CompCert C Compiler* [Online]. Available: http://compcert.inria.fr/compcert-C.html

[21] S. Gupta, N. Dutt, R. Gupta, and A. Nicolau, "SPARK: A high-level synthesis framework for applying parallelizing compiler transformations," in *Proc. VLSI Design*, 2003, pp. 461–466.

[22] D. D. Gajski, N. D. Dutt, A. C. Wu, and S. Y. Lin, *High-Level Synthesis: Introduction to Chip and System Design*. Boston, MA, USA: Kluwer Academic Publishers, 1992.

[23] D. Sarkar and S. De Sarkar, "A theorem prover for verifying iterative programs over integers," *IEEE Trans. Softw. Eng.*, vol. 15, no. 12, pp. 1550–1566, Dec. 1989.

[24] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Princiles, Techniques, and Tools*. Stanford, CA, USA: Pearson Education, 2006.

[25] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 2nd ed. Cambridge, MA, USA: MIT Press, 2001.

[26] C. Karfa, C. A. Mandal, D. Sarkar, S. R. Pentakota, and C. Reade, "A formal verification method of scheduling in high-level synthesis," in *Proc. ISQED*, San Jose, CA, USA, 2006, pp. 71–78.

[27] C. A. Mandal and R. M. Zimmer, "A genetic algorithm for the synthesis of structured data paths," in *Proc. VLSI Design*, Calcutta, India, 2000, pp. 206–211.

[28] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri, "NuSMV: A new symbolic model checker," *Int. J. STTT*, vol. 2, no. 4, pp. 410–425, 2000.

[29] J. Cong, P. Zhang, and Y. Zou, "Optimizing memory hierarchy allocation with loop transformations for high-level synthesis," in *Proc. DAC*, San Francisco, CA, USA, 2012, pp. 1233–1238.

Kunal Banerjee, photograph and biography not available at the time of publication.

Chandan Karfa, photograph and biography not available at the time of publication.

Dipankar Sarkar, photograph and biography not available at the time of publication.

Chittaranjan Mandal, photograph and biography not available at the time of publication.

See the paper entitled "Verification of Loop and Arithmetic Transformations of Array-Intensive Behaviors," TCAD vol. 32, no. 11, page 1800 for authors' photographs and short biographies.